Cost- and Workload-Driven Data Management in the Cloud

Inauguraldissertation

zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Ilir Fetai
aus Oberlangenegg, Bern

Basel, 2016
Originaldokument gespeichert auf dem Dokumentenserver der Universität Basel
edoc.unibas.ch

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät

auf Antrag von

Prof. Dr. Heiko Schuldt, Universität Basel, Dissertationsleiter
Prof. Dr. Norbert Ritter, Universität Hamburg, Korreferent

Basel, den 20.09.2016

Prof. Dr. Jörg Schibler, Dekan

To my family

Zusammenfassung

Die vorliegende Arbeit widmet sich einer zentralen Aufgabenstellung, die im Umfeld verteilter Datenverwaltung in der Cloud auftritt: Das richtige Gleichgewicht finden im Spannungsfeld zwischen Verfügbarkeit, Konsistenz, Latenz und Kosten, das vom CAP/PACELC erfasst wird. Im Zentrum der Arbeit steht die Entwicklung der kosten- und workload-basierten Konsistenz-, Partitionierungs- und Quorum-Protokolle, die gemeinsam als CCQ-Protokolle bezeichnet werden. Erstens, die Entwicklung von C^3, welches ein adaptierbares Konsistenzprotokoll ist. Die wessentliche Eigenschaft von C^3 ist es, die optimale Konsistenzstufe zur Laufzeit zu bestimmen unter der Betrachtung der Konsistenz- und Inkonsistenz-Kosten. Zweitens, die Entwicklung von Cumulus, das in der Lage ist dynamisch die Partitionierung der Daten an die Last der Anwendungen anzupassen. Das Ziel von Cumulus ist es, verteilte Transaktionen zu vermeiden, da diese einen hohen Kosten- und Performance-Aufwand mit sich bringen. Und drittens, die Entwicklung von QuAD. QuAD ist ein quorum-basiertes Replikationsprotokoll, welches es erlaubt die Quoren in einem voll-replizierten Datenbanksystem dynamisch zu bestimmen, mit dem Ziel die bestmögliche Performance zu erreichen.

Das Verhalten der CCQ-Protokolle wird durch realitätsnahe Kostenmodelle gesteuert, die das Ziel der Kostenminimierung für die Sicherstellung der gewünschten Garantien verfolgen. Das Verhalten der Protokolle wird ständig beurteilt, und gegebenfalls angepasst basierend auf den Kostenmodellen, und unter Betrachtung der Anwendungslast. Diese Eigenschaft der Adaptierbarkeit ist entscheidend aus Sicht der Anwendungen, welche in der Cloud betrieben werden. Diese Anwendungen zeichnen sich aus durch eine hohe dynamische Last, und müssen gleichzeitig hochverfügbar und skalierbar sein.

Die Adaptierbarkeit zur Laufzeit kann erhebliche Kosten verursachen, die den Nutzen der Adaptierbarkeit übersteigen. Um dagegen zu wirken, wurde ein Kontrollmechanismus in die CCQ-Kostenmodelle integriert. Der Mechanismus stellt sicher, dass das Verhalten der Protokolle nur dann angepasst wird, wenn dadurch ein signifikanter Nutzen für die Anwendung entsteht.

Um die praktische Anwendbarkeit der CCQ-Protokolle untersuchen zu können, wurden diese in einem prototypischen Datenbanksystem implementiert. Die Modularität der Protokolle ermöglicht die nahtlose Erweiterung der Optimierungsmöglichkeiten mit geringem Aufwand.

Schlussendlich bietet diese Arbeit eine quantitative Evaluierung der Protokolle mittels einer Reihe von Experimenten unter realistischen Bedingungen in der Cloud. Die Ergebnisse bestätigen die Umsetzbarkeit der Protokolle, und deren Fähigkeit die Anwendungskosten zu reduzieren. Darüber hinaus demonstrieren sie die dynamische Adaptierbarkeit der Protokolle ohne die Korrektheit des Systems zu verletzen.

Abstract

This thesis deals with the challenge of finding the right balance between consistency, availability, latency and costs, captured by the CAP/PACELC trade-offs, in the context of distributed data management in the Cloud. At the core of this work, cost and workload-driven data management protocols, called CCQ protocols, are developed. First, this includes the development of C^3, which is an adaptive consistency protocol that is able to adjust consistency at runtime by considering consistency and inconsistency costs. Second, the development of Cumulus, an adaptive data partitioning protocol, that can adapt partitions by considering the application workload so that expensive distributed transactions are minimized or avoided. And third, the development of QuAD, a quorum-based replication protocol, that constructs the quorums in such a way so that, given a set of constraints, the best possible performance is achieved.

The behavior of each CCQ protocol is steered by a cost model, which aims at reducing the costs and overhead for providing the desired data management guarantees. The CCQ protocols are able to continuously assess their behavior, and if necessary to adapt the behavior at runtime based on application workload and the cost model. This property is crucial for applications deployed in the Cloud, as they are characterized by a highly dynamic workload, and high scalability and availability demands.

The dynamic adaptation of the behavior at runtime does not come for free, and may generate considerable overhead that might outweigh the gain of adaptation. The CCQ cost models incorporate a control mechanism, which aims at avoiding expensive and unnecessary adaptations, which do not provide any benefits to applications.

The adaptation is a distributed activity that requires coordination between the sites in a distributed database system. The CCQ protocols implement safe online adaptation approaches, which exploit the properties of 2PC and 2PL to ensure that all sites behave in accordance with the cost model, even in the presence of arbitrary failures. It is crucial to guarantee a globally consistent view of the behavior, as in contrary the effects of the cost models are nullified.

The presented protocols are implemented as part of a prototypical database system. Their modular architecture allows for a seamless extension of the optimization capabilities at any level of their implementation.

Finally, the protocols are quantitatively evaluated in a series of experiments executed in a real Cloud environment. The results show their feasibility and ability to reduce application costs, and to dynamically adjust the behavior at runtime without violating their correctness.

Acknowledgements

I am deeply grateful to my advisor, Prof. Dr. Heiko Schuldt, for giving me the opportunity to obtain a Ph.D, for the many excellent ideas he provided, and patiently guiding me to the correct path. Without his guidance, encouragements, and help, I could never finish my thesis.

I wish to thank my colleges of the DBIS group, especially Filip, Nenad, Ihab and Ivan for the great time we had, and for many valuable discussions from the very beginning.

Many master and bachelor students have contributed to the development of ClouD-Man, especially Alexander Stiemer, Damian Murezzan and Daniel Kohler. I had a great time working with such highly motivated and talented students.

A special thanks goes to my parents Menduale and Imer, and to my sister Nermine for everything they gave to me in the life, their encouragement and great support. My nephew Loni, and my niece Rina, have always been a source of energy during the writing of this thesis.

My biggest gratitude goes to my wife, Samije about her great love and unconditional support, and unending patience during my Ph.D studies.

And finally, I would like to thanks my one and only Ylli, for making me the happiest person in the world and patiently waiting every night to write yet another sentence in my thesis before telling him a good-night story.

Contents

List of Figures

List of Tables

1

Introduction

CLOUD COMPUTING is a term coined to describe the delivery of on-demand resources to customers. Cloud resources refer to the applications, the hardware, and the system software delivered as services. The idea of the Cloud is not new [Bir12, ÖV11]. It is based on a combination of well-known models, such as Service Oriented Architectures (SOA), virtualization, self-* properties of systems, and others, and has been made popular by companies, such as Amazon, Google or Microsoft, that started offering their existing resources and services to customers. The resources delivered by the Cloud can be divided into three main categories. 1.) Infrastructure-as-a-Service (IaaS) that consists of infrastructure services, such as computing power, storage resources, etc. Examples include Amazon Web Services (AWS), Elastic Compute Cloud (EC2) [awsa] and Azure [azu]. 2.) Platform-as-a-Service (PaaS) that consists of platform services, such as development tools and environments. Examples include AWS Elastic Beanstalk [awsb] and Google App Engine [app]. 3.) Software-as-a-Service (SaaS) that consists of application software delivered as a service. Examples of popular SaaS include Salesforce [Sal], Google Apps [Goo] and others. Cloud Computing is emerging towards a model that supports *everything-as-a-service* (XaaS), such as Data-as-a-Service (DaaS) [WTK+08], Archiving-as-a-Service (AaaS) [BS15a], etc.

The Cloud has considerably reduced the barrier for building large-scale applications, such as Instagram [ins], Foursquare [fou], etc., which are characterized by a massive number of users and their high scalability and availability demands [Aba09]. Availability denotes the percentage of time a system is functioning according to its specification, whereas scalability denotes the ability of a system to cope with an increasing load [Aba09].

These demands inherently lead to distributed systems that are complex and costly in terms of both the components needed for building them and the expertise required for their management. Prior to the Cloud, application providers had to conduct considerable upfront investments for building the necessary deployment infrastructure without having validated the market success of their applications.

With the *pay-as-you-go* cost model of the Cloud, application providers pay only for the resources they consume and can thus avoid the upfront investments that are considered a business disabler. The entire complexity and risk of building and managing

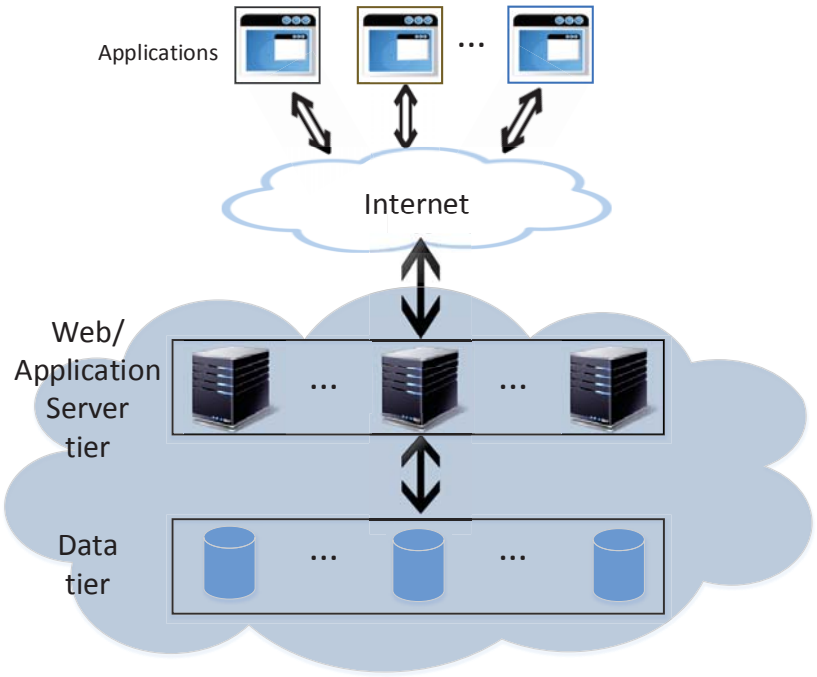


Figure 1.1: Software stack of applications deployed in the Cloud [Aba09].

expensive distributed infrastructures shifts to the Cloud provider [GG09,ÖV11], as they can amortize the costs across many customers – a property known as the *economy of scale* [KK10].

The *elasticity* property of Cloud services allows for their dynamic tailoring to the application demands by provisioning and de-provisioning these services based on for example application load, failures, etc. It supports the *scale-out*, also known as horizontal scalability, of applications, by allowing them to add commodity servers on demand and use them in the load distribution. Scale-out is critical for Cloud applications, as, in contrast to *scale-up* (also known as vertical scalability), it does not require any hardware updates, which considerably reduces the reaction time. Moreover, *scale-up* requires high-end hardware that is more expensive compared to commodity hardware as the economy of scale does not apply to the same degree [McW08].

1.1 Data Management in the Cloud: Challenges and Opportunities

The typical software stack of applications deployed in the Cloud is depicted in Figure 1.1. The data tier consists of the data denoted as the *Database (DB)*, and the software managing the data denoted as the *DBMS*. The DB and DBMS are jointly referred to as *Database System (DBS)* [ÖV11]. In the context of this thesis, we only consider applications that use a DBS, which is almost always the case in practice. It is crucial to provide high availability and scalability guarantees not only at the application and web

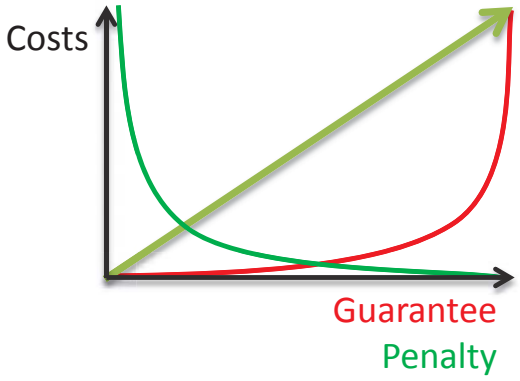


Figure 1.2: Guarantee vs. penalty costs.

tier, but also at the data tier, as in contrary the guarantees of the entire application may degrade [Aba09, DAE10, KK10]. Moreover, from a performance point of view, the DBS usually becomes the bottleneck [DAE10, YSY09].

To satisfy the availability demands applications have towards the data tier, Cloud providers usually replicate and distribute data across sites in different data-centers [Aba09]. Such an approach, also known as *geo-replication*, is crucial not only with regards to the availability but also for the performance as it allows to place the data in a geographical location close to the user. Moreover, the additional resources in a *Ditributed Database System (DDBS)* can be used to distribute the load and thus increase the scalability of the system. *Elasticity*[1] denotes the ability of the Cloud to scale resources out and down dynamically based on, for example, application load. Scale down is an important aspect as it allows to save costs by undeploying unnecessary resources. While elasticity is nicely applicable to the web and the application tier (Figure 1.1), implementing elastic DBSs is challenging as it requires the transfer (copying) of a possibly considerable amount of data over the network.

DDBS face the trade-offs captured by the CAP-theorem[2], which states that any distributed system can provide two of the three properties, namely consistency, availability, and partition-tolerance. The CAP-theorem was turned into a formal definition and proved in [GL02]. Distributed systems cannot sacrifice tolerance to network partitions as that would require to run on an absolutely reliable network [Hal10]. Thus, they are left with the choice of consistency or availability [Vog09, DHJ+07, Simb, Cas, Mon].

While CAP is about failures, there are also consistency-latency trade-offs during normal operations [Aba12]. The stronger the consistency model the higher the performance

[1] While scalability is a static property that guarantees the ability of a system to scale to a large number of users, elasticity is a dynamic property, which enables a system to scale out and down at runtime with possibly no downtime [Aba09].

[2] Notice that the CAP trade-off was already observed in [JG77]: "Partitioning - When communication failures break all connections between two or more active segments of the network ... each isolated segment will continue ... processing updates, but there is no way for the separate pieces to coordinate their activities. Hence ... the database ... will become inconsistent. This divergence is unavoidable if the segments are permitted to continue general updating operations and in many situations it is essential that these updates proceed." [BD].

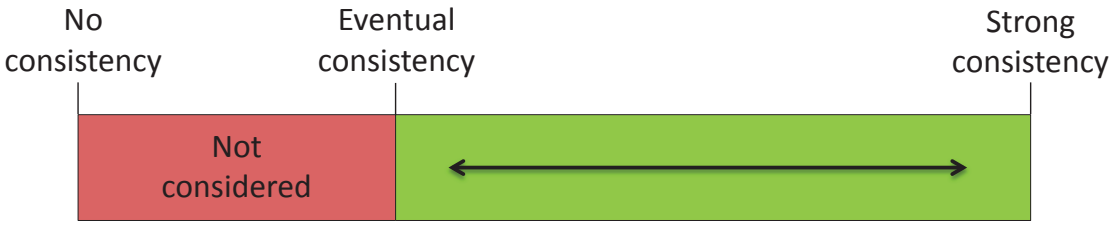


Figure 1.3: Continuum of consistency models.

penalty and vice-versa. The intuition is that strong consistency models require more coordination than weaker models [BG13, Hal10]. The consistency-performance trade-off is valid also for single copy-systems, which is one of the reasons why commercial databases have ever since included weaker models [BFG+13, BDF+13]. However, with the deployment of large-scale applications in the Cloud, which have stronger emphasis on performance, this trade-off has become even more tangible.

The fine-grained pricing of resources and actions in the Cloud allows charging customers based on the *pay-as-you-go* cost model. There is a trade-off between the desired level of guarantees and the costs. Increasing the level of guarantees also increases the costs and vice-versa. Applications require the DDBS to be *cost-driven*, i.e., they should consider the costs that incur as the consequence of enforcing the desired guarantees – *guarantee costs*. The guarantee costs should determine not only the most appropriate level of guarantee but also the most optimal approach in terms of costs for providing the chosen level of guarantee, as there might be more than one approach that leads to the same level of guarantee. They all are more suited for a certain scenario and less for others, i.e., generate different costs depending on the concrete scenario. Decreasing the level of guarantees may reduce guarantee costs. However, it might lead to a long-term business impact as a reduced guarantee level may directly affect the end-user behavior. For example, frequent unavailability of an application may lead to a loss of customers. The effect of reduced guarantees can be captured by application specific *penalty costs*. The DDBS should find the right balance between the guarantee and penalty costs, and it should provide that level of guarantees that incurs the minimal overall costs (Figure 1.2).

1.2 Problem Statement

The choice of the guarantees, and the protocols implementing those guarantees should be influenced by the application requirements, its properties, such as the workload, and the costs (Figure 1.2). Existing DBSs are over-customized and earmarked with a focus on a very specific application type and provide only a limited means of influencing their behavior. For example, commercial DBSs, such as Oracle [oraa], and Postgres [siP], provide strong data consistency, such as One-Copy Serializability (1SR) or Snapshot Isolation (SI), and thus take limited scalability and availability into account [GHOS96]. The consistency models can be implemented by different *protocols* [BG84, KA00a]. Usually

these protocols are tailored to a specific scenario and neglect crucial parameters, such as the application workload. Moreover, they are rigid and not able to adapt their behavior at runtime.

Not only SQL (NoSQL) is a term coined to denote a new type of DBSs that are non-relational, schema-less, that support horizontal scalability and provide high-availability [SF12]. NoSQL databases provide only relaxed consistency guarantees known as *Eventual Consistency (EC)* in order to increase scalability and availability as a consequence of the CAP theorem. However, while this nicely serves several novel types of applications, it has turned out that it is very challenging to develop applications that demands strong consistency on top of weakly consistent DBSs [Ham10].

In summary, existing databases have the following limitations. First, they provide only a static choice of the desired guarantees as well as only a limited means of steering these guarantees by applications. Second, they consider only specific protocols and protocol behavior (configuration) and do not include any means of dynamically reconfiguring the protocols at runtime. This means that these protocols are rigid and unadaptable to, for example, shifting application workload. Third, they neglect the costs incurring from the resources needed to provide the desired guarantees, which is a critical factor for the applications deployed in the Cloud. Therefore, novel concepts and solutions are needed that can cope with the dynamic nature of applications deployed in Cloud, and that do consider the costs as a first-class-citizen of their optimization models.

This thesis considers the development of adaptive[3], *cost* and *workload-driven* protocols for data consistency, partitioning, and replication. Our protocols can adapt dynamically (reconfigure) their behavior to shifting application workload based on cost models that find the right balance between guarantee, penalty and reconfiguration costs. Moreover, they incorporate online reconfiguration approaches that guarantee consistency even in the presence of failures. The cost models ensure that the adaptations only happen within the boundaries specified by the application providers, i.e., do not violate in any case the desired guarantees.

1.2.1 Data Consistency

Initially, NoSQL databases provided only weak consistency, such as EC [Vog09]. However, while this nicely serves several novel types of applications, it has turned out that it is very challenging to develop traditional applications on the top of weakly consistent databases [AEM+13, Ham10]. The complexity arises from the necessity of compensating the missing consistency guarantees at the application level. In reaction to that, today, NoSQL databases provide a range of consistency models stronger than EC (Figure 1.3). Applications can thus choose the most suited consistency model based on their demands such as, for instance, their performance requirements. In the Cloud with its *pay-as-you-go* cost model where each resource and each action 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 consistency costs, weak con-

[3]Denotes the ability of a system to adjust its behavior at runtime to different requirements and constraints.

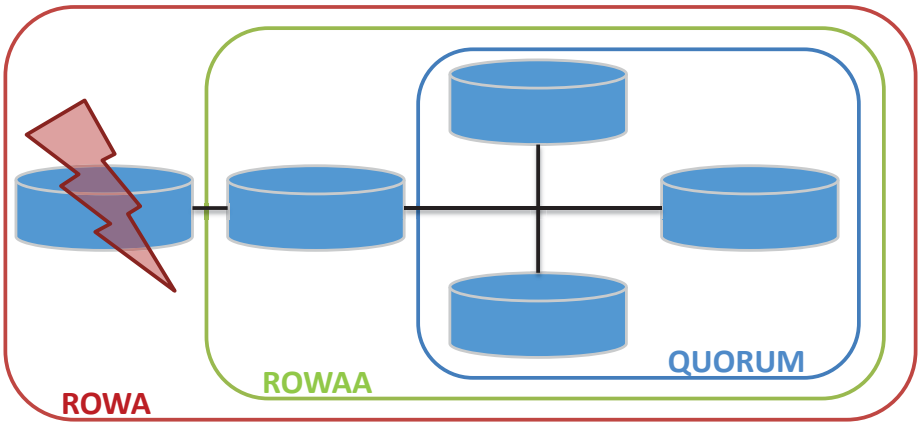


Figure 1.4: Eager replication protocols.

sistency in turn may generate high inconsistency costs for compensating access to stale data (e.g., book oversells) [KHAK09, FS12].

Current approaches allow only a static choice of consistency that is not able to cope with the dynamic nature of the application workload. Thus, the decision on the consistency model may not be optimal as it is based on knowledge that will potentially be outdated at execution time. Additionally, current approaches do not provide a means of influencing the data consistency based on the monetary costs [KHAK09, FK09].

1.2.2 Data Partitioning

Update transactions in the presence of replication, require additional coordination between sites if applications demand strong consistency guarantees [KJP10, CZJM10, TMS+14]. This is the case for various types of Online Transaction Processing (OLTP) applications, such as financial applications, that cannot sacrifice consistency and thus, need guarantees like 1SR which, in turn, requires expensive distributed commit protocols like *Two-Phase Commit (2PC)* or Paxos [GL06]. These protocols are expensive since they require some costly rounds of network communication. This additional overhead is considerable if geo-replication is used [BDF+13]. Moreover, 2PC may block in case of failures, and this may lead to a decreased availability level as resources are unavailable during the blocking phase. Non-blocking distributed commit protocols, such as Paxos [GL06, Ske81], can be used at even higher costs than 2PC. At the same time, despite the high latency due to the overhead of transaction coordination [BDF+13, Dea], these applications need highly scalable databases [Asl11].

Shared-nothing architectures [Sto86], that make use of data partitioning (also known as sharding), are able to manage data in such a way that distributed transactions are avoided – or at least that their number is minimized. This eliminates or minimizes the necessity of expensive distributed commit protocols.

Existing partitioning protocols are mainly static and thus tailored to a specific workload. This makes them unsuitable if the workload shifts at runtime, which is inherent for applications deployed in the Cloud. Dynamic approaches are necessary that are able

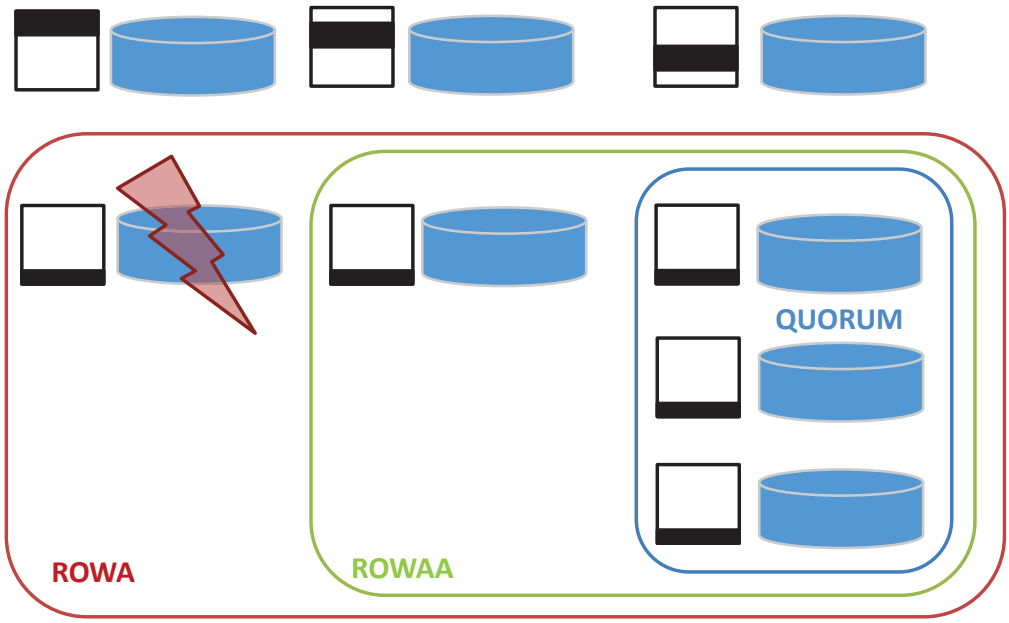


Figure 1.5: Partitioning with replication.

to adapt to workload changes with the goal of retaining the advantages of the data partitioning. However, the process of partitioning and reconfiguring the system is expensive and should be done only when it provides a benefit to the application, otherwise the reconfiguration costs may overweigh its benefit.

1.2.3 Data Replication

Data replication is a mechanism used to increase the availability of data by storing redundant copies (replicas) at usually geographically distributed sites. In the case of read-only transactions, data replication can increase system scalability by using the additional processing capacities of the hosts where replicas reside to balance the load. However, in the case of update transactions, depending on the desired data consistency guarantees, data replication may generate a considerable overhead and thus decrease the overall system scalability and availability [KJP10]. For many applications, 1SR is the desired level of data consistency. It guarantees serializable execution of concurrent transactions and a one-copy view on the data [BHG87]. Usual protocols implementing 1SR are based on Two-Phase Locking (2PL) for the synchronization of concurrent transactions and 2PC for the eager commit of replica sites in case of update transactions [BHG87]. However, as depicted in Figure 1.4, protocols differ in the strategies with regards to the commit strategy, which impacts both the provided availability and the overhead for transactions. While Read-One-Write-All (ROWA) protocols update all replica sites eagerly, Read-One-Write-All-Available (ROWAA) will consider only those sites that are available, leading to an increased level of availability compared to ROWA at the cost of increased complexity for site reconciliation [KJP10]. Quorum protocols consider only a subset of all sites when committing, leading to a decreased overhead

for update transactions. It is well known that quorum protocols are better suited for update-heavy workloads compared to the ROWA(A) approaches [JPAK03].

The size of the quorums, as well as the properties of the sites constituting these quorums, is one of the major factors for the overall transaction overhead. Existing quorum-based protocols generate quorums of different sizes, and usually neglect the site properties, or consider them without the ability to adapt the quorums if the properties change.

Replication or data partitioning alone solve only parts of the problem. While replication is expensive for update transactions, data partitioning does not provide any availability guarantees in case of site failures. Hence, partitioning needs to be combined with replication to find a trade-off between consistency, availability, and latency that satisfies application requirements (Figure 1.5). Existing approaches consider either replication or partitioning in an isolated manner without providing a holistic model that would combine both dimensions with the goal of satisfying availability and performance requirements.

1.2.4 Challenges Related to the Development of Cost and Workload-driven Data Management Protocols

In summary, existing data management approaches have the following limitations. First, they neglect the cost parameter, which is becoming increasingly important for applications deployed in the Cloud. Second, they are tailored to a subset of scenarios characterized by their workload and are unable to adapt their behavior at runtime in case the workload shifts. This rigidity may lead to a violation of application requirements and a considerable increase in costs.

Thus, *cost and workload-driven* protocols are necessary, which are able to cope with dynamic application workloads and that consider the costs as a first-class citizen. The challenges related to the development of such protocols are as follows:

1. It is necessary to develop models that capture relevant costs for different data management properties, such as data consistency, partitioning and replication. These models should steer the protocol behavior, so that the costs for providing the desired application guarantees are minimized.

2. It is necessary to develop or incorporate workload prediction models, which allow the protocols to adjust their behavior at runtime so that application requirements continue to be satisfied.

3. It is necessary to develop low cost and safe approaches, which are able to reconfigure the protocols if their behavior is to be adapted. The low cost requirement is crucial for ensuring that the reconfiguration costs do not outweigh its gain. Safety denotes the ability to reconfigure a protocol without violating its correctness.

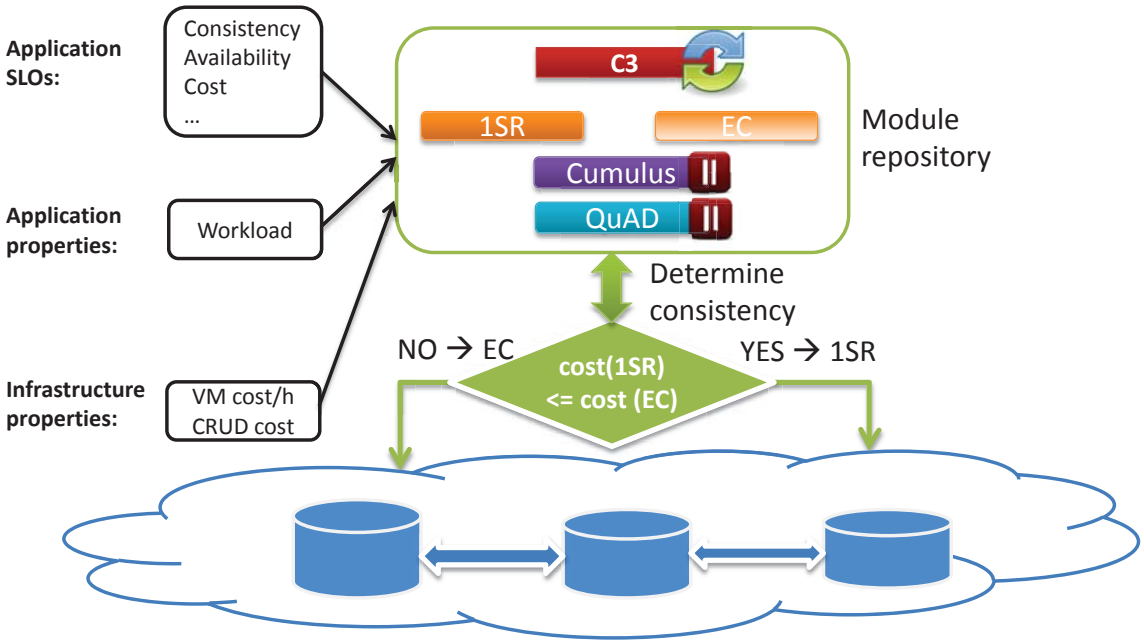


Figure 1.6: C^3: Cost and workload-driven consistency control protocol.

1.3 Thesis Contributions

In what follows, we will summarize our cost and workload-driven data consistency, partitioning and replication protocols, that tackle with the aforementioned challenges. In this thesis, our protocols will be jointly referred to as CCQ.

1.3.1 C^3: Cost and Workload-driven Data Consistency

C^3 is a cost and workload-driven data consistency protocol, which can dynamically adjust the consistency level at runtime so that application costs are minimized [FS12]. The consistency level of transactions is determined by finding the right balance between the operational costs – denoted as consistency costs – incurring for providing that consistency level, and the application specific inconsistency costs that incur for compensating the effects of relaxed consistency (Figure 1.2). As depicted in Figure 1.6, in the current version, C^3 considers the 1SR and EC consistency models. However, considering its modular architecture, C^3 can be easily extended with further consistency models and protocols, allowing it to extend its optimization space (Figure 1.3).

The contributions in context of C^3 are as follows:

We introduce C^3, a meta-consistency protocol, which determines the optimal consistency level of transactions based on a cost model that finds the right balance between consistency and inconsistency costs. Consistency costs denote the costs that incur from the consumption of resources and services necessary to enforce a certain consistency level, whereas inconsistency costs denote the costs for compensating the effects of a relaxed consistency level. Moreover, the consistency decision

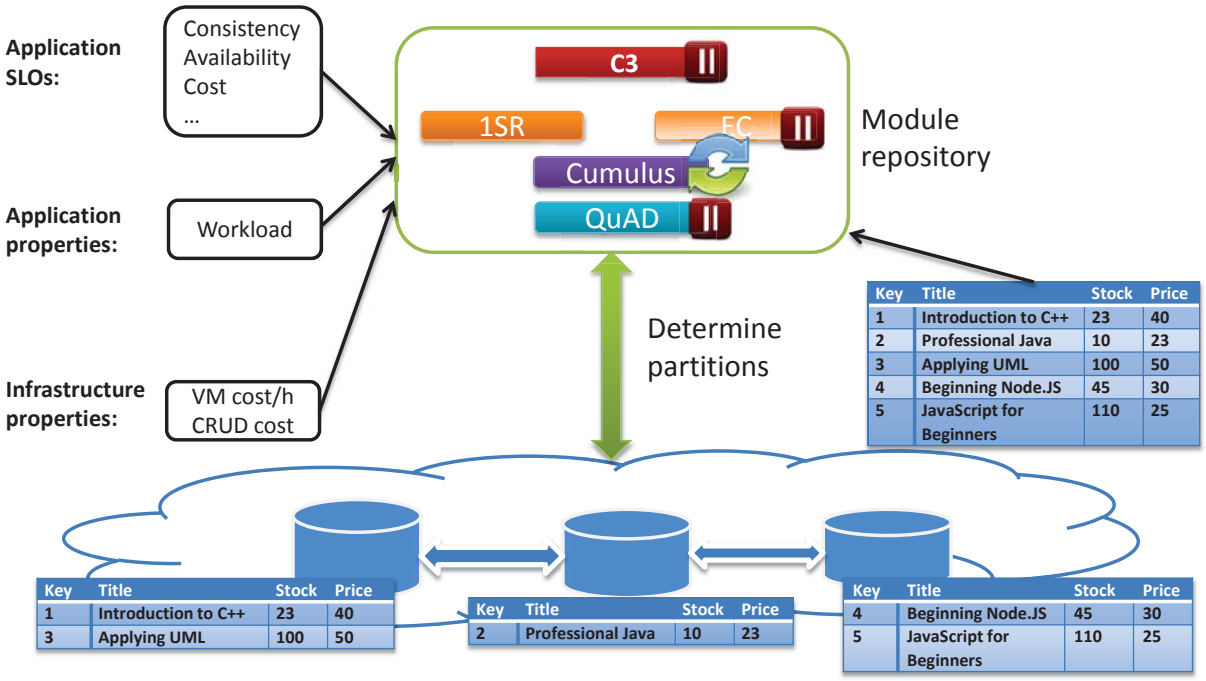


Figure 1.7: Cumulus: Cost and workload-driven data partitioning protocol.

will be continuously assessed, and C^3 will initiate a reconfiguration, i.e., will adjust the consistency level, if the gain in doing so outweighs the reconfiguration costs.

C^3 implements a stop-and-copy reconfiguration approach, which guarantees that all sites have the same view on the active consistency level even in presence of failures.

C^3 provides an intuitive Application Programming Interface (API) that allows developers to enforce a certain consistency level or to define application specific inconsistency costs and let C^3 to adaptively adjust the consistency.

C^3 is able to handle multi-class transaction workloads by adjusting the most optimal consistency level for each of the classes separately.

We describe a complete implementation of C^3 and present thorough evaluations conducted in the EC2 infrastructure. The evaluations compare C^3 with static approaches in terms of costs and performance. The results show the feasibility of C^3 and its capability to reduce application costs. Moreover, the results show that C^3 is able to adjust transaction consistency at runtime and handle multi-class transactions.

1.3.2 Cumulus: Cost and Workload-driven Data Partitioning Protocol

The second contribution is related to the development of a cost and workload-driven partitioning protocol called Cumulus [FMS15]. As depicted in Figure 1.7, the main goal of Cumulus is, given a certain application workload, to determine a partition schema that reduces or completely avoids distributed transactions. Cumulus will monitor the application workload and readjust the schema so that the advantages of data partitioning are retained.

The contributions in context of Cumulus are as follows:

We introduce Cumulus, a cost and workload-driven protocol that is able to partition the data based on a model that captures the costs of distributed transactions. Cumulus is able to dynamically adjust the partitions based on the application workload and the cost model. Cumulus finds the right balance between the reconfiguration costs and the gain resulting from a reduced number of distributed transactions with the goal of avoiding unnecessary and expensive reconfigurations.

Cumulus implements a reconfiguration approach, which ensures that all sites have the same view on the active partitions even in presence of failures.

Cumulus is able to distinguish between significant (frequent) and insignificant (infrequent) transactions in the workload. It will consider only significant transactions for the definition of partitions, as considering all transactions: i.) may not be feasible in an adaptive protocol, and ii.) partitions tailored to insignificant transactions will generate low or no benefit at all.

We describe the implementation of Cumulus and present its evaluation results. The evaluations compare Cumulus, in terms of partitions quality, to static approaches, and approaches that do not incorporate a means of analyzing the workload. The results show that Cumulus considerably outperforms the other approaches, and we depict its ability to adjust the partitions at runtime in reaction to workload shifts.

1.3.3 QuAD: Cost and Workload-driven Quorum Replication Protocol

The third contribution, called QuAD, considers the development of a cost and workload-driven quorum replication protocol that considers the properties of the sites, such as their load and network proximity, when constructing the quorums [SFS15] (Figure 1.8). QuAD is based on the idea of avoiding 'weak' sites from the read and commit paths of transactions, where weak can have different meanings, such as slow and distant, but also expensive. QuAD jointly considers the load of the sites and their distance, i.e., the Round-Trip Time (RTT), when determining the quorums. Additionally, it seeks a possibly balanced assignment of sites to quorums, since if some sites are frequently included in the quorums, they may become a bottleneck.

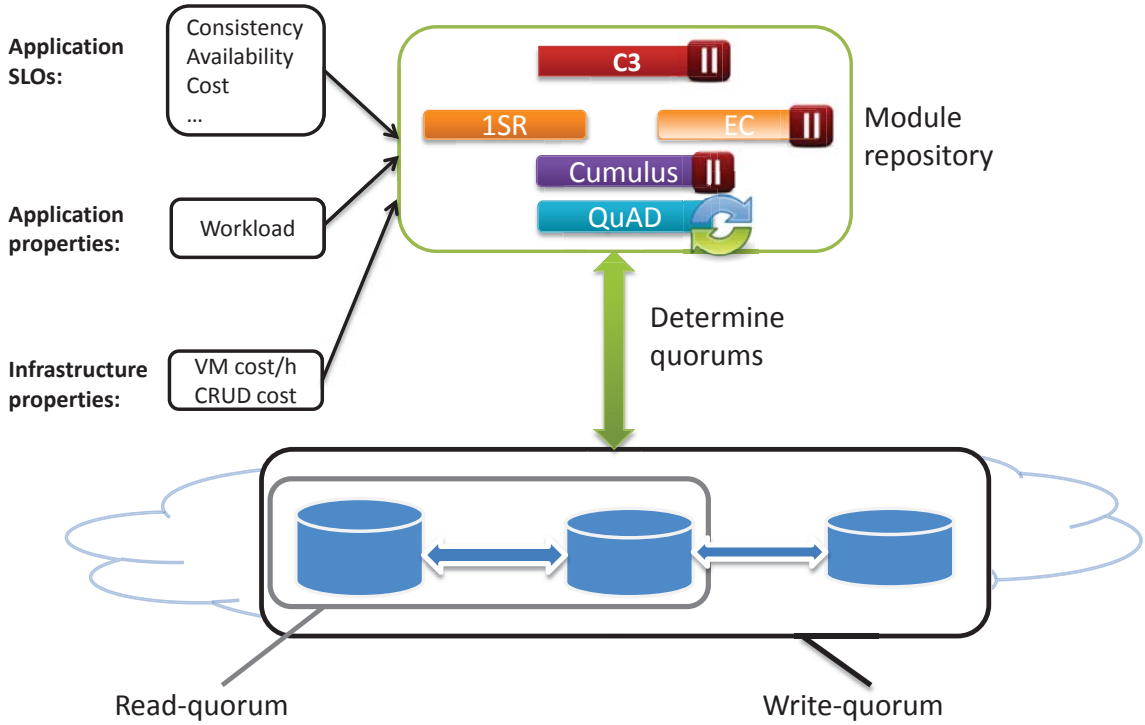


Figure 1.8: QuAD: Cost and workload-driven quorum-based replication protocol.

QuAD is able to react to changes in the system, such as increased load at sites, new sites joining the system, or site failures, and consequently to adapt its quorums to address these changes.

The contributions in context of QuAD are as follows:

We provide a cost model for the quorum construction that jointly considers the load and network distance between sites. The goal of the cost model is to construct the quorums in such a way so that weak sites are avoided from the read and commit paths of transactions, if possible. QuAD is able to dynamically adapt its quorums in reaction to workload changes and site failures. A reconfiguration is initiated only if the gain outweighs the reconfiguration costs.

The QuAD reconfiguration approach ensures a consistent view on the quorum configurations even in presence of failures.

We describe the implementation of QuAD and present thorough evaluations conducted in the EC2 infrastructure. We compare QuAD to other quorum protocols that i.) neglect site properties, ii.) consider only a subset of site properties, and that iii.) are non-adaptive. The evaluations show that QuAD considerably outperforms, in terms of transaction performance, both static quorum protocols, and dynamic protocols that neglect site properties during the quorum construction process. Moreover, they show that QuAD is able to reconfigure the quorums in case of workload shifts or site failures.

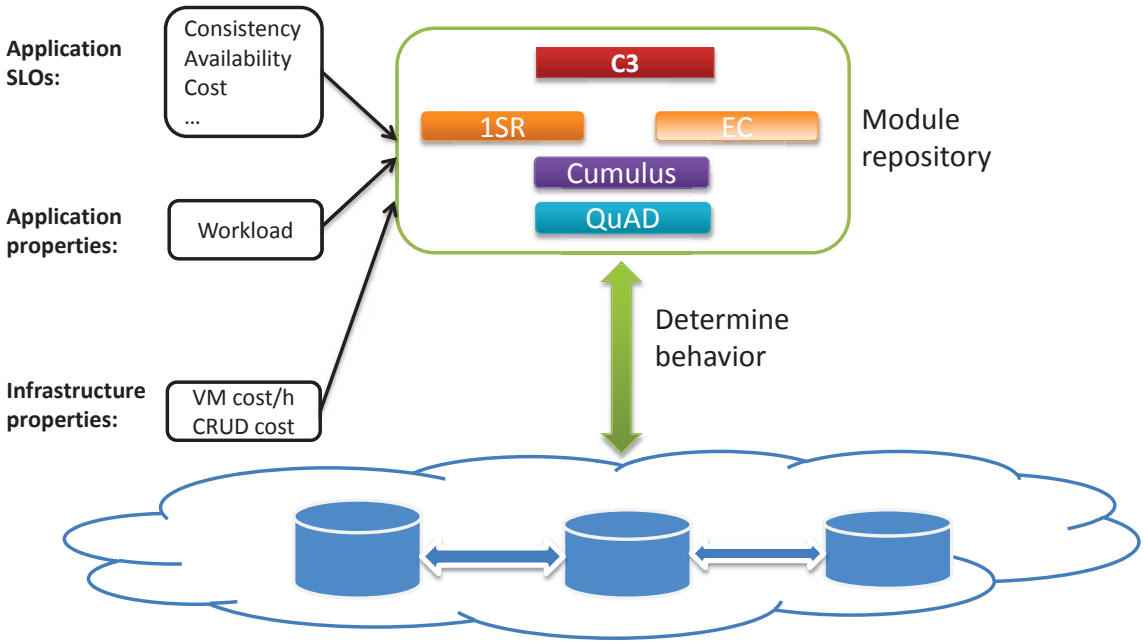


Figure 1.9: PolarDBMS: Policy-based and modular DBMS.

1.3.4 PolarDBMS: Policy-based and Modular DBMS

In [FBS14] we introduced our work in progress system PolarDBMS (*Po*licy-based and modu*lar* DBMS for the Cloud). PolarDBMS is based on a modular architecture, with each module (protocol) providing certain data management functionality, such as data replication (ROWAA, quorum), data consistency, atomic commitment, consensus protocols, etc. The main objective of PolarDBMS is to automatically select and dynamically adapt the module combination that best provides the desired guarantees by incorporating the incurring costs, the application objectives and workload, and the capabilities of the underlying system (Figure 1.9).

In context of PolarDBMS, this thesis provides concrete contributions towards its realization. Our CCQ protocols are implemented as modules, which can be seamlessly integrated into PolarDBMS. While each of the modules tackles one aspect of data management, PolarDBMS serves as an integrator, which, based on a holistic model, steers the modules and their behavior. The holistic model considers the requirements of different applications, and the requirements of the Cloud provider, which aims to provide the best possible data management behavior.

1.4 Thesis Outline

This thesis is organized into nine chapters. The first two chapters, i.e., this chapter and Chapter 2, provide an overview on Cloud Computing as well as on challenges and opportunities related to the data management in the Cloud. We have summarized the research field of this thesis and the problem statement as well as the main contributions.

The problem statement is further detailed in Chapter 2 based on a concrete application scenario. The embeddings of this thesis' contributions in the context of a long-term project that aims at building a fully-fledged Service Level Agreement (SLA)-driven and modular DBS are also described in Chapter 2.

In Chapter 3 we describe the fundamentals of the data and transaction model for DBSs, we summarize main concepts related to data consistency, replication and partitioning, and provide a formal model for capturing data freshness that complements existing consistency models. Chapter 4 introduces the big picture and the main concepts, and definitions related to the development of cost and workload-driven data management protocols in general, and to the development of the CCQ protocols in particular.

Chapter 5 is devoted to the concepts developed as part of the contributions. We provide an in-depth description of the CCQ protocols, and their cost and configuration models.

In Chapter 6 we describe the implementation details of the CCQ protocols.

Chapter 7 discusses qualitative evaluations of our contributions, and quantitative comparisons to other approaches on the basis of the Transaction Processing Performance Council (Transaction Processing Performance Council (TPCC)) benchmark.

Finally, Chapter 8 summarizes related work, and Chapter 9 provides an outlook on possible future work and concludes.

2

Background and Motivating Scenario

IN THIS CHAPTER, we describe an online shop application scenario that illustrates the challenges of data management in the Cloud. Based on the scenario, we derive a generic interaction cycle between customers, i.e., application providers and Cloud providers, with the goal of satisfying the application requirements as good as possible by considering general constraints (e.g., legal constraints), multi-tenancy (requirements of different applications) as well as provider's capabilities and its requirements.

When considering all layers of the online shop application depicted in Figure 2.1, there are different actors, such as end users, application providers, service providers, cloud providers, and others, that may provide or consume services. These roles are dependent on the context, and may change if the context changes. For example, the application provider provides services towards the end users, and consumes services of the underlying service provider. A *client* denotes a consumer of a service provided by a *server*. A service may denote an application, a domain specific service, such as a payment service, Cloud infrastructure, data management service, etc. Each client specifies its requirements towards the server in form of SLAs. An SLA contains financial aspects of service delivery, penalties, bonuses, terms and conditions, and performance metrics denoted as Service Level Objectives (SLOs) [KL02]. The underlying service will generate guarantees, denoted as Service Level Guarantees (SLGs), that will best match the specified SLAs by considering all possible constraints.

Each layer of an application must not only consider the specified requirements in form of SLAs, but must continuously monitor them, the generated SLGs and adjust its configuration if the SLGs significantly deviate from the requirements, either due to a change in SLAs or shift in end user behavior (e.g., increased load). Usually, this has a cascading effect on all layers, which may lead to their reconfiguration.

In this thesis, we focus on the data layer, more concretely on the development of the adaptive data management protocols under the umbrella of PolarDBMS. PolarDBMS is based on a modular architecture, and considers SLAs to determine its module composition and the module configuration that best satisfy the given SLAs. It will continuously monitor the fulfillment of the SLAs, and dynamically adapt the configuration of existing modules, or exchange modules if necessary in order to reflect changes in the SLAs or client behavior. In what follows we will motivate the need for PolarDBMS based on a

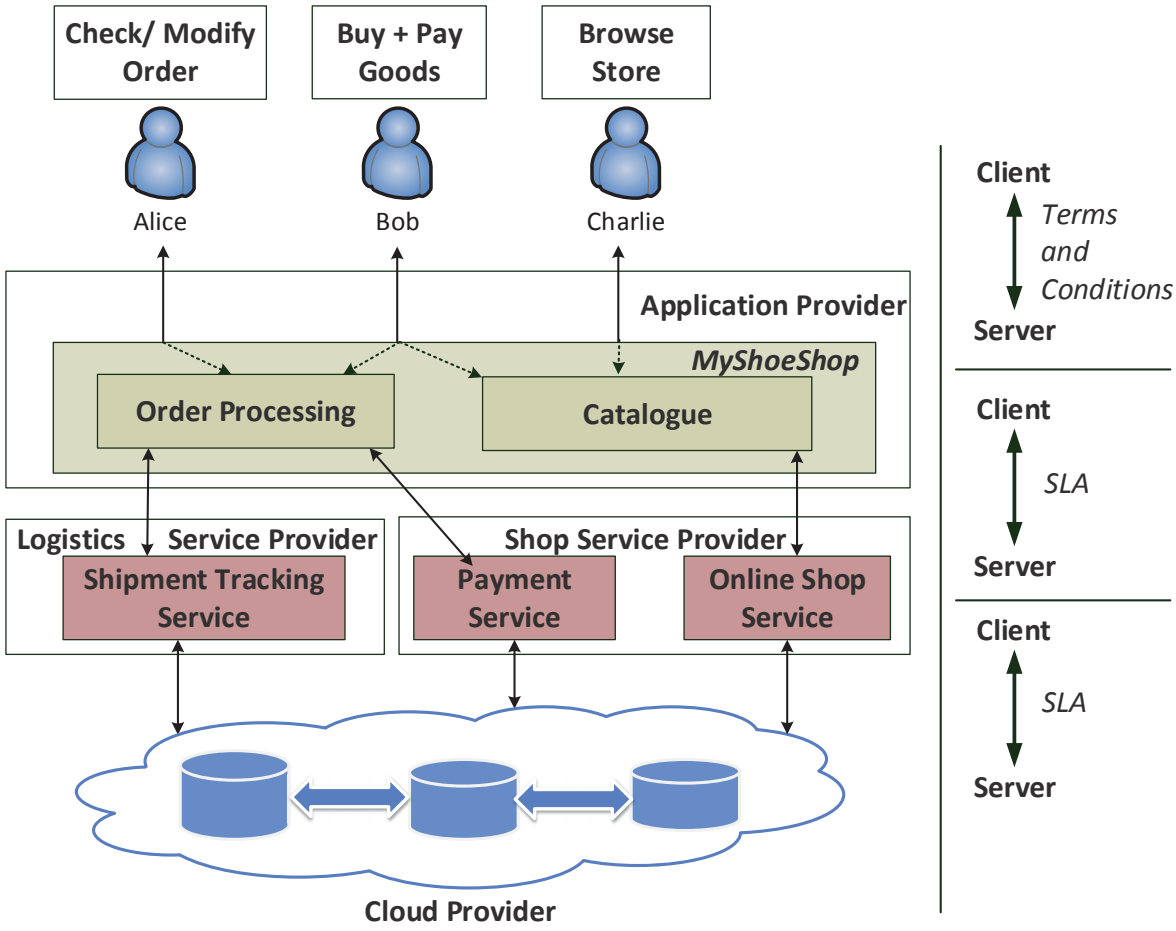


Figure 2.1: Online shop application scenario [FBS14].

concrete scenario. While PolarDBMS considers the full spectrum of data management, the CCQ protocols consider only a subset of data management properties, namely data consistency, partitioning and replication.

2.1 Online Shop

This application scenario describes a simplified online shop, which is depicted in Figure 2.1. 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 `MyShoeShop` application, several subsystems are responsible for the different actions. While the `Catalogue` 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

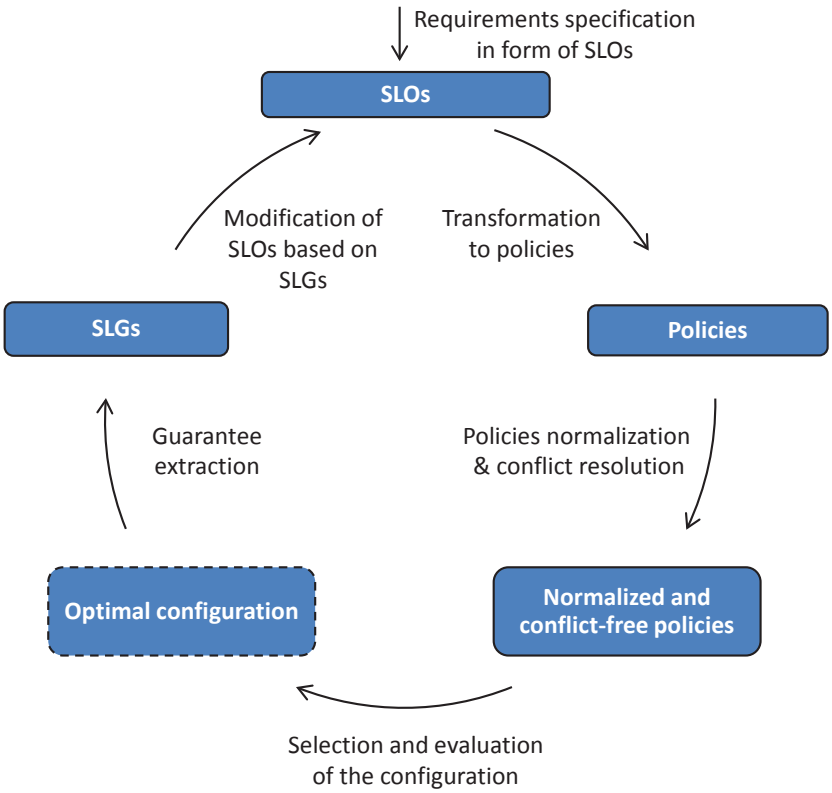


Figure 2.2: Client-server interaction cycle in the Cloud [FBS14].

provider hosts a `Payment Service`. In this context, the term service is used to denote a reusable functionality for a set of business domains, that is typically not directly accessed by end customers. In contrast to a service, an application provides functionality of benefit for the end customer, and uses one or more services to achieve its goals (cf. programming in the large [DK75]).

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 DBSs (e.g., NoSQL and NewSQL systems [Mel00]). 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 to specify the details of the bilateral relationship. This applies also for the interaction between the service providers and the underlying DBS provider. Furthermore, the aforementioned ToC 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 with regards to the guarantees provided towards the end user.

As today's businesses are rapidly evolving, a dynamic adaptation to changing requirements is necessary. The `MyShoeShop` provider in the scenario may, for instance,

SLO	Description
availability	Defines the desired degree of availability. The *availability* SLO is mapped to a number of parameters, such as the number of replicas (*#replicas*) data is replicated to and the the distance between them.
maxBudget	Defines the maximum budget that can be spent for providing the desired guarantees.
upperBoundAccessLatency	Defines the desired upper bound of access latency.

Table 2.1: Summary of SLOs.

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, in particular DBSs 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. For example, the `MyShoeShop` provider may choose a relational DBS or a NoSQL database for its application. In contrast to NoSQL databases, which provide relaxed consistency guarantees, relational DBSs provide strong consistency at the cost of scalability and availability. As today's applications are basically available to everyone, and as customer behavior and requirements towards the application may shift at any time, the choice of the DBS may turn out to be suboptimal. Currently, it is difficult, if not impossible, to adapt the DBS even by highly skilled administrators to satisfy the application requirements, let alone the DBS to adapt its behavior at runtime without any manual intervention. Faced with such a situation, application providers usually either choose to remain in the status quo, which is a clear business disabler, or initiate expensive migration processes to another DBS, which may not be the most appropriate choice anymore by the time migration has finished. This might be a consequence of in meanwhile changed legal or business constraints, or customer behavior.

This urgently demands a dynamically adaptable system, which allows for a flexible interaction between customers and providers in broader terms, and between applications and DBSs more specifically.

2.2 Optimal Interaction Cycle between Clients and Servers

An optimal interaction cycle between clients and cloud providers in a business interaction is depicted in Figure 2.2. A client using a service provided by a server specifies her requirements towards that service in form of SLOs that are included in the SLA (see Table 2.1). An SLO specifies an expected guarantee with regards to a parameter [KL02].

For example, a client might define a lower bound of availability that has to be guaranteed by a service:

$$SLO : availability \geq 0.95$$

In the example above *availability* denotes the parameter, and 0.95 its value, i.e., the minimum level of availability to be guaranteed. The system providing the service transforms all SLOs of an SLA 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 of different clients into account. At the end of this process, the best possible configuration is chosen and transformed into SLGs. An SLG is a commitment of the provider on the fulfillment of an SLO:

$$SLG : availability \geq 0.99.$$

It might however be that the provided SLG does not fulfill the corresponding availability SLO. A client may require an $availability \geq 0.95$, whereas the provider may only be able to guarantee $availability \geq 0.9$. In that case, the client has the possibility to adapt its SLO or decide to use another service provider.

An adaptation of the SLOs leads to the restart of the entire interaction 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.

2.3 Integrated Data Management

The CCQ protocols developed as part of this thesis consider only a subset of a data management properties and SLOs. They rely on the ability of PolarDBMS to resolve conflicting requirements of different clients and to validate the satisfiability of the resulting policies. Moreover, based on these policies, PolarDBMS will choose the most appropriate module composition and configuration so that the requirements are optimally satisfied. PolarDBMS enables the implementation of the optimal interaction cycle depicted in Figure 2.2 at the data management layer, which gives the application the full flexibility to react to changed legal or business constraints, and client behavior. In what follows we will describe the high-level architecture of PolarDBMS, its components, as well as their interactions.

As depicted in Figure 2.3, clients specify their SLOs and submit to PolarDBMS. As part of phase a)., the submitted SLOs are integrated with the existing system capabilities as described in the Example 2.1. During this phase, the *integrator* decides if the specified SLOs can be satisfied at all given the system capabilities.

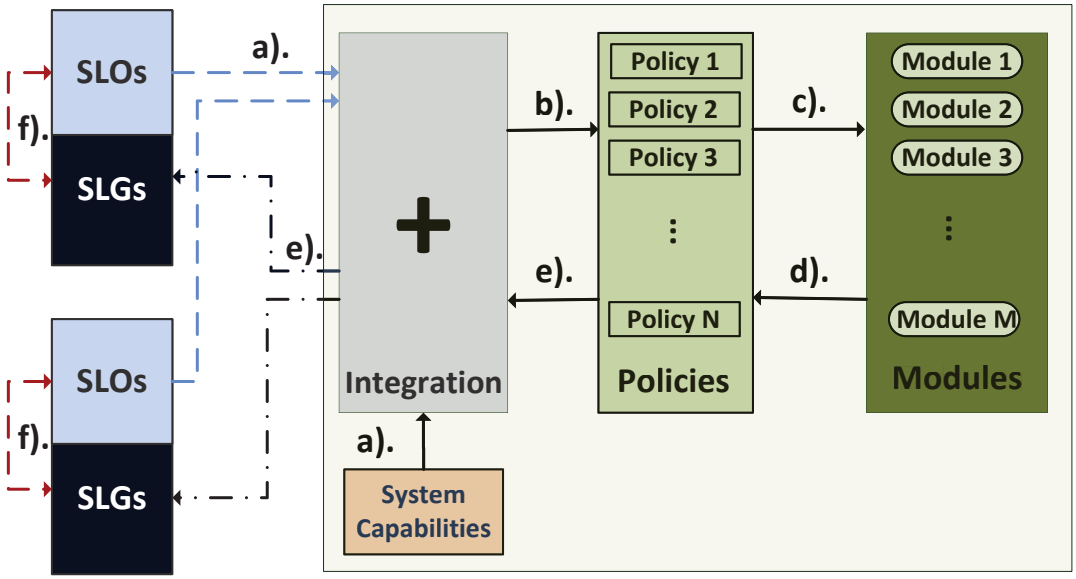


Figure 2.3: Interaction flow in PolarDBMS [FBS14].

Example 2.1 (System Capabilities)

Let us assume that a client has defined the following SLO: $availability \geq 0.99995$. Additionally, the client may require the data to be stored only in the EU. However, the provider runs data centers only within the US. This is a sort of a business constraint, which, along with possible legal, organizational and functional constraints, defines the system capabilities. In this case, the client must be notified that the requirements on the data storage location cannot be satisfied.

In the same scenario, the client may also specify that it desires 1SR consistency for its application. However, the DBS provider is only capable of providing EC consistency. This is a functional constraint that hinders the satisfiability of the client's requirements.

In both cases it is clear that the provider, given its capabilities, cannot satisfy the requirements, and must reject them. In reaction to that, the client may adapt its SLOs or change the provider, which would restart the interaction cycle (Figure 2.2).

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.) as described in the Example 2.2.

Example 2.2 (Policies)

The availability SLO is mapped to a policy, which consists of various parameters, such as for example the number of machines (#*replicas*) data has to be replicated to (phase b)). High availability, thus high #*replicas* leads to higher costs. The relationship between #*replicas* and costs is reflected in another policy. This needs to be taken into account in case the client wants to limit the costs 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.

Certain parameters (e.g., *#replicas* needed to satisfy the desired level of availability) can be estimated without involving the capabilities of the concrete functional building blocks of PolarDBMS, which are represented by modules. However, for the final decision on the fulfillment of other SLOs (e.g., $uperBoudAccessLatency$) a collaboration with the underlying modules might be necessary (Example 2.3). 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, which aims to find the optimal combination of modules together with their configurations (see dashed box in Figure 2.2).

Example 2.3 (Negotiation Process)

The guarantees provided by certain modules, such as the $upperBoundAccessLatency$, may be dependent on the application workload. For instance, while ROWAA is suitable for read-only or read-heavy workloads, quorum-based Replica Protocols (RPs) are more appropriate for workloads that contain a large portion of update transactions.

Any of the RP modules require the application workload as input to determine the level of guarantees they can deliver, which is typically not known a priori and must be predicted. Different workload prediction modules may exist that differ in terms of overhead and prediction accuracy. Again, it is the task of PolarDBMS to select the most appropriate prediction module by considering the different available constraints.

The ability to predict the application workload is however based on the assumption that there are some runtime data available about the application behavior. As the application workload is highly dynamic, the satisfiability of the client requirements needs to be continuously monitored. The negotiation process may be restarted any time when there is a significant change in the workload.

As described in the Example 2.3 there might be multiple modules that implement the same data management property. Moreover, as data management properties are not orthogonal, they might also be influenced by policies related to other data management properties as the consequence of the CAP trade-off (Example 2.4).

Example 2.4 (Integrated and Adaptive Data Management)

Let us assume that the client has specified a lower bound of availability guarantee that must not be violated ($availability \geq 0.99995$), as well as an upper bound of latency ($upperBoundAccessLatency \leq 50ms$). By considering the consistency-latency and consistency-availability trade-offs, PolarDBMS will choose the most appropriate consistency model and module implementing the model so that the latency requirement ($upperBoundAccessLatency$) is satisfied.

In the aforementioned case, the decision on the consistency model is left to PolarDBMS. However, in addition to the availability and latency requirements, the client may specify that DBS must enforce a certain consistency model, such as 1SR. In this case, PolarDBMS may consider the application workload to determine the most appropriate 1SR implementation (module) (see Example 2.3).

The choice of the consistency model may also be influenced via the costs. For example, instead of enforcing a certain consistency model, application may quantify the overhead for compensating the effects of relaxed consistency models in terms of costs. PolarDBMS should then jointly consider all requirements, the application workload and infrastructure properties (e.g., #*replicas* and their network distance), and decide on the most appropriate consistency model and implementation so that the requirements are optimally satisfied.

The dependencies between the different data management properties, and modules implementing these properties, imply the necessity of finding the optimal module set, which delivers the best *module guarantees* with regards to the integrated objectives.

The guarantees are then transformed back to policies as part of phase d.). At this point, the negotiation process is finished and the policies are transformed into client SLGs (phase e.)). The contrast between the originally defined SLOs and the offered SLGs (phase f.)) may lead to the client choosing another DBS provider or to the adaptation of the SLOs, if e.g., $expBudget > maxBudget$. In the second case, the client may modify its desired availability requirement. This leads to the adjustment of the entire interaction cycle.

2.4 Modular Database Systems

For requirements regarding data management in the Cloud, the underlying DBS 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 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, application workload 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 DBSs [GD01].

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 [MD88, W+74]. The use of a high level language allows clients to easily specify the requirements of their applications and shields details of the underlying DBS. 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 CCQ protocols are implemented as PolarDBMS-modules and consider a subset of possible data management properties and client SLOs. The choice of which CCQ modules and configurations are deployed is steered by the interaction cycle depicted in Figure 2.2. The generated SLGs are continuously monitored, and if they do not satisfy the SLOs, either the module configuration is adapted or the active module is replaced by another (more suitable) CCQ module. Moreover, the CCQ modules can be integrated (combined) to jointly consider a subset of SLOs that affect different data management properties (Example 2.4).

3

Foundations of Transaction and Data Management

IN THIS CHAPTER, we will provide an overview of the transaction and data model for single-copy and distributed database systems that lay the foundation for the subsequent chapters. We will first summarize the main correctness models for concurrent transactions in single-copy database systems, and describe protocols that implement these models. Distributed database systems host data in different locations (sites). This is in sharp contrast to single-copy databases, which consist of a single site. While data distribution provides considerable advantages in terms of performance and availability, from the formal point of view, it necessitates the extension of the correctness models to consider the location of data. We will provide an in-depth discussion on these extensions, and describe protocols that implement the models in context of distributed database systems. Fully replicated databases define a subtype of distributed database systems, in which data object are physically present at multiple sites, i.e., have multiple copies (replicas). In this context, we will define the *freshness* concept, which is eminent for capturing correctness in distributed databases with replication, and describe protocols that, by controlling the copies, define the freshness of data observed by transactions.

This chapter is concluded with a summary of the fundamental CAP/PACELC tradeoffs in distributed database systems, which have been the main driving force for the development of different correctness models and protocols implementing these models.

3.1 Single Copy Database Systems

A *database* is a collection of objects denoted as denoted as $LO = \{o_1, o_2, \cdots\}$ (see Table 3.1). Each object o consists of an id o_{id} and a value denoted as val. It is up to an application to organize the objects into types and decompose their values to a set of attributes. This aspect is however not relevant in the context of this thesis. The values of all data objects from LO at any time point denote the database *state* [BHG87]. A DBS denotes the hardware and software that support access to the database through a set of operations denoted as OP [BHG87]. In this thesis we will consider the *read/write* model,

Symbol	Description
LO	Denotes the set of all data objects.
o	Denotes a logical data object: $o \in LO$.
o_{id}	Denotes the *id* of o.
o^v	Denotes the v version of o.
val	Denotes the value of an object.
T	Denotes the set of all transactions.
t	Denotes a transaction: $t \in T$.
t^r	Denotes a read-only transaction.
t^u	Denotes an update transaction.
OP	Denotes the set of all operations.
op	Denotes an operation: $op \in OP$.
r	Denotes a read operation: $r \in OP$.
w	Denotes a write operation: $r \in OP$.
AC	Denotes the set of all actions.
$AC.t$	Denotes the set of actions of transaction t.
ac	Denotes an action: $ac \in AC$.
$ac.op$	Denotes the operation of action ac.
$ac.o$	Denotes the object o on which $ac.op$ operates.
$Term$	Denotes the set of termination actions.
$term$	Denotes a termination action: $term \in Term$.
$t.term$	Denotes the termination action of transaction t.
c	Denotes a commit (termination) action: $c \in Term$.
a	Denotes an abort (termination) action: $a \in Term$.
ac_y	Denotes an action of transaction t_y.
ac_y^i	Denotes the i^{th} action of transaction t_y.
$RS(t)$	Denotes the read-set of transaction t.
$WS(t)$	Denotes the write-set of transaction t.
SCH	Denotes the set of all schedules.
sch	Denotes a schedule: $sch \in SCH$.
$sch.T$	Denotes the set of transactions T of schedule sch.
$RF(sch)$	Denotes the set of all *reads-from* between transactions of schedule sch.
$start(t)$	Denotes the start time of transaction t.
$commit(t)$	Denotes the commit time of transaction t.
$ts(t)$	Denotes the timestamp of transaction t.

Table 3.1: System model: symbols and notations.

in which OP consists of the *read* and *write* operations: $OP = \{r, w\}$. An action ac denotes an operation that acts on a specific logical: $ac \in OP \times LO$. A $r(o_i)[val_i]$ returns the value val_i of o_i without any side-effect, whereas a $w(o_i)[val_i]$ sets the value of o_i to val_i.

In [Gra81] the *transaction* concept was introduced, in which a transaction is defined as a collection of actions that form a single logical unit. The transaction concept defines a contract in terms of guarantees that are to be provided to applications by the DBS. In particular, these guarantees include the following aspects [WV02]:

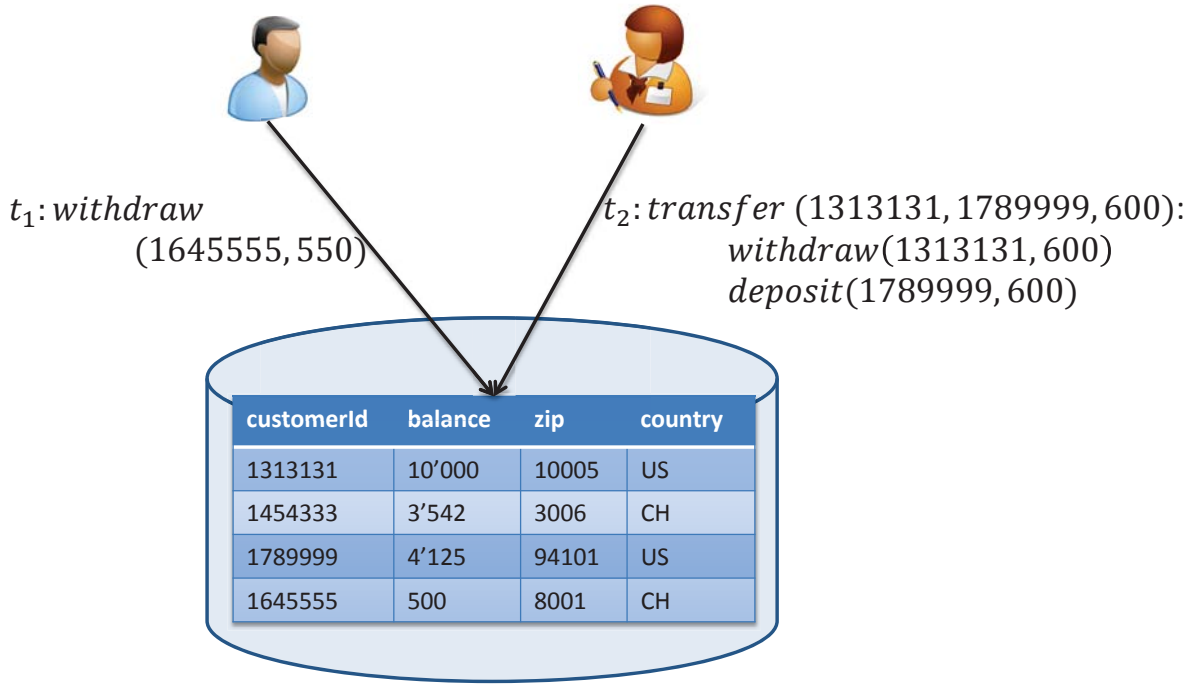


Figure 3.1: Transactions in a bank scenario.

1. Correct execution of *concurrent* or parallel data access.

2. Correct execution of data access in case of *failures*.

A transaction is defined as follows:

Definition 3.1 (Transaction). *A transaction t is a tuple: $t = (AC, \prec)$, with AC defining the set of actions and $\prec$ denoting the precedence relation of actions:*

$$AC = \{ac^1, ac^2, \cdots ac^k\} \cup Term$$
$$\prec \subseteq (AC \times AC)$$

A $term \in \{c, a\}$ denotes a termination action. Any termination action must be ordered after all other actions of a transaction with regards to $\prec$, i.e., $ac^i \prec term$ for all $1 \leq i \leq k$. The set of objects that are read by a transaction t denote its *read set* ($RS(t)$), whereas the objects that a transaction writes denote its *write set* ($WS(t)$).

A *read-only* transaction (t^r), in contrast to an *update* (t^u) transaction, does not include any write action, i.e., $WS(t^r) = \emptyset$.

3.1.1 ACID Properties

Transactions provide certain guarantees commonly referred to as ACID properties: (1) atomicity, (2) consistency, (3) isolation, and (4) durability.

Atomicity. Refers to the guarantee that a transaction is treated as unit of work. This means that either all operations of the transaction have succeeded or the transaction does not have any effect on the data. If the `withdraw` action of the `transfer` transaction in Figure 3.1 succeeds, but the `deposit` fails, then the old value on the withdrawn account (1313131) should be recovered.

Consistency. Is related to the guarantee that transactions should maintain consistency of database, which is defined by application-specific constraints. For example, the `withdraw` transaction in Figure 3.1 should not violate the constraint that an account can not have a negative balance.

Isolation. Defines that the effects of interleaved transaction execution should correspond to that of a serial execution. In the example with the bank in Figure 3.1 this means that, although different clients may simultaneously initiate transactions t_y and t_z that have non-disjoint read/write ($RS(t_y) \cap WS(t_z) \neq \emptyset$) or write/write sets ($WS(t_y) \cap WS(t_z) \neq \emptyset$), i.e., are in conflict, their effects on the database should correspond to a serial execution of the transactions. The coordination of concurrent conflicting transactions is denoted as Concurrency Control (CC) and implemented by a Concurrency Control Protocol (CCP).

Durability. Requires that once a transaction successfully commits, its changes to the data survive failures.

3.1.2 Data (In-) Consistency

In [FJB09] the authors define data and transaction consistency as follows:

Definition 3.2. *Data or database is consistent if its state satisfies given integrity constraints, and a transaction is consistent if its actions on a consistent database result in consistent states.*

From the Definition 3.2 one can conclude that given a consistent state of the database, if the workload consists of read-only transactions then it is impossible to generate data inconsistencies. Relaxing or avoiding ACID guarantees in case of workloads that contain update transactions, may result in different *inconsistencies*, which are summarized as follows.

Dirty read. In this case a transaction t_1 reads a data object which has been modified by another transaction t_2. If t_2 aborts, then t_1 has read an object or a value that never existed.

Non-repeatable read. A transaction t_1 reads a data object o_i. Later, another transaction t_2 updates the value of o_i and commits. t_1 re-reads o_i and gets a different value.

Phantom read. In such a case, a transaction t_1 reads a set of data objects satisfying a certain condition. Another transaction t_2 creates (inserts) new objects that satisfy the condition or deletes some of the existing objects. A second read from t_1 provides a different set of objects.

Lost-update. It occurs when a transaction t_1 reads an object o_i. Another transaction t_2 updates o_i and commits. t_1 updates the value of o_i based on the earlier value (it has received on the read) and commits. Then the update of t_2 has gone.

Constraint violation. Let us assume two objects $o_i = 5$ and $o_j = 10$ and the constraint: $o_i + o_j \geq 10$. Transaction t_1 reads $r_1(o_i)[5]$ and $r_1(o_j)[10]$. Later, another transaction t_2 reads both objects ($r_2(o_i)[5]$, $r_2(o_j)[10]$) and sets the value of o_i to 0 ($w_2(o_i)[0]$). t_2 can successfully commit as, from its viewpoint, it does not violate the constraint. Next, t_1 updates the value of o_j to 5 and successfully commits, as it is not aware of the t_2's update. The end result does however violate the constraint as $o_i + o_j = 5 < 10$.

3.1.3 Concurrency Control

Concurrency control is the activity of coordinating the interleaved execution of transactions [BHG87]. The main goal of CC is the concurrent execution of transactions (increased performance) with a consistency correctness corresponding to a serial (non-concurrent) execution. In a serial system, transactions would be executed one after the other, which would lead to total transaction ordering. In that case, no inconsistencies would occur. However, such a system would be highly inefficient as it would not allow any concurrency or parallelization of transaction execution.

A Concurrency Control Model (CCM) defines a correctness model for the execution of concurrent transitions. It defines allowed/disallowed interleaving of transitions and thus possible/impossible inconsistencies (see Section 3.1.2) [BBG+95]. A specific CCM, such as Serializability or SI, is implemented by a CCP. The same CCM can be implemented by different CCPs. For example, lock-based or timestamp-based protocols may be used to provide serializable execution of transactions [WV02]. It is crucial however that protocols comply with the correctness model, i.e., they should only produce transaction interleavings that are permitted by the model. Although it is not necessary that a protocol produces every allowed interleaving, it is required that each interleaving of concurrent transactions must be correct with regards to the correctness model.

The stronger the CCM the more inconsistencies are forbidden (the stronger the consistency) and vice-versa [BHG87]. Based on the CAP/PACELC trade-offs and on the observation that not every application demands strong consistency [Fek05], existing DBSs usually provide a range of models and protocols, leaving the choice of suitable model to the application developers. The choice may consist of a single correctness model or a mix of models and protocols [BBG+95, Ady99, Fek05, ALO00].

A correctness model is defined in terms of generated *schedules*. A schedule is a materialization of the correctness criteria, i.e., an interleaved execution of transactions conforming to that correctness criteria [WV02, BBG+95]. Schedules can be *complete*, i.e., contain all actions of each transaction; or *incomplete*. The first type of schedules is also known as *histories*. Incomplete schedules are prefixes of histories and from now on will be simply called schedules.

Definition 3.3 (Schedule)**.** *A schedule is a triple* $sch = (T, \mathbb{AC}, \prec_{AC})$*, with* $\mathbb{AC} = AC.t_1 \cup ... \cup AC.t_y \cup ... \cup AC.t_z$ *denoting the set of actions of all transactions from* T *with*

$\prec_{AC} \subseteq (\mathbb{AC}, \mathbb{AC})$ *defining a partial order between the actions in* $\mathbb{AC}$*, and* $\prec_{\mathbb{AC}}$ *containing* $\prec_t$ *of all* $t \in T$*.*

Definition 3.4 (Committed Projection)**.** *The committed projection of a schedule sch is the set* $\mathbb{AC}^c$ *consisting of actions of committed transactions only:*

$$\mathbb{AC}^c = \cup\, AC.t_y : \forall t_y \in T^c \text{ with } T^c = \{t_y \in T^c \mid t_y.term = c\} \tag{3.1}$$

A CCM defines correctness criteria for correct schedules. A schedule *sch* can either be correct or not with regards to that CCM:

$$ccm : SCH \mapsto \{false, true\}, \text{with } SCH \text{ denoting the set of all schedules} \tag{3.2}$$

A CCP implementing a CCM must only generate correct schedules as defined by the *ccm* function:

$$\forall sch \in SCH : ccm(sch) = true \tag{3.3}$$

In what follows we will summarizes common CCMs and protocols implementing those CCMs.

Serializability

The serializable correctness model informally defines that all schedules that are `equivalent` to a serial execution of transactions are correct.

Definition 3.5 (Serial Schedule)**.** *Let* $AC.t$ *denote the set of actions of transaction* $t \in sch.T$*. A schedule sch is serial if for all pairs of actions* $ac_y, ac_z \in \mathbb{AC}$ *with* $ac_y \in AC.t_y$ *and* $ac_z \in AC.t_z$ *and* $y \neq z$*, if* $ac_y \prec_{sch} ac_z$ *then* $ac_y \prec_{sch} ac'_z$ *for all actions* $ac'_z \in AC.t_z$*.*

In a serial schedule there is no interleaving of transactions, i.e., any transaction is fully executed before the start of another transaction. We assume any serial schedule to be correct, although different serial schedules can lead to different intermediate and final data object states.

Final State Serializability

Definition 3.6 (Final State Serializability)**.** *Two schedules sch and* sch' *are said to be final state equivalent, if they result in the same final state from a given initial state, which means that they map initial database state into the same final database state [WV02]. A schedule sch is final state serializable if it exists a schedule history* sch' *so that sch is final state equivalent to* sch'*.*

View Serializability

Definition 3.7 (Reads-From)**.** *Let sch denote a schedule and* t_y*,* t_z *transaction in sch.* $r_z(o)$ *reads from* $w_y(o)$ $(y \neq z)$*, if* $w_y(o)$ *is the last write action so that* $w_y(o) <_{sch} r_z(o)$ *[WV02].*

Definition 3.8 (View Equivalence)**.** *Two schedules sch and* sch' *are view equivalent if the following conditions are met:*

1. *If t_z reads the initial value of a data object in sch, then the same applies for sch′.*
2. *If t_z reads a value of a data object written by t_y in sch (*reads-from*), then the same applies for sch′.*
3. *If t_z executes the final write on a data object in sch, then the same applies for sch′.*

Definition 3.9 (View Serializability)**.** *A schedule is view serializable if it is view equivalent to some serial schedule.*

Determining view serializability of a schedule is an NP-complete problem [WV02].

Conflict Serializability

Conflict serializability is based on the notion of conflicts between actions that is defined as follows:

Definition 3.10 (Conflicts)**.** *Two actions $ac_y \in t_y$ and $ac_z \in t_z$ are in conflict if they access the same data object and at least one of them is a write action: $(ac_y.o = ac_z.o) \wedge (ac_y.op = w \vee ac_z.op = w)$*

The conflict equivalence between schedules is defined as follows:

Definition 3.11 (Conflict Equivalence and Conflict Serializability)**.** *The schedules sch and sch′ are conflict-equivalent if the following conditions hold:*

1. *$AC.sch = AC.sch'$, i.e., both schedules have the same set of actions, and $sch.T = sch'.T$.*
2. *All pairs of conflicting actions appear in both schedules sch and sch′ in the same order ($\prec_{sch}$ and $\prec_{sch'}$).*

A schedule sch is Conflict Serializable (CSR) if it (its committed projection) is conflict-equivalent to a serial schedule. Every conflict serializable schedule is also view serializable.

A view serializable schedule that is not conflict serializable contains blind writes, i.e., writes on data object without prior reads.

Conflict serializability of a schedule can be proved if its precedence graph is acyclic [BHG87]. The precedence graph contains nodes that denote committed transactions, and edges from a node t_y to a node t_z if $\exists ac_y, ac_z : ac_y \in t_y, ac_z \in t_z$ and ac_y is in conflict with and precedes ac_z. If the precedence graph is acyclic then the serializability order can be obtained from its topological sorting.

Order Preserving Serializability

Two types of order preserving schedules are well known in literature: Order-Preserving Conflict Serializable (OCSR) and Commit Order-Preserving Conflict Serializable (COCSR) schedules [Pap79, BSW79].

Order preserving conflict serializability requires that two transactions that are not interleaved appear in the same order in a conflict-equivalent schedule [WV02].

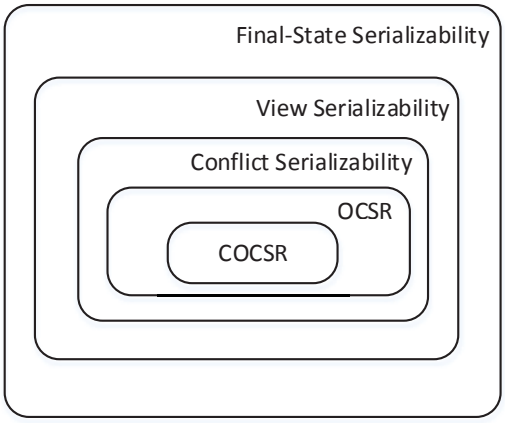


Figure 3.2: Relationship between correctness models [WV02].

Definition 3.12 (OCSR Equivalence and Order-Preserving Conflict Serializability)**.** *Two schedules sch and sch′ over the same set of transactions T are OCSR equivalent if AC.sch = AC.sch′ and for all transactions t_y, t_z, if t_y occurs completely before t_z in sch′, then the same holds also in sch. A schedule is OCSR if it is order-preserving conflict serializable equivalent to a serial schedule.*

Example 3.1 (CSR Schedule that is not OCSR)

The following schedule is CSR, but not OCSR [WV02]:

$$sch = w_1(o_1)r_2(o_1)c_2w_3(o_2)c_3r_1(o_2)c_1 \tag{3.4}$$

An equivalent serial order would be $t_3t_1t_2$. However, there is a mismatch between the serialization order and the actual execution order, since t_2 has already committed when t_3 starts.

Definition 3.13 (Commit Order-Preserving Conflict Serializability)**.** *A schedule sch is COCSR if the order of commits corresponds to the conflict-serialization order.*

Example 3.2 (OCSR Schedule that is not COCSR)

The following schedule is OCSR, but not COCSR [WV02]:

$$sch = w_3(o_1)c_3w_1(o_2)r_2(o_2)c_2r_1(o_1)c_1 \tag{3.5}$$

Strict serializability is even stronger than order preserving serializability as it requires the real time of transactions to be taken into account - it adds the strictness constraint on order-preserving serializability [HLR10].

The following relationships depicted Figure 3.2 exist between the aforementioned correctness models [Vog09]:

1. View-Serializability is a restriction of Final-State Serializability.
2. Conflict-Serializability is a restriction of View-Serializability.
3. Order-Preserving-Serializability is a restriction of Conflict-Serializability.
4. Commit-Order-Preserving Serializability is a restriction of Order-Preserving-Serializability.
5. Strictness property restricts further the Order-Preserving-Serializability.

3.1.4 Multiversion Concurrency Control

A Monoversion Database (MoVDB) denotes a database in which a write action replaces the existing value of a data object – *update in place* (Figure 3.3a). At any point in time, for each object a single value exists in the database. In contrast to MoVDBs, in a Multiversion Database (MuVDB) a write action does not replace the existing value of a data object, but instead creates a new *version* of that data object containing the new value (Figure 3.3b). We assume that each version of objects is maintained and ignore issues related to storage space. If required, purging of versions may be applied. Additionally, we assume that versions are transparent to applications unless more complex read semantics are supported that require version-awareness, which is the case with archiving applications [BS15b].

We will use the following notation to denote an action on a MoVDB: $r(o)[value]$ and $w(o)[newValue]$. A multiversion read action is defined as $r(o^v)[value]$ and returns the value of the version v of o. Each write action $w_v(o)[newValue]$ generates the version v of o that contains the $newValue$: $o^v[newValue]$, assuming that v does not exist yet, i.e., $v > v'$ for all existing versions v'.

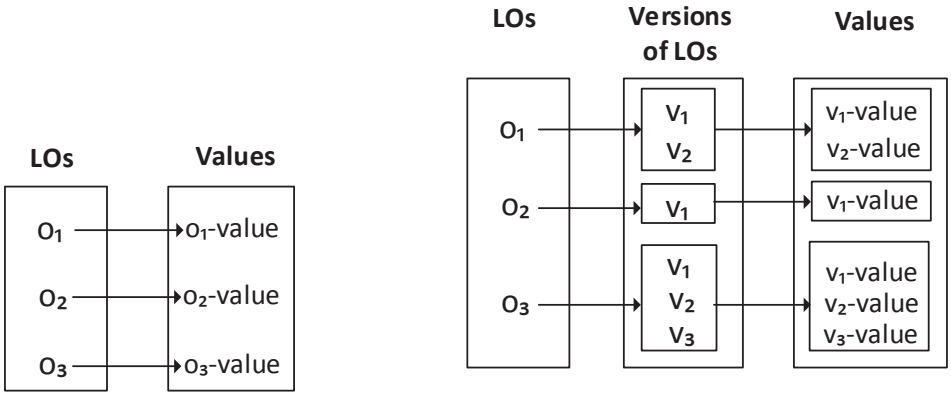
As it can be seen in Figure 3.3b, the object o_i has a corresponding set $V(o_i)$ of all versions at time τ, with each version having a concrete value associated to it. Each object has an initial (invisible) version denoted as $o_i^{initial}[NULL]$, and a latest version at time τ denoted as $o_i^{latest_\tau}$.

The definition of correctness for multiversion concurrency control is based on the assumption that the existence of versions is transparent to user transactions, which implies that they naturally expect the same behavior as in the MoVDB case.

In context of MuVDBs, we distinguish two type of histories, namely a *multiversion* history denoted as *muvh*, and *singleversion* histories denoted as *movh* [BHG87]. In *movh* interpretation of a *muvh* each read step is mapped to the last preceding write step. Notice that a *movh* represent the users' monoversion view on a MuVDB. It follows that the correctness provided to users should correspond to that of MoVDBs, the interpretation of actions by the scheduler is what changes in a MuVDB.

Definition 3.14 (Multiversion History)**.** *A multiversion history muvh for T is a partial order $\prec$ for which the following holds [WV02, BHG87]:*

1. *A translation function vf maps each $w_y(o)$ into $w_y(o^y)$, and $r_y(o)$ into $r_y(o^z)$. This means that each action of each transaction in a history is translated to a multiversion action.*



(a) Monoversion database: state at time τ (b) Multiversion database: state at time τ

Figure 3.3: Monoversion vs. multiversion databases.

2. *The ordering of actions inside a transaction is stipulated:* $\forall t_y \in T$ *and for all actions* $ac_y^i, ac_y^j \in t_y : ac_y^i \prec_{t_y} ac_y^j \implies vf(ac_y^i) \prec vf(ac_y^j)$.
3. *If* $vf(r_z(o)) = r_z(o^y)$, *then* $w_y(o^y) \prec r_z(o^y)$.
4. *If* $w_y(o) \prec_{t_y} r_y(o)$, *then* $vf(r_y(o)) = r_y(o^y)$.
5. *If* $vf(r_z(o)) = r_z(o^y)$, *and* $y \neq z$ *and* $c_z \in muvh$, *then* $c_y \prec c_z$.

A multiversion schedule is a prefix of a multiversion history.

In Definition 3.14, condition (3) defines that a version of an object can be read only when it has been produced. Condition (4) defines that if a transaction writes o before it reads o, then it must read that version o that it previously created. Condition (5) defines that before a transaction t commits, all transactions that t read from must already have committed.

Serializability

Definition 3.15 (Multiversion Reads-From)**.** *In a MuVDB a transaction* t_z *reads* o *from* t_y *if* t_z *reads the version* o^y *produced by* t_y.

Definition 3.16 (Multiversion Equivalence)**.** *Two mvh are equivalent if they have the same operations and the same* reads-from *relationships. A history mvh is view serializable if there exists a serial monoversion history movh for the same set of transactions such that mvh is view equivalent to movh, i.e., has the same* reads-from *as the monoversion history mvh.*

As described in [WV02], it is not enough to require the view equivalence of a multiversion schedule to a serial multiversion schedule, since a serial multiversion schedule must not have *reads-from* compatible with a serial monoversion schedule. The problem of deciding whether a multiversion history in multiversion-view serializable is NP complete [WV02].

Definition 3.17 (Multiversion Conflict)**.** *A multiversion conflict in a multiversion schedule mvsc is a pair of actions r_z and w_y such that $r_z(o^i) \prec_{mvsc} w_y(o^y)$.*

Notice that a multiversion conflict is defined on a data object and not on the versions of data objects. Two write-actions $w_z(o^z)$ and $w_y(o^y)$ are not in conflict since both would generate unique versions.

Definition 3.18 (Multiversion Conflict Serializability)**.** *A mvh is multiversion conflict serializable if there is a serial monoversion history with the same transactions and the same ordering of multiversion conflicting actions.*

Multiversion conflict serializability has polynomial-time membership problem [WV02]. A multiversion schedule is multiversion conflict serializable if its multiversion conflict graph is acyclic. The multiversion conflict graph has transactions as nodes and an edge from t_z to t_y if t_z reads the same object which is written by t_y and read comes before write (so there is a multiversion conflict).

SI

SI is a popular correctness model for MuVDBs, in which reads never block writes and vice-versa [BBG+95]. It is included in major databases, such as Oracle [oraa] and PostgreSQL [siP]. In Oracle it is even the strongest correctness model provided and is called as serializable CCM.

Definition 3.19 (Snapshot Isolation)**.** *Let $start(t)$ denote the start time of transaction t and $commit(t)$ its commit time. A multiversion schedule of transactions T satisfies the SI correctness if following holds [WV02]:*

- *A read action $r_y(o)$ is mapped to the most recent committed write action $w_z(o)$ as of $start(t_y)$.*
- *The write sets of two concurrent transactions are disjoint. Two transaction t_y, t_z are concurrent if their life-times overlap, i.e., if either $start(t_y) < start(t_z) < commit(t_y)$ or $start(t_z) < start(t_y) < commit(t_z)$.*

The increased concurrency of SI compared to the serializable model comes at a cost. It is well-known that SI permits certain non-serializable transaction execution, namely the write-skew inconsistency (constraint violation) may occur in SI-based schedules [BBG+95, WV02] (Example 3.3).

Example 3.3 (Write-Skew Inconsistency)

Let us assume two objects $o_i = 5$ and $o_j = 10$ and the constraint $o_i + o_j \geq 10$. Transaction t_1 reads o_i and o_j. t_2 reads o_i and o_j and then sets the value of o_j to 0. Afterwards, t_1 updates o_j to 5 leading to the write-skew inconsistency as the result violates the constraint $o_i + o_j = 5 < 10$.

3.1.5 Concurrency Control Protocols

The *scheduler*, a module of the DBMS, is responsible for ensuring that the generated schedules are correct as defined by the correctness model. However, checking for equivalence can be a very time consuming task. For example, in case of u transactions, in the worst case $u!$ serial schedules need to be checked. Instead of checking for equivalence, each transaction should follow a CCP and that CCP ensures the generation of correct schedules only.

The *scheduler* forwards received actions to a CCP, which then outputs correct schedules with regards to the correctness model. Notice that a transaction may be aborted due to reasons external or internal to the CCP. External reasons are mainly failures, which arise outside the *scheduler*. In that case, the termination action is simply forwarded to the CCP. A CCP can also decide to abort a transaction if it decides that an action may lead to a violation of the correctness model. Depending on when the violation test is done, CCPs can be classified as *pessimistic* or *optimistic*.

Pessimistic CCPs check for possible violations at the time they receive an action. If the execution of an action does not violate the correctness criteria, that action is forwarded to the underlying layers of the DBMS for execution; otherwise the corresponding transaction is blocked or aborted. Optimistic CCPs, also known as certification-based CCPs, check for violations once the commit action is received for a transaction. If the execution of the commit would lead to a correctness violation, then one or more transactions are aborted.

In what follows we will provide a summary of CCPs for single-copy DBSs.

2PL

In lock-based protocols the *lock* mechanism is used as a means to control access to data objects. There two types of locks, *exclusive (X)* and *shared(S)* locks. The owner of an exclusive lock on data object can read and write that data object, whereas if only a shared lock is owned that data object can only be read but not written. A lock on a data object can be acquired only if the requested lock is *compatible* with already held locks on that data object. The compatibility matrix is defined as follows:

	SHARED	EXCLUSIVE
S	TRUE	FALSE
X	FALSE	FALSE

Table 3.2: Lock compatibility matrix.

Each transaction before it can execute an action, has to acquire the corresponding lock on the object. For a read action a shared lock has to be acquired, whereas for a write action an exclusive lock. The locks are released immediately after the operation is executed. Such a simple protocol would not forbid the well-known anomalies [WV02].

2PL is a well-known lock-based protocol that ensures conflict-serializable schedules. The Basic Two-Phase Locking (B2PL) has two phases, namely a growing phase and a shrinking phase. In the growing phase, transactions may obtain locks, but are not

allowed to release any lock. In the shrinking phase, transactions release locks and are not allowed to obtain any new lock. It is well-known that B2PL can generate deadlocks, a situation in which no transaction can proceed as they all wait for locks held by the other transactions.

Conservative Two-Phase Locking (C2PL) requires that each transaction acquires all locks at the beginning. If a transaction cannot acquire all locks, then it does not hold any lock. It is part of the implementation detail if the transaction is immediately aborted, retries the lock acquisition or is put in a wait queue. Similar to the B2PL, locks are released regularly in phase two.

Strict Two-Phase Locking (S2PL) releases shared locks regularly in phase two, whereas the exclusive locks are released only after the end of the transaction. Strickt Strong Two-Phase Locking (SS2PL) keeps all locks, shared and exclusive ones, until after the end of a transaction. SS2PL can be used to generate COCSR schedules. In general, there are conflict-serializable schedules that can not be obtained when 2PL is used [WV02].

Timestamp-Ordering (TO)

In timestamp-based protocols each transaction will receive a unique timestamp denoted as $ts(t)$. All operations of t inherit the $ts(t)$. A timestamp-based schedule has to order conflicting actions based on the timestamps of their transactions. For two conflicting actions $ac_y(o)$ and $ac_z(o)$ the following has to hold:

$$ac_y(o) \prec ac_z(o) \Leftrightarrow ts(t_y) < ts(t_z)$$

The basic approach to enforce the TO rule is as follows [BHG87]:

- Each accepted action is immediately forwarded for execution in first-come-first-serve order.
- An action is only rejected if it comes too late. An action $ac_y(o)$ comes too late, if another conflicting action $ac_z(o)$ with $ts(t_z) > ts(t_y)$ has already been processed. In that case t_y is aborted.

3.1.6 Concurrency Control and Recovery

While CC addresses the isolation of transactions, it is also necessary to consider the recovery of transactions in case of failures in order to ensure the atomicity and durability guarantees. Different failures may occur during the transaction execution, such as failures in the operating system or DBS, transient failures, media failures and others. In what follows we will focus on the aspect of transaction recovery, which is needed to implement transaction rollbacks in case of failures, i.e., to *undo* the effects of transactions in presence of concurrent transactions. The atomicity property requires the ability to completely undo the effects of an aborted transaction (Example 3.4).

Recoverable Schedules

Example 3.4 (Impossible Recoverability)

Let us consider the following schedule [WV02]:

$$w_1(o_1)r_2(o_1)c_2a_1$$

In the depicted schedule t_2 reads o_1 that was written by t_1 and commits. Therefore, t_1's effects cannot be undone. Although acceptable by a serial scheduler, such a schedule should be avoided as it violates the atomicity guarantee.

Based on the Example 3.4 we can conclude that correctness models are needed that guarantee both serializability and transaction recovery.

Definition 3.20 (Recoverable Schedules)**.** *Let $RF(sch)$ capture all* reads-from *relations between transactions in sch (see Definition 3.7):*

$$RF(sch) = \{(t_y, o, t_z) | r_z(o) \text{ reads-from } w_y(o)\}$$

The schedule sch is recoverable, if the following condition holds: $\forall t_y, t_z \in sch, \forall (t_y, o, t_z) \in RF(sch) \wedge y \neq z \Rightarrow c_y \prec_{sch} c_z$.

Schedules that Avoid Cascading Aborts

As depicted in Example 3.5, recoverable schedules do not avoid cascading aborts, which may occur if a transaction reads values from a transaction that aborts.

Example 3.5 (Cascading Aborts)

Consider the following schedule: $w_1(o_1)r_2(o_1)w_2(o_2)a_1$. As t_2 reads from t_1 and t_1 aborts it is necessary to also abort t_2.

The example above necessitates the definition of schedules that avoid cascading aborts.

Definition 3.21 (Avoiding Cascading Aborts)**.** *A schedule sch avoids cascading aborts if the following condition holds: $\forall (t_y, o, t_z) \in RF(sch) \wedge y \neq z \Rightarrow c_y \prec_{sch} r_z(o)$.*

Strictness

In order to allow undo of actions, usually the *before-image* of objects are written. However, that does not guarantee correct undo of effects as shown by the following example:

Example 3.6 (Incorrect Undo)

Let us consider the following schedule: $w_1(o_1)w_2(o_1)c_2a_1$. Applying the before image as a consequence of the t_1's abort may lead to t_2's effects being undone.

Definition 3.22 (Strict Schedule)**.** *A schedule sch is strict if the following applies:* $w_z(o) \prec_{sch} ac_y(o) \wedge y \neq z \Rightarrow a_z \prec_{sch} ac_y(o) \vee c_z \prec_{sch} ac_y(o)$.

Rigorousness

While a strict schedules avoids write/read and write/write conflicts between uncommitted transactions, rigorous schedules additionally avoid read/write conflicts between uncommitted transactions.

Definition 3.23 (Rigorous Schedules)**.** *A schedule sch is rigorous if the following condition holds:* $\forall r_z(o) \prec_{sch} w_y(o), y \neq z \Rightarrow a_z \prec_{sch} w_y(o) \vee c_z \prec_{sch} w_y(o)$.

2PL and Transaction Recovery

Both B2PL and C2PL described in Section 3.1.5 generate schedules which are not free of cascades. S2PL generates strict schedules, whereas SS2PL generates exactly the class of rigorous schedules [WV02].

3.2 Distributed Database Systems

Symbol	Description
S	Denotes the set of sites.
s	Denotes a site: $s \in S$.
$PC(o_i)$	Denotes the set of physical copies of o_i.
$pc_{i,j}$	Denotes the physical copy of o_i hosted at site j: $pc_{i,j} \in PC(o_i)$.
rep_{full}	Denotes the mapping of logical objects to sites in a fully replicated DDBS.
$rep_{partial}$	Denotes the mapping of logical objects to sites in a partially replicated DDBS.
rep_{full}	Denotes the mapping of logical objects to sites in a fully replicated DDBS.
$PART$	Denotes the set of data partitions.
$part$	Denotes a data partition: $part \in PART$.
σ_{Pr_i}	Denotes the selection operator applied on a set of logical objects. The operator will select only those objects that satisfy the predicate Pr_i.
Π_{ATT_i}	Denotes the projection operation on a set of attributes ATT_i. The operator will select all values contained in the attributes.
$rstart(t)$	Denotes the real start time of transaction t.

Table 3.3: Distributed database model: symbols and notations (see also Table 3.1).

In what follows we will extend the single-copy database model described in Section 3.1 to the distributed database case. The list of symbols and notations is summarized in Table 3.3.

A *distributed database system* consists of a set of sites $S = \{s_1, s_2, s_3, ...\}$, that host the database, i.e., the logical objects (LO), and the software that manages the data, which is denoted as a DBMS. We assume the sites to have identical (homogeneous) DBMSs.

As described in [ÖV11], both properties, namely the distribution of data and the existence of network, distinguish DDBS from single-copy DBS. These properties, despite the clear advantages of DDBSs, lead to problems that are fare more complex with regards to desired guarantees, such as data consistency, compared to single-copy DBSs (Figure 3.4).

There are different reasons that have led to the development of DDBSs, ranging from availability, performance, scalability and elasticity, to economical and sociological aspects, such as assigning responsibility of data to units of an enterprise, that provides them with all necessary resources to accomplish their tasks, and increases their responsibility towards business [d'O77].

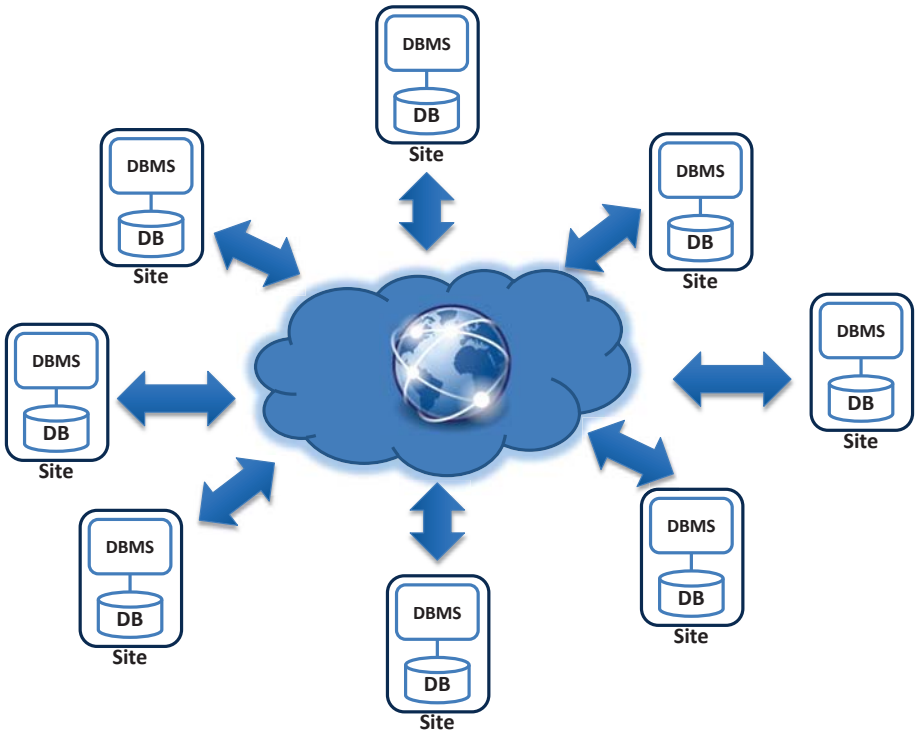


Figure 3.4: Distributed database system.

3.2.1 Data Distribution Models

DDBSs differ in the distribution model of logical objects to sites, which is closely related to the complexity degree of the functionality required for providing the desired data guarantees, such as availability and scalability.

Based on the distribution model, each logical object o_i is mapped to a set of physical copies $PC(o_i) = \{pc_{i,j}, pc_{i,k}, \cdots, pc_{i,r}\}$ with $pc_{i,j}$ denoting the physical copy of o_i hosted at site j.[1] In what follows we will define the common distribution models, which are depicted in Figure 3.5.

Data Replication

Definition 3.24 (Full Replication)**.** *In a fully replicated DDBS each object is present at each site. The mapping function of logical objects to sites is defined as follows:*

$$rep_{full} : LO \mapsto \underbrace{S \times \cdots \times S}_{|S| \ times}$$

Definition 3.25 (Partial Replication:)**.** *In a partially replicated DDBS, each logical object is available only at a subset of sites (Equation (3.25)). We distinguish between pure partial replication and hybrid partial replication. In a pure partial replication no site contains all data objects,*

[1] In our model we assume that each logical object is mapped to at least one site.

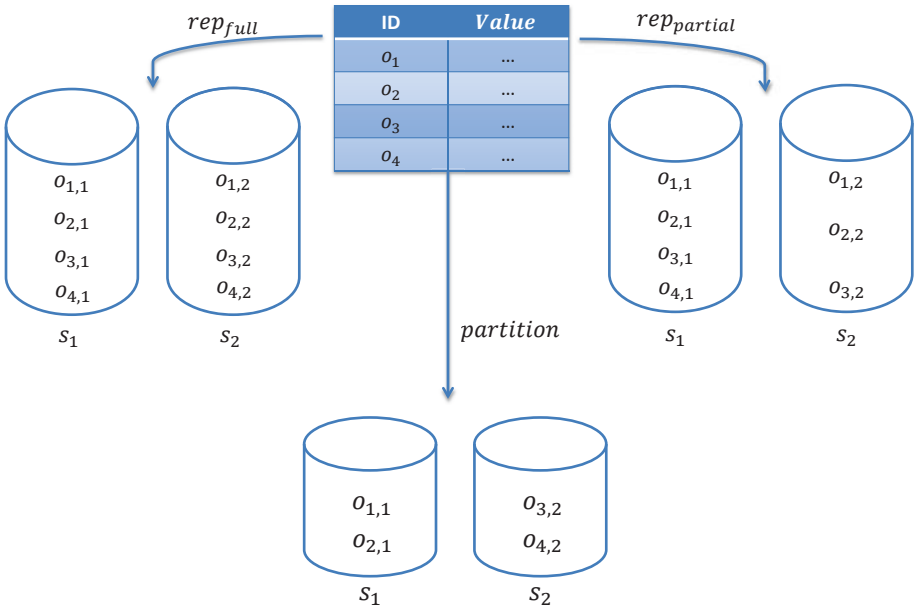


Figure 3.5: Logical objects and their mapping to sites.

whereas hybrid partial replication means that some sites contain copies of all and others contain a copy of only a subset of data objects:

$$rep_{partial} : LO \mapsto \mathcal{P}(S) \setminus \{\{S\}\}$$

Data Partitioning

The goal of data partitioning is to generate subsets of data denoted as partitions: $PART = \{part_1, part_2, \dots\}$. The partitions should satisfy the following conditions:

- Completeness. The decomposition of logical objects to partitions is complete iff: $\forall o \in LO \; \exists part \in PART : o \in part$.
- Reconstruction. There should exist an operator ∇ that reconstructs LO from $PART$.
- Disjointness. The partitions should be disjoint, i.e., $\forall i, j \wedge i \neq j : part_i \cap part_j = \emptyset$.

There are three basic approaches to partition a database, namely horizontal, vertical and hybrid [ÖV11]. In what follows we will provide the definitions for each of the different approaches based on the relational model. We assume that the object values have been decomposed into a set of attributes: $ATT = \{att_1, att_2, \dots, att_n\}$.

Definition 3.26 (Horizontal Partitioning)**.** *Horizontal partitioning splits data across partitions based on predicates applied to a set of attributes. In terms of the relational model, horizontal partitioning can be defined as a selection on the logical objects [SKS05]:*

$$part_i = \sigma_{Pr_i}(LO)$$

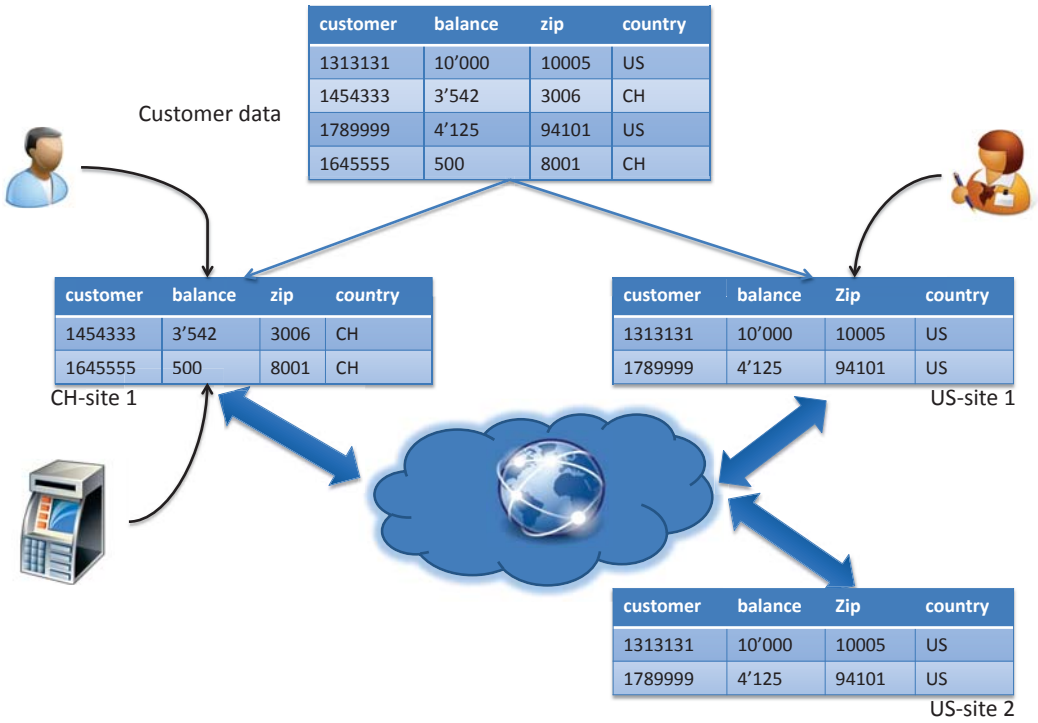


Figure 3.6: Banking application with partitioned and replicated data.

Definition 3.27 (Vertical Partitioning). *In the vertical partitioning, each partition is defined as a projection on a subset of attributes [SKS05]:*

$$part_i = \Pi_{ATT_i \subset ATT}(LO)$$

Definition 3.28 (Hybrid Partitioning). *Hybrid partitioning consists of a horizontal partition followed by a vertical partitioning, or a vertical partition followed by a horizontal partitioning:*

$$part_i = \begin{cases} \sigma_{Pr_i}(Pi_{ATT_i}(LO) \\ \Pi_{ATT_i}(\sigma_{Pr_i}(LO)) \end{cases}$$

As our data model is agnostic to the concrete decomposition of the object values, we will stick to the horizontal partitioning based on the o_{id}, i.e., the predicate Pr_i is applied on o_{id} (Figure 3.5).

Once the partitions are determined, the next task is their mapping to sites, which is also known as *allocation*:

$$allocation : PART \mapsto S$$

The *allocation* function is surjective, i.e., each site will get a partition and some sites may get multiple partitions assigned. By mapping $PART$ to $\mathcal{P}(S)$ it is possible to neglect some sites during the allocation process.

3.2.2 Advantages and Challenges of Distributed Database Systems

Let us consider a simple bank with branches at two different locations, for instance in Zurich and New York. Clerks in Zurich must have access to Swiss customers, i.e., cus-

tomers having a Swiss zip code, and those in New York must access customers residing in the US in order to perform their tasks. In a centralized DBS, depending on the geographical location of the DBS, either the Zurich' clerks or the New York' clerks or both would incur a high network latency for accessing their customers. Such a latency may become a major bottleneck to performance [BDF+13].

However, as depicted in Figure 3.6, we can partition the data and place all Swiss customers to a DBS located in Zurich and all US customers to a DBS located in New York. This would provide satisfiable performance to the corresponding clerks. Moreover, by replicating certain data partitions, we can increase the availability of particular data in order to satisfy business or legal requirement. Furthermore, in case of an increased load, i.e., increased number of clerks accessing the data, the DDBS can add additional sites and incorporate these sites in the load distribution.

In summary, DDBSs provide clear advantages compared to centralized DBSs. However, clients, i.e., applications should be shielded from the underlying low-level details of the system, which means that the DDBS should provide *transparent* data management. The effort for ensuring transparency in DDBSs is considerably higher compared to that of centralized DBSs. This is related to the fact that data is not available at a single site, which necessitates a network communication between the sites. Moreover, site and network failures may occur.

If we consider the banking application depicted in Figure 3.6, the clerks should be provided with data transparency that can be materialized as follows:

- **Partitioning transparency**. The clerks should not be aware how the data has been partitioned.
- **Location transparency**. The clerks should not be aware of the physical location of objects.
- **Replication transparency**. The DBS may replicate single objects, a subset of objects or all objects to increase availability or performance. The existence of multiple copies an object should not be of concern to the clerks.

Distributed Transactions

A *distributed transaction* is a transaction that accesses data residing on at least two different sites. Distributed read-only transactions consist of read actions that access objects residing at different sites, and distributed update transactions must commit their changes at two or more sites. As depicted in Figure 3.7 the actions of a global (distributed) transaction are mapped to local transactions consisting of actions on physical copies of the objects residing at a specific site. The global transaction t must have a termination action in every local (sub-) transaction, that must globally be consistent. This means that the same termination action must be present at each local transaction and this requires a coordination between sites (local sub-transactions). In order to achieve a global consensus on the faith of a distributed transaction by considering various failures a *distributed commit protocol* must be used [WV02]. 2PC is the most widely utilized protocol for the coordination of distributed transactions. It introduces a *transaction coordinator* that mediates between the transaction commit and the participating sites – also

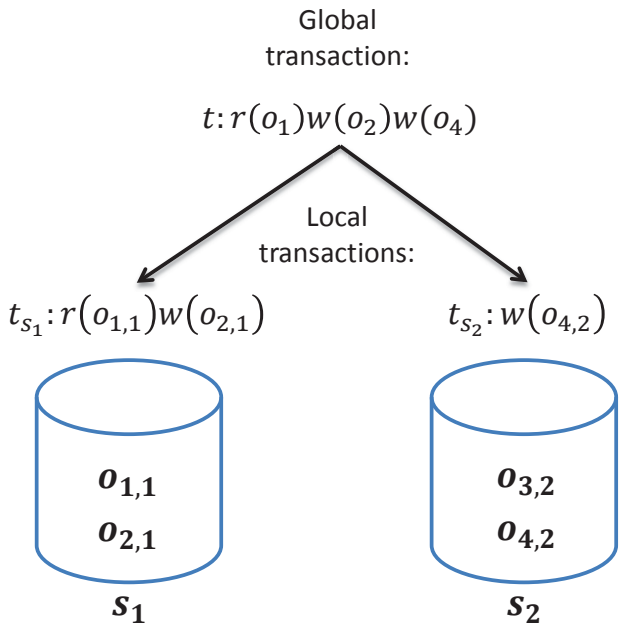


Figure 3.7: Local vs. global transactions.

referred to as *agents* – in two phases, namely the *preparation phase* and the *commit phase* (Figure 3.8):

- *Preparation phase* – In the first phase the coordinator will ask every agent whether it is ready commit a transaction by sending a *PREPARE* message. Each agent will reply with *YES* or *NO* vote depending on whether it is able to commit or not. After an agent has voted, its *uncertainty phase* begins, and the agent is not allowed to unilaterally take any decisions prior to receiving the final decision from the coordinator.

- *Commit phase* – In the second phase the coordinator collects all votes and will decide based on them on the fate of the transaction. If all agents have voted with *YES*, the decision is to commit, otherwise to abort the transaction. The agents are notified via a *COMMIT* or *ABORT* messages about the final decision. Upon reception decision, each agent will either commit or abort its local transaction, and acknowledge that with an *ACK* message to the coordinator.

2PC guarantees the *safety property*, i.e., that "nothing bad will ever happen" [WV02]. This means that all sites of a DDBS will never decide on a different outcome. However, it does not guarantee the *liveness property* in case of failures. It is well known that 2PC is a blocking protocol.

Let us assume that the coordinator fails after having received the votes. During that period of time, the agents remain blocked (uncertain) waiting for the decision. This has considerable impact on the overall availability of the system as all resources must remain locked during the uncertainty phase. By the use of the cooperative termination protocol the agents can reach a decision even if the coordinator is unavailable. However,

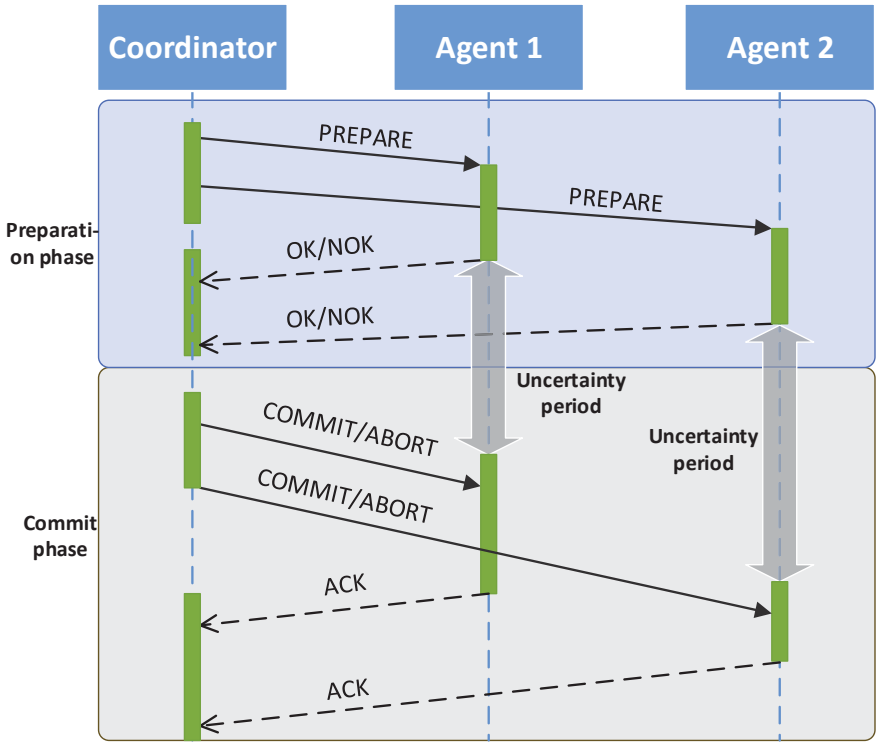


Figure 3.8: Message flow in 2PC between the coordinator and two agents.

the precondition is that at least one of the agents received the decision. If all of them are uncertain, then the cooperative termination protocol causes considerable extra costs, without being able to reach a decision [BG84].

Three-Phase Commit (3PC) protocols avoids blocking in case of site failures only at an additional message overhead. Moreover, it may violate agreement (safety) in case of link failures [WV02]. Paxos is another agreement protocol that is safe and largely-live, i.e., avoids blocking if less than half of the sites fail [Lam02].

Data distribution does however not per se lead to distributed transactions. Let us assume that the data has been partitioned in such a way that it perfectly matches the access patterns of transactions, which are defined by the read and write sets of transactions. By a perfect match we mean that each single partition can serve transactions without a coordination with any other partition, i.e., each transaction is non-distributed. In that case, data management challenges are similar to those of a single-copy DBS.

3.2.3 Concurrency Control for Distributed Databases

With respect to concurrency control for distributed databases we distinguish between *local* and *global* correctness. By local correctness we mean the correctness provided at a particular site, whereas global correctness is the system-wide correctness. Local correctness does not imply global correctness [WV02, BHG87].

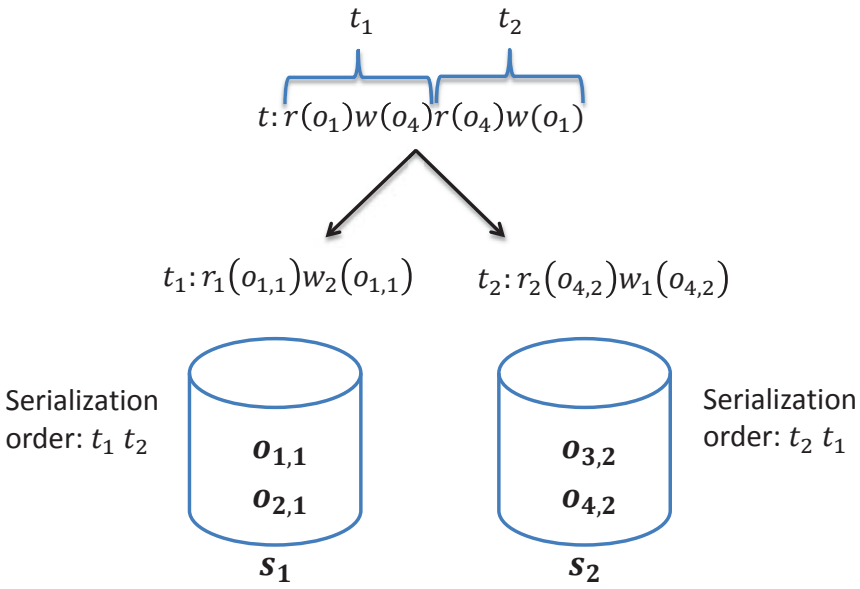


Figure 3.9: Local vs. global correctness.

Example 3.7 (Local vs. Global Correctness)

In the scenario depicted in Figure 3.9, a transaction t is submitted for execution to a distributed (partitioned) DBS. Transaction t is split in two sub-transactions t_1 and t_2 that are serialized differently at the sites. While site s_1 serializes them as t_1 t_2, s_2 decides for the opposite order, namely t_2 t_1. From the local point of view, the execution is correct. However, from the global point of view it is incorrect.

The globally provided correctness guarantees may be stronger, weaker than or equivalent to the local correctness guarantees. Example 3.7 depicts a scenario in which the global CCM is weaker than the locally provided CCMs. In [BHEF11a] the authors provide a protocol that guarantees global serializability on top of a systems with sites providing locally only SI correctness. [SW00] follows similar goals, by providing a protocol that guarantees serializability for federated databases, when the sites use different correctness models and protocols.

Distributed 2PL

The 2PL protocol for single-copy systems can be easily extended for the DDBSs. Now, each transaction must transmit lock requests and unlock request to each of the sites in the DDBS in a consistent manner. It should be avoided that a transaction releases locks at one site and obtaining locks at other sites. Such a Distributed 2PL implies global knowledge, i.e, that each site has knowledge on local schedulers. Compared to a centralized 2PL the message overhead is considerable. Moreover, based on the variant of 2PL deadlocks are possible, and their detection and resolution becomes more complex compared to the single-copy case.

Distributed Timestamp-Ordering

As already described for the centralized case, timestamp-ordering requires that if two operations are in conflict, then they are ordered based on the timestamps of the corresponding transactions. This part of the protocol remains the same also for DDBSs. However, the assignment of timestamps is more difficult. So, one can have a dedicated site in the system responsible for assigning timestamps or do the timestamp assignment in a distributed manner by using for example lamport clocks. Their global uniqueness can be achieved by adding the site id as a prefix to the local lamport clock [TvS07a].

3.3 Distributed Database Systems with Replication

Symbol	Description
$freshness(o, v, w)$	Defines how outdated or recent is o^v with regards to another version w of o.
$latest_\tau(o)$	Denotes the most recent version of o as of time τ.
$s.LO$	Denotes the set of logical objects hosted by the site s.
$t.LO$	Denotes the set of objects accessed by transaction t. $t.LO = RS(t) \cup WS(t)$.
$R.All(o_i)$	Denotes the set of all read actions applied to all physical copies of o_i.
$W.All(o_i)$	Denotes the set of all write actions applied to all physical copies of o_i.
$wts(o)$	Denotes the write timestamp of object o.
$rstart(t)$	Denotes the real start time of transaction t.

Table 3.4: Distributed database with replication: symbols and notation.

A distributed database, in which multiple copies of data objects are available is called a *replicated* DBS (see Section 3.2.1). The main reasons for data replication are as follows [ÖV11]:

Data availability. By storing multiple copies of all or subset of data, the system becomes resilient to failures. Even if some of the sites become unavailable, data can be accessed from other sites.

Performance. Data replication allows the placement of data closer to the query, i.e., to the user accessing the data. This is especially beneficial for large-scale applications that serve users residing at different geographical locations.

Scalability. The available copies can be used to distribute the load. If the load increases, new copies can be deployed and used for load distribution, so that the response time remains acceptable.

3.3.1 Data (In-) Consistency

In addition to the inconsistencies for single-copy databases described in Section 3.1, in DDBS with replication, relaxing ACID guarantees, may result in additional inconsistencies, which are materialized as follows:

Recency violation. Different sites hosting replicas of an object may, at least temporarily, reflect different states. This inconsistency may have a cascading effect if transactions update object values based on stale values and may lead to other inconsistencies, such as the lost-update or constraint violation. It should be noted that recency violations are inherent to replicated databases, as in non-replicated ones there is always a single copy of an object that reflects the most recent value.

Inconsistent ordering of updates. In these case different sites reflect different views on what the value of an object should be. Inconsistent ordering can be *temporal* or *permanent*. The previous case may be a result of either the impossibility of communication between sites, or the temporal avoidance of coordination in order to not penalize availability and performance. However, the sites will eventually converge towards the consistent state. In the latter case, there is a missing consensus on what the final value of an object should be.

3.3.2 Data Freshness

As we described in Section 3.1.2, in DDBSs with replication, sites may, in certain points in time, reflect different data states. This divergence between the hosts may lead to situations in which applications access stale data, i.e., do not access that version of data objects that represents the most *recent* value as of time τ.

In what follows we will provide a model capturing recency in replicated DBSs. Let us assume that a *timestamp oracle* assigns a system wide unique timestamp to transactions [2]. Each update transaction assigns at commit its timestamp to all objects that it has modified. The *oracle* is free to generate the timestamp using different strategies, such as for example using its local physical clock, a simple counter, etc. Our recency model requires however the generated timestamps to be strictly monotonic increasing.

Definition 3.29 (Object Freshness)**.** *Let $latest_\tau(o)$ denote the globally most recent (committed) version of o at time τ (Table 3.4). The freshness of an object o_i with the timestamp (version) v defines how outdated or recent o^v is with regards to $latest_\tau(o)$:*

$$freshness(o, v) = \begin{cases} 1 & \text{if } v > latest_\tau(o) \\ \frac{v}{latest_\tau(o)} & \text{otherwise} \end{cases}$$

We can now extend the definition of freshness at the level of sites as follows:

[2]The timestamp must not reflect the real time of transactions. For example, two transactions t_1 and t_2 may in real-time start in the sequence t_1 t_2. However, as t_2 may reach first the timestamp oracle it will also get a lower timestamp than t_1.

Definition 3.30 (Site Freshness)**.** *Let $s.LO$ denote the set of objects hosted by s, o^v denote the current version (at time τ) of $o \in s.LO$ at s, and $latest_\tau(o)$ denote the globally latest version of o. Then, $freshness(s)$ defines how recent s is with regards to the globally most recent versions of the data it hosts:*

$$freshness(s) = \arg\min_{o \in s.LO} freshness(o, v)$$

Similar to the recency of a site with regards to all objects it hosts, we can define the recency of a site with regards to a transaction t, i.e., objects accessed by t:

Definition 3.31 (Site Freshness with regards to a Transaction)**.** *Let $t.LO$ denote the set of objects accessed by a transaction t. The freshness of site s with regards to t is defined as follows:*

$$freshness(s, t) = \arg\min_{o \in t.LO \wedge o \in s.LO} freshness(o, v)$$

In existing DBSs, such as SimpleDB [sima], often, data freshness is defined in terms of an upper bound of the inconsistency window ($inconsistencyWindow \in [0, \infty[$), which defines the time frame in which sites diverge with regards to some data objects. An $inconsistencyWindow = 0$ means that the state of objects at different sites are guaranteed to represent the most recent versions. The bigger the $inconsistencyWindow$ the lower the freshness of outdated objects, with $freshness \to 0$ as $inconsistencyWindow \to \infty$.

3.3.3 Replication Protocols

The presence of multiple object copies requires a mapping of an action to an action on a set of physical copies. This task is commonly referred to as Replica Control (RC) [KJP10] and is implemented by a RP.

An RP defines the mapping of actions on logical objects to actions on physical copies:

Definition 3.32 (Mapping of Actions on *LO*s to Actions on *PC*s)**.** *Let $R.All(o_i) = \{r(pc_{i,1}), r(pc_{i,2}), \ldots, r(pc_{i,|S|})\}$ denote the set of read actions applied to all physical copies, and $W.All(o_i) = \{w(pc_{i,1}), w(pc_{i,2}), \ldots, w(pc_{i,|S|})\}$ the set of write actions applied to all physical copies. The concrete mapping of an action to the physical copies is defined by the RP:*

$$r(o_i) \mapsto \mathcal{P}(R.All(o_i))$$
$$w(o_i) \mapsto \mathcal{P}(W.All(o_i))$$

Concurrency control and replica control are not orthogonal. Moreover, they can not be isolated from other tasks, such as load balancing. The goal of load balancing is to choose the optimal site for executing a transactions based on their load, capacity, network distance, monetary cost, etc. Clearly, load balancing strategy interferes with concurrency and replica control, that determine the overall consistency guarantees. For example, assuming strong consistency requirements, transactions can be load balanced only between sites that are up-to-date.

RPs can be classified according to *where* transaction are coordinated and *when* the results of updates are propagated to other replica sites in the system [GHOS96]. According to the first parameter an RP can either be a *primary copy* or *update-anywhere*. In

the primary copy approach there is a dedicated replica site responsible for the transaction coordination, whereas in the update anywhere approach a transaction can be executed at any site. According to the second parameter an RP can be *eager* or *lazy*.

In what follows we will summarize RPs based on the *when* parameter by considering only update-anywhere approach, which is the most challenging, but also the most interesting approach.

Read-One-Write-All and Read-One-Write-All-Available (ROWA(A)) RPs

In the ROWA approach a read can access a single physical copy, whereas a write must eagerly update all sites:

$$r(o_i) \mapsto R.All(o_i)$$
$$w(o_i) \mapsto \{\{W.All(o_i)\}\}$$

An eager update means that the transaction must ensure that affected set of sites receive the updates before the result is returned to the client. In contrast to ROWA, the ROWAA approach will eagerly update only the available sites. This means that, sites that are not available at the time of update propagation, will receive the updates after their recovery. This increases the availability of writes, and the cost of complex reconciliation once the failed sites recover.

Quorum-based RPs

In quorum-based RPs, only a subset of replica sites is updated eagerly. However, the subsets must be chosen in such a way that any two writes or a write and read on the same data object overlap. This is known as the *intersection property*. For quorum-based RPs, the intersection property is crucial as it allows a consistent decision taken by a subset of sites on behalf of all sites [JPAK03], which, in turn, is necessary for guaranteeing access to most recent data. It is well known that quorum-based RPs have a lower overhead compared to ROWA(A) for writes at the cost of increased overhead for reads [JPAK03]. Reads must access a subset of sites that form a read quorum. Based on timestamps that are attached to the updates, it is possible to determine the most recent version in a read quorum, which is then also guaranteed to be globally the most recent one due to the intersection property. The mapping of reads and writes is defined as follows:

$$r(o_i) \mapsto R(o_i) \in \mathcal{P}(R.All(o_i)) \wedge |R(o_i)| \leq |S|$$
$$w(o_i) \mapsto W(o_i) \in \mathcal{P}(W.All(o_i)) \wedge |W(o_i)| \leq |S|$$
$$R(o_i) \cap W(o_i) \neq \emptyset$$

In context of quorum-based, RPs $R(o_i)$ denotes the *read quorum* (rq) of o_i and $W(o_i)$ the *write quorum* (wq) of o_i.

Different quorum RPs have been developed, such as the Majority Quorum (MQ) [Tho79, Gif79] or the Tree Quorum (TQ) [AE90]. All have the intersection property in common, but they differ on the costs generated for transaction execution. Additionally,

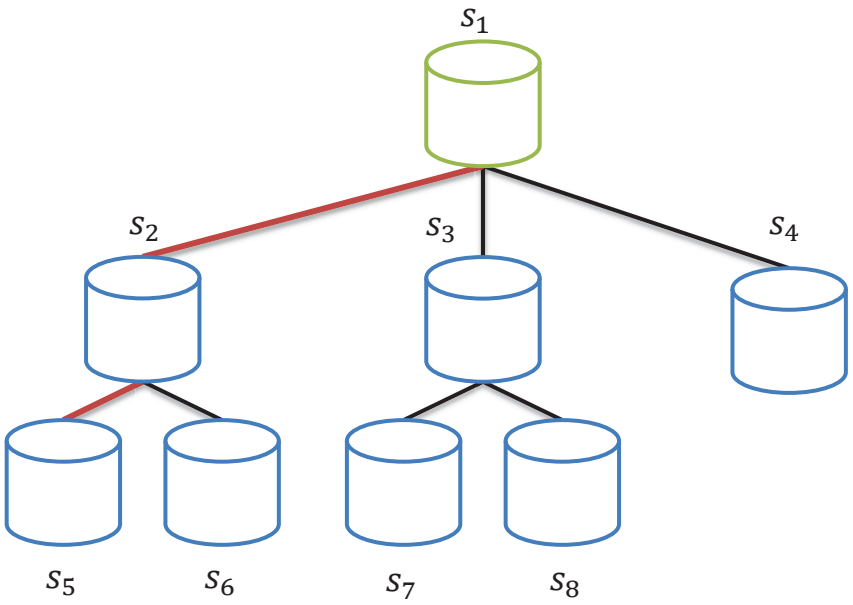


Figure 3.10: Example of quorum construction using LWTQ: $rq = \{s_1\}$ and $wq = \{s_1, s_2, s_5\}$.

they differ in the overhead generated for organizing the sites in a certain logical structure, which is a precondition for the intersection property in some protocols, and for maintaining that structure [JPAK03]. Typically, the quorums are the same for all data objects: given $q \in \{wq, rq\}$, then $q(o_1) = q(o_2) = \cdots = q(o_{|LO|})$.

Quorum RPs can be static or dynamic with regards to quorum construction. In static protocols the quorums do not change except in cases of site failures, whereas dynamic protocols are able to adapt the quorums to, for example, application workload.

MQ The MQ protocol is a simple quorum-based RP, in which each site s is assigned a number of votes $votes(s) \geq 0$. The quorums are then chosen in such a way so that they exceed half of total votes [Tho79, Gif79, JPAK03]:

$$\begin{aligned} \#votes &= \sum_{s \in S} votes(s) \\ wq &= \lfloor \frac{\#votes}{2} \rfloor + 1 \\ rq &= \lfloor \frac{\#votes + 1}{2} \rfloor \end{aligned} \tag{3.6}$$

In the simplest form, each site has the same amount of votes, all with the same weight. It follows that wq can be created from a majority of sites, whereas an rq by half of the sites if $|S|$ is even, or majority of sites if $|S|$ is odd.

TQ In the TQ protocol, the sites are logically organized in a tree structure with a *degree*, which defines the maximum number of children for each site in the tree, and *height*, which defines the longest path from the tree root to a leaf node. A tree quorum $q = \langle l, ch \rangle$ is constructed by selecting the root of the tree, and adding ch children to the

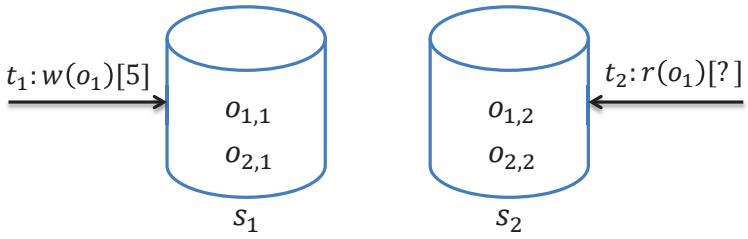


Figure 3.11: Behavior of transactions in a lazy replicated database system.

root. Then, child nodes are recursively added to each node until the depth l is reached [AE92]. In order to guarantee the intersection property, the read ($rq = \langle l_r, ch_r \rangle$) and write quorums ($wq = \langle l_u, ch_u \rangle$) must overlap. This means that the following conditions must hold: $l_r + l_u > h$ and $ch_r + ch_u > degree$. Moreover, two write quorums must overlap: $2 \cdot l_u > height$ and $2 \cdot ch_u > degree$.

The Log-Write Tree Quorum (LWTQ) is a special instance of the TQ with $rq = \langle 1, degree \rangle$ and $wq = \langle height, 1 \rangle$ [AE92]. Using such a strategy leads to reads accessing a single site (root) in a failure-free environment, and updates accessing a path down the tree. In the example depicted in Figure 3.10, a read quorum consists of the root node (s_1), whereas the write quorum may consist of s_1, s_2 and s_5, as it must access one node at each level. If a site that is not part of the write quorum receives an update transaction, then it can either forward it to the root for execution, or can access the read quorum for getting the most recent data, process the transaction and eagerly commit all sites consisting the write quorum. According to the analysis in [JPAK03], LWTQ has best scalability compared to other TQs.

Lazy Protocols

In lazy RPs update transactions commit only at local site. The updates to other sites are propagated *later*. The decoupling of the update propagation from the transaction execution, and response delivery to the client, has considerable performance advantages compared to eager approaches as there no network communication during the transaction execution. This effect can be considerable if the sites are distributed in a wide area network. Thus, the mapping of writes on logical objects to writes on physical copies is defined as follows:

$$w(o_i) \mapsto W.All(o_i)$$

This behavior of propagating updates after the response has been delivered to the client, together with the consistency demands of applications, have a considerable impact on the behavior of read actions as described in the Example 3.8.

Example 3.8 (Behavior of Reads in a Lazy Replicated DBS)

Let us consider a fully replicated DBS with two sites depicted in Figure 3.11. The update-transaction t_1, which updates o_1, is executed at s_1 and successfully commits at τ_a. The updates from s_1 to s_2 are propagated at $\tau_a + \delta$. This means that at any time point

τ with $\tau_a < \tau < \tau_a + \delta$ the sites s_1 and s_2 reflect different values of o_1. This period is also known as the *inconsistency window*. Let us assume that t_2 reads o_1 at s_2 during the inconsistency window. If the application is satisfied with an outdated value, then it is sufficient to read the value of o_1 at s_2, and return its value to the client. However, if the application requires access to the most recent version of o_1, then t_2 must access o_1 also at s_1 in order to satisfy the consistency requirements.

Based on Example 3.8 we can define the mapping of a read action as follows:

$$r(o_i) \mapsto R(o_i) \in \mathcal{P}(W.All(o_i))) \wedge |R(o_i)| \leq |S|$$

From the above we can conclude that there are three approaches to update propagation in lazy replicated DBSs:

Push approach. Each site is responsible to pro-actively propagate updates to other sites in the system. In the Example 3.8, s_1 is responsible to propagate the t_1's updates to s_2.

Pull approach. Sites request the missing updates from other sites. In the Example 3.8, s_2 is responsible to retrieve t_1's updates from s_1.

Combined approach. Is a combination of the push and the pull approach. For example, sites may periodically propagate their updates to other sites. However, if a transaction is executed by an outdated site, it might initiate refresh transactions to pull the updates from the most recent sites. In the Example 3.8, s_1 will periodically propagate updates to s_2. As t_2 is accessing data at s_2 during the inconsistency window, if it demands most recent data, then it will pull the updates from s_1.

The freshness guarantees provided to applications (transactions) are determined in the push approach by the propagation frequency. The greater its value, the smaller the inconsistency window. In the pull approach it is the mapping of the read actions that determines the freshness, whereas in the combined approach, the freshness is determined by both, the propagation frequency and the read behavior.

3.3.4 Consistency Models for Distributed Databases with Replication

As already defined, concurrency control and replica control determine the consistency guarantees provided by a distributed database with replication. In what follows we will summarize common consistency guarantees (models) for replicated databases [KA00a, FR10].

One-Copy Serializability (1SR)

1SR was originally defined in [BHG87]. Informally, it requires that the effects of an interleaved transaction execution on a replicated database is equivalent to a serial execution on a single-copy database. Strong 1SR imposes the real-time ordering of transactions, and is defined as follows [ZP08]:

Definition 3.33 (Strong Serializability)**.** *Schedule sch is strongly serializable iff there is some serial schedule sch_s such that (a) $sch \equiv sch_s$ and (b) for any two transactions t_y and t_z , if the commit of t_y precedes the submission of t_z in real time, then t_y commits before t_z is started in sch_s.*

Example 3.9 (Real Time Ordering of Transactions)

Let us assume two transactions, namely transaction t_1 that modifies the user password, and the login transaction t_2, that is initiated by the same user shortly after t_1. Although in real time t_2 has been started after t_1 the serialization order may correspond to t_2 t_1. In that case the user would not be able to login with the new password. Such a situation would be prohibited by Strong 1SR.

Although strong 1SR is the desirable consistency model, it is expensive and impossible to achieve in a distributed database. However, as depicted in Example 3.9, in some cases it is necessary to order certain transactions based on their real time. In [ZP08] the authors propose to enforce real time ordering only for transactions inside the same *session*. A session denotes an interactive information exchange between a client and a system. As described in [BDF+13], a session describes a context that should persist between transactions. A mapping of transactions to sessions can be done using *labels*. All transactions having the same label are assigned to the same session, i.e., context is shared between these transactions. The definition of strong session 1SR is as follows [ZP08]:

Definition 3.34 (Strong Session 1SR)**.** *Schedule sch is session consistent iff there is some serial schedule sch_s such that (a) $sch \equiv sch_s$ (i.e., sch is 1SR) and (b) for any two transactions t_y and t_z that belong to the same session, if the commit of t_y precedes the submission of t_z in real time, then t_y commits before t_z is started in sch_s.*

Snapshot Isolation

According to the original definition of SI (Section 3.1.4) [BBG+95], a transaction t_y with a real start time $rstart(t_y) > commit(t_z)$ is not guaranteed to see the t_y's updates. This is a consequence of the fact that the start time of t_y can be smaller than its real start time: $start(t_y) \leq rstart(t_y)$.

Such an inconsistency is called *transaction inversion* [DS06]. Again, if we consider Example 3.9 with the user submitting a transaction for updating the password, and subsequently another one for login, the user may not be able to login with the second transactions. Usual implementation in single-copy databases would choose the start time of a transaction by considering its real start time in order to avoid transaction inversion. In replicated databases the transaction for login may be run against a copy that does not yet reflect the updated password, which might be the case if lazy synchronization is used. Transaction inversion is prevented by strong SI [DS06][3]:

[3]Strong SI is called Conventional SI in [EZP05], Global Strong SI in [DS06] and Strict SI in [SMAdM08].

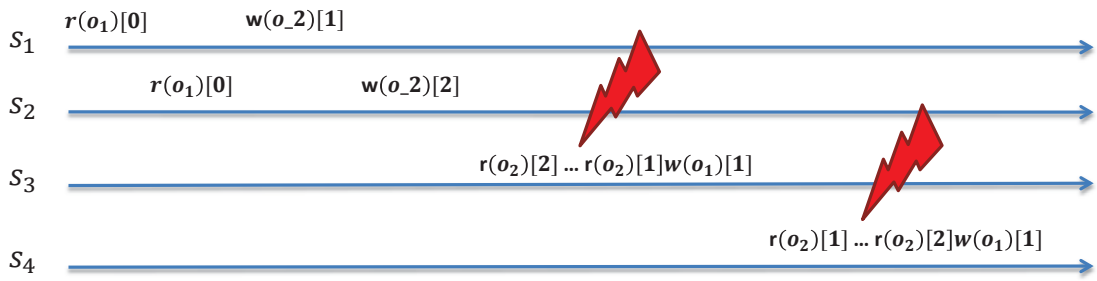


Figure 3.12: Example of causal schedule that is not serializable.

Definition 3.35 (Strong SI). *A schedule sch is strong SI iff it is SI (see Definition 3.19), and if, for every pair of committed transactions t_y and t_z in sch such that t_y's commit precedes the first action of t_z , $start(t_z) > commit(t_y)$.*

Similar to strong session 1SR, it is possible to enforce strong SI only inside a session [DS06].

Causal Consistency

Causal consistency models order transaction actions based on the `happens-before` relationship [Lam78]. If an action ac_y `happens-before` ac_z ($ac_y \rightarrow ac_z$), then ac_z is causally dependent on ac_y. Actions that are causally unrelated are called `concurrent` actions ($||$).

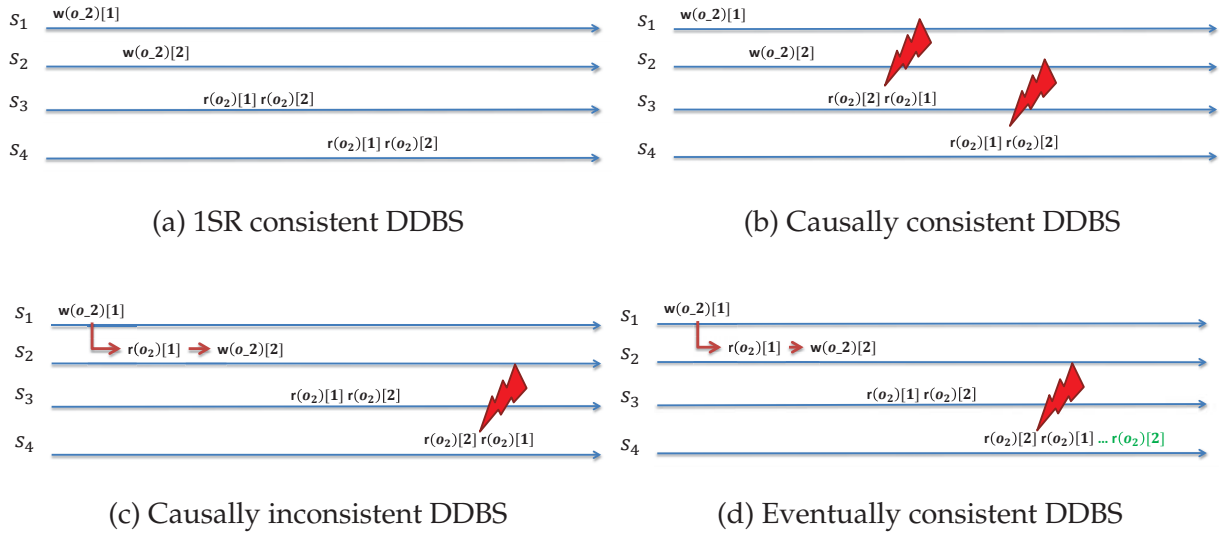
Example 3.10 (Ordering of Actions based on their Causal Dependency)

Let us assume three transactions t_1, t_2 and t_3, and two data objects o_1 and o_2 with initial values of *null*. t_1 executes a write operation on $w_1(o_1)[5]$, whereas t_2 reads ($r_2(o_1)$) and updates o_2 based on the read value for o_1 ($w_2(o_2)[10]$). $w_2(o_2)$ is causally dependent on $w_1(o_1)$, i.e., $w_1(o_1) \rightarrow w_2(o_2)$. The third transaction $t3$, that is started after t_1 and t_2 have successfully committed, reads the values of o_1 and o_2: $t_3 = r_3(o_1)\ r_3(o_2)$.

If $t_3 : r_3(o_1)[null]\ r_3(o_2)[10]$, then t_3 observes a causally incorrect ordering of operations. Causally correct orderings would be: $t_3 : r_3(o_1)[5]\ r_3(o_2)[null]$ and $t_3 : r_3(o_1)[5]\ r_3(o_2)[10]$.

Notice the difference compared to (conflict-) serializability: only writes with a read-from relationship are ordered, i.e., writes with a read in-between. Two transactions that write on the same object without a read in-between can be ordered differently at different replica sites. The example in Figure 3.12 depicts a schedule that is causally consistent, but not serializable as the write actions on o_2 are observed in different orders at s_3 and s_4.

Most existing causal correctness models capture only the *explicit* causality. *Potential* causality, in contrast to explicit causality, considers all actions that could have influence a certain action [Lam78, ANB+95]. For example, in social network a reply R to a user



(a) 1SR consistent DDBS

(b) Causally consistent DDBS

(c) Causally inconsistent DDBS

(d) Eventually consistent DDBS

Figure 3.13: (a) Depicts an 1SR DDBS. (b) Depicts a causally consistent DDBS. As writes of s_1 and s_2 are causally unrelated, s_3 and s_4 may observe the writes in different order. The schedule is thus not 1SR. (c) Depicts a causally inconsistent DDBS. Although the write at s_2 is causally dependent on the write at s_1, s_4 observes them in the opposite order. (d) Depicts an eventually consistent DDBS, as s_4 observes the writes of $_1$ and s_2 temporarily in the wrong order.

comment C defines an explicit causality $C \rightarrow R$. However, R might have been also influenced by a news in a newspaper, rumors, etc., and all this actions consist the set of actions that have potentially influenced R.

Eventual Consistency

EC is a weak consistency model that was made popular in the context of CAP [SK09]. In [BLFS12] the authors provide a formal definition of EC which is based on the *visibility* and *arbitration* order.

The visibility order defines the updates that are visible to a transaction, whereas the arbitration order the relative order of the updates. It is well known that eventual consistency can tolerate network partitions, as the arbitration order can remain partially determined for some time after transaction commit [BLFS12].

Two approaches are well-known for the determination of the update order, namely the Thomas' Write Rule (TWR) [Ber99] and commutative replicated data types [SPB+11]. By using the TWR, each transaction receives a timestamp, and each object stores the timestamp of the last update applied to it. An update operation on an object is only applied, if the timestamp of the update is greater than the timestamp of the last applied update. This rule enforces that the most recent updater wins. TWR is defined as follows [WV02]:

Definition 3.36 (Thoma's Write Rule)**.** *Given a transaction t_y with a timestamp $ts(t_y)$. For any object o with the timestamp $wts(o)$ modified by t_y: if $ts(t_y) > wts(o)$ then action $w_y(o)$ is processed, otherwise it is ignored.*

System Type	Description
AP	- Available - Tolerant to Network Partitions
CP	- Consistent - Tolerant to Network Partitions

Table 3.5: System types according to CAP

[SPB+11] defines many convergent and commutative data types, which allow arbitrary ordering of operations, at anytime and anywhere.

As depicted in Figure 3.13 eventual consistency is weaker than both the serializable consistency and causal consistency, as the ordering of conflicting updates may remain partial for some time.

3.4 Fundamental Trade-offs in the Management of Data in Distributed Database Systems

Distributed databases face the trade-offs captured by the CAP-theorem, which states that any distributed system can provide two of the three properties, namely consistency, availability and partition tolerance. The CAP-theorem was turned into a formal definition and proved in [GL02]. What does it mean to sacrifice consistency or availability? Before answering this question, it is necessary to define what a network partition means.

Definition 3.37 (Network Partition [GL02])**.** *In case of a network partition all messages sent from a node [site] in one component of the partition to nodes [sites] in another component are lost.*

Network partitions are real and prevalent and cannot be negotiated when designing a distributed database system, as that would mean to run on an absolutely reliable network [BFG+13, BDF+13, Ham10]. As a consequence, DDBS can choose between availability and consistency. This leads to two basic types of DDBS, namely DDBS that provide high availability, and sacrifice consistency (AP), and those that sacrifice availability for consistency (CP) (Table 3.5).

AP DDBSs

Definition 3.38 (High Availability [GL02])**.** *A system is highly available if a client that contacts a running site will eventually receive a response from that site, even in the presence of arbitrary, indefinitely long network partitions.*

From the Definition 3.38 it can be concluded that, if a site requires any sort of coordination with any of the sites in the system, then high-availability cannot be guaranteed. Let us consider the banking example depicted in Figure 3.6. If the connection from the ATM to the back-end fails, then striving towards high availability would mean that the

client should be able to withdraw money at the cost of a possible account overdraw (constraint violation). Surprisingly, in contrast to common believe, this is how banking applications in reality behave [Hof, Hal10, BG13]. Banks employ other mechanisms to compensate for negative account balances, i.e., they do not prevent inconsistencies, but compensate them.

In [BFG+13, BDF+13] the authors provide further levels of availability which are defined as follows:

Definition 3.39 (Sticky Availability)**.** *A system is sticky available if, whenever a client's transaction is executed against a replica site that reflects all of its prior operations, the client will eventually receive a response, even in presence of indefinitely long partitions.*

Definition 3.40 (Transactional Availability)**.** *A system provides transactional availability if a transaction can contact at least one replica site for every object it accesses, and that the transaction will eventually commit or internally abort. Sticky transactional availability is provided if given sticky availability, a transaction will eventually commit or internally abort.*

One of the important question is the compatibility of the consistency models with the different availability levels. As concluded in [BFG+13, BDF+13], causal consistency is the strongest model achievable in case of network partitions compatible with the sticky availability level. All models that require coordination, such as those that avoid lost-update, constraint violations, recency violations, etc., are not achievable in AP systems.

CP DDBSs

CP databases in presence of network partitions would sacrifice availability for consistency. In the banking scenario in Figure 3.6, during a failed communication between the ATM and the back-end, the user would not be able to withdraw money in order to avoid possible constraint violations. This can happen as two clients can simultaneously withdraw money from the same account. Notice that even with one client attached to an account it might be that an internal bank process may regularly charge the account for the provided bank services, and if no communication is possible from the ATM to the back-end, then the balance may go into negative.

PACELC Trade-Offs

In [Aba12] the authors extend CAP to PACELC, which also addresses the consistency-latency trade-off. While CAP is about failures, there are also consistency-latency trade-offs during normal operations (*E*lse *L*atency *C*onsistency). The stronger the consistency model, the higher is the performance penalty. The intuition is that for strong consistency models, such as 1SR, the DDBS has to invest more resources in coordination compared to weaker models such as eventual consistency.

The consistency/performance trade-off is not only relevant for distributed databases, but has been well-known also in single-copy databases. This has been one of the main reasons for providing a range of isolation levels. This trade-off however has become even more tangible with the deployment of large scale applications

in the Cloud, that naturally distribute and replicate data across different data centers for scalability and availability. A coordination between sites in such a deployment would require several rounds of network communication, which, depending on the location of the sites, may become a major performance bottleneck [BFG+13]. Usual approaches tackle with the coordination overhead by relaxing consistency [PST+97, HP94, KS91, BO91].

Consistency-Programmability Trade-Off

Weak consistency is difficult to reason about, and it additionally imposes challenges to application developers as they need to treat different possible failure cases [BGHS13, BFG+12, ADE11, Ham10].

In reaction to CAP/PACELC trade-offs, many NoSQL databases have been developed that provide only eventual consistency guarantees. However, the popularity of NoSQL databases has attracted also applications that demand stronger consistency guarantees than eventual consistency. Such applications need to compensate missing consistency guarantees at application level, which increases their development cost and complexity. Recent versions of NoSQL databases provide tunable consistency, which allows clients to specify the desired consistency guarantees at request level[4]. For example, SimpleDB [sima] has introduced the `consistent read` feature. The default behavior of a read request is eventually consistency, that might return a stale value for an object. However, via a flag the user can request strongly consistent read, that returns a result reflecting the values of all successful writes prior to the read .

Consistency-Cost Trade-Off

The higher coordination overhead required for ensuring strong consistency inherently leads to more actions being executed, and higher resource consumption (Central Processing Unit (CPU), Random Access Memory (RAM), etc.). In the Cloud there is a price tag associated with each action and resource consumption. Thus, the stronger the consistency model the higher the generated monetary costs [KHAK09, FS12, IH12].

Relaxed consistency models generate less cost. However, if we consider the online shop application depicted in Figure 2.1 that uses SimpleDB is its database, books oversells might occur (if eventually consistent read is used). This in turn, may lead to an overhead for the customer care and administrative (monetary) cost, as a consequence of the inconsistencies. Clearly, the full impact of an inconsistency is difficult to be assessed as it includes aspects such as the loss of customers.

In summary, strong consistency generates high consistency costs. Weak consistency generates less consistency costs, but may generate many expensive inconsistencies.

[4]The development of data consistency has roughly undergone following phases, namely the development of transactions as an abstraction, the development of NoSQL databases that do not provide transactional guarantees, the development of databases with limited transactional support, such as ElasTras [DEA09], G-Store [DAE10], Relational Cloud [CJP+11], introduction of adjustable consistency in existing NoSQL databases, and the return of transactions with MegaStore [BB+11], Spanner [CD+13], and others.

BASE, CALM and ACID 2.0

As a consequence of the CAP/PACELC trade-offs described above, NoSQL databases usually provide only BASE (*B*asically *A*vailable, *S*oft State, *E*ventually Consistent) guarantees, i.e., relax consistency for availability, as opposed to traditional ACID guarantees.

However, BASE databases increase the complexity of application development if strong consistency is required (consistency-programmability trade-off). Recently, the CALM (*C*onsistency *a*s *L*ogical *M*onotonicity) has been developed, which captures programs that are consistent by design and that can be safely run on an eventually consistent database [ACHM11]. The idea of CALM is to provide a theoretical ground for checking which programs are safe to run with eventual consistency, and which not. ACID 2.0 (*A*ssociativity, *C*ommutativity, *I*dempotence and *D*istributed) describes a set of design patterns that allow the implementation of safe application on top of eventually consistent databases. CRDTs (*C*ommutative, *R*eplicated *D*ata *T*ypes) define a set of standard data types that are provably eventually consistent. If correctly used, applications are guaranteed to never produce any inconsistency, i.e., safety violation.

4

Cost- and Workload-Driven Data Management

IN THIS CHAPTER, we lay out the ground for the development of our CCQ protocols by summarizing common concepts and models. We will start this chapter with an introduction to the *configuration* concept for capturing the CCQ behavior, which is driven by a cost model and the application workload. Moreover, we formalize the notion of *cost* and that of the *workload*, and provide a detailed description of the workload prediction based on Exponential Moving Average (EMA). The prediction model is common to all CCQ protocols and provides raw data, out of which further information is derived by each CCQ protocol based on the concrete cost model.

4.1 Introduction

The CCQ protocols are characterized by their adaptive *behavior* driven by the application and infrastructure cost, as well as by the application workload. The behavior of a CCQ protocol is defined by its *configuration* chosen from the set of possible configurations that defines the *configuration space*. Each configuration from the configuration space is able to satisfy the application requirements at different *degrees* and *costs*.

We distinguish between *operational costs* and *penalty costs*. Operational costs capture the overhead generated for enforcing certain guarantees, such as resource consumption, necessary processing and storage activities to be executed, impact on performance and others. Penalty costs are application specific costs that incur as the result of a requirement violation, i.e., of a satisfiability lower than the desired degree. The goal of each CCQ protocol is to choose that configuration from its configuration space that incurs the overall minimal costs. A workload shift may necessitate a reconfiguration, i.e., a transition from one configuration to another. A transition to another configuration may lead to cost reduction, i.e., may generate a *gain* for application. However, it may also generate *transition costs*, which are defined in terms of reconfiguration activities. The CCQ protocols will consider all these different costs when choosing the optimal configuration.

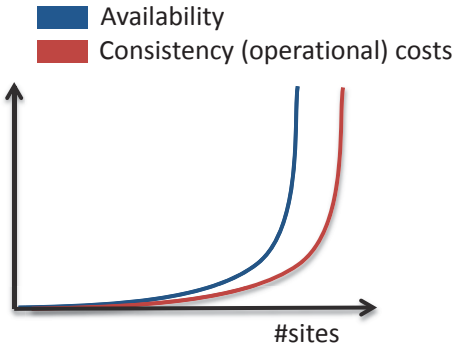


Figure 4.1: Availability and consistency costs with increasing number of sites.

In what follows, we will provide concrete examples that depict the different possible aspects that influence the CCQ configurations. We assume that the application requirements have been validated and hence are achievable by the underlying system as described in Section 2.2.

Example 4.1 (Data Consistency)

Let us assume that the provider of the online application described in Figure 2.1, has defined that all data should be highly available (e.g., $availability \geq 0.99999$), and that the satisfiability of this requirement must not be violated, as its violation would incur extremely high (possibly infinite) penalty costs. The penalty costs may be a consequence of customer loss, loss of image, and others, and are application specific. In order to satisfy the availability requirement, the underlying DBS must replicate the data across different geographical locations. The higher the availability degree, the more resources (sites) need to be deployed.

Satisfying the desired availability avoids/reduces penalty costs, but may impact the costs of other data management properties, such as the costs of data consistency. The more sites available, the higher the costs for strong consistency, as more messages need to be exchanged via the network (Figure 4.1). The consistency costs can be reduced by relaxing consistency. However, weak consistency may generate penalty costs. Both consistency and inconsistency costs are highly dependent on the application workload and number of available sites, which are dynamic and may change over time. Therefore, the DBS must choose the optimal consistency by considering all parameters, and must continuously assess its decision.

The operational costs can also be measured in terms of performance gain or loss. In this concrete example, the number of sites is determined by the availability requirement. Thus, these resources are there and the application provider is charged for them. Increasing the performance, by for exampling relaxing consistency, decreases the operational costs, as it generates a utility to the application in terms of, for example, user satisfiability. Clearly, this is not easy quantifiable.

In summary, the configuration of the DBS, is determined by the availability requirement, which is mapped to a number of replica sites, and the concrete consistency level. As application workload is dynamic the configuration choice needs to be continuously

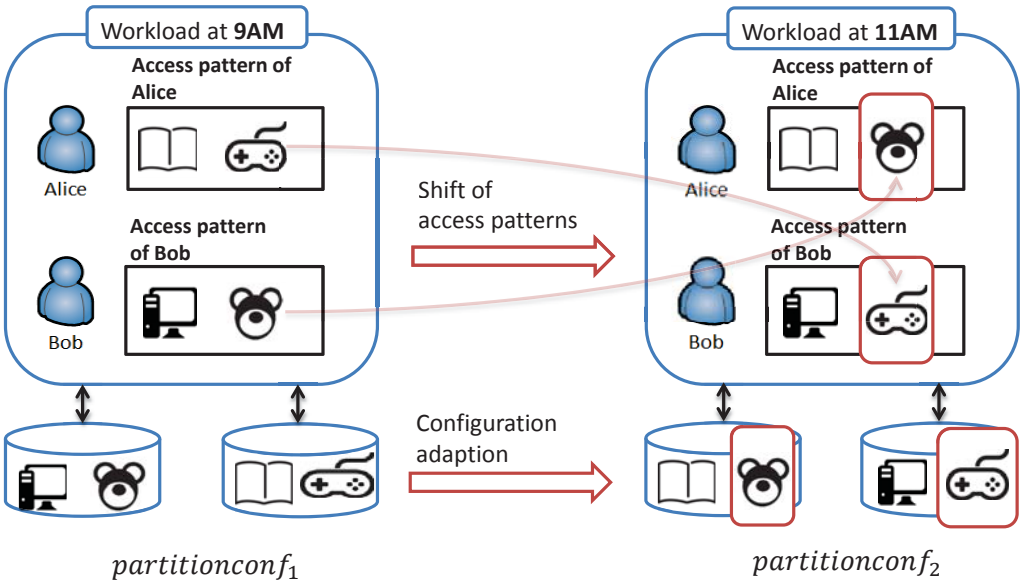


Figure 4.2: Adaptation of the partitions in case of a workload shift. Alice and Bob are end-users of the online shop. The different possible roles in context of the online shop are defined in Section 2 (see Figure 2.1).

monitored and, if necessary, adapted. While a new configuration may generate a gain in terms of cost reduction, the reconfiguration may also generate an overhead, which outweighs the gain. For example, an adaptation of consistency from EC to 1SR may generate transition costs as a consequence of site reconciliation (Section 3.3.4).

Example 4.2 (Data Replication)

In the previous example we assumed that the application provider left the decision on the optimal consistency to the underlying DBS. Let us now assume that the application provider, in addition to the availability requirement, specifies that 1SR must also not be violated by defining infinite penalty costs for data inconsistency. This additional requirement does not mean that there is no room for optimization. The DBS can choose the most suitable 1SR protocol, or the most suitable RP by considering the application workload. If the workload is read-only, then ROWAA is the best choice. However, in case of an update-heavy workload a quorum protocol may be the optimal choice with regards to performance and costs. Furthermore, it is also possible to adapt the CCP by, for example, exchanging the pessimistic protocol with an optimistic one or vice-versa. In this example, the concrete protocol used to implement 1SR is what determines the configuration at runtime.

Example 4.3 (Data Partitioning)

If the following example we assume that no availability requirements whatsoever are

specified by the application provider. This means that the data do not need to be replicated. However, it defines that 1SR must be guaranteed. The main overhead of 1SR incurs from the network communication in case of distributed update transactions, that generates a performance penalty, and incurs high monetary cost. As no availability requirements are defined, data can be partitioned in order to minimize or completely avoid distributed transactions. Different partition configurations are more or less suitable for a given workload. The suitability is defined in terms of distributed transactions, the less distributed transactions a configuration generates, the more suitable it is for that workload. As the workload is dynamic, the configuration needs to be continuously updated in order to reflect the changes in the workload. For example, given the scenario depicted in Figure 4.2, $partitionconf_1$ may be the most suitable configuration for the workload at $9am$. However, if later, at $11AM$ the workload changes, $partitionconf_1$ becomes unsuitable. In this example, the active partitions determine the configuration.

A configuration adaptation may necessitate a transfer of a large amount of data between the sites. This means that a decision to reconfigure the partitions should only be taken if the new configuration provides a significant gain.

Example 4.4 (Integrated Data Management)

During the lifetime of the online shop depicted in Figure 4.2, the provider may observe that certain partitions are more critical in terms of availability than others. As a consequence, these important partitions need to be replicated for high availability. For the replicated partitions the DBS needs to tackle with the same challenges described in the Examples 4.1 and 4.2. Thus, as depicted in Figurer 4.3, it can either relax the consistency to reduce the overhead for update transactions, or enforce strong consistency (e.g., if applications demand 1SR), and choose the most appropriate RP based on the application workload. As described in Example 4.1, the choice of the consistency model can also be influenced based on, for example, penalty costs.

This example also depicts the dependencies between the data management properties and their impact on each-others behavior. The decision to replicate a certain partition ($partition_1$) leads to the necessity of re-evaluating the decision on the consistency model, and the protocol implementing the model.

Based on the examples above we can conclude the following:

1. Applications have different requirements towards the underlying DBS. A violation of one or more requirements may incur application specific *penalty costs*.
2. Each specific *configuration* generates *operational costs* that can be defined in terms of activities to be executed and resources to be consumed for satisfying the application requirements given a specific workload.
3. A *reconfiguration* may decrease the penalty and operational costs. However, it might generate *transition costs*, which must be considered when determining the most suitable configuration.

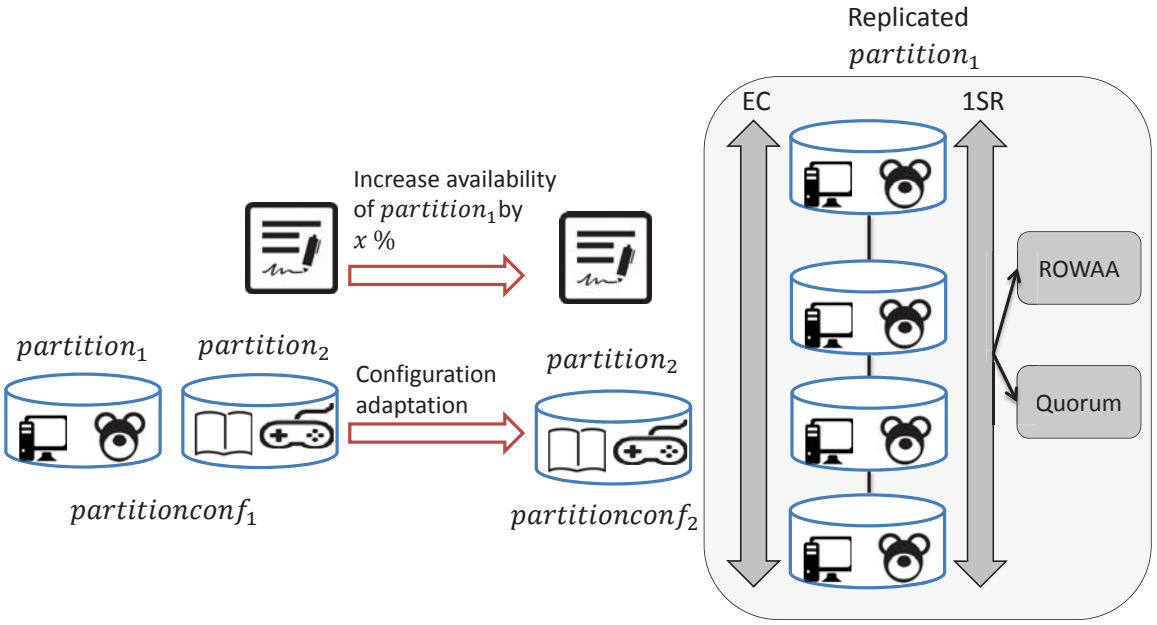


Figure 4.3: Integrated data management that jointly considers data consistency, partitioning and replication.

4. The previous examples consider only a subset of SLOs and data management properties influenced by the SLOs[1]. As described in Section 2.3, an integrated data management approach should additionally consider conflicts between SLOs. Moreover, as the different data management properties are not orthogonal it is crucial to consider their dependencies (see Example 4.4).

4.1.1 Monetary Cost

In the Cloud, based on its pay-as-you-go cost model, each and every activity executed and a resource consumed has a price tag associated with it. This, in contrast to traditional infrastructures, has made the monetary cost more transparent, and has led to the cost becoming a first-class citizen in the development of applications in general, and data management in particular [FK09].

The cost models differ between Cloud providers and types of services. AWS, for example, charges its customer for the IaaS machine instances on hourly bases, and the concrete price depends on the machine configuration (e.g., number of CPUs, RAM, etc.). In addition to hardware resources, providers charge their customer also for the different actions they execute, down to the level of single messages. If we consider AWS S3[2],

[1]The term 'data management property' is in purpose abstract as our analysis on the granularity is not final. For example, we can consider data management properties at the DDBS,DBS or protocol level. A protocol can be divided into further subprotocols that consider data management properties at the subprotocol level. For example, the query execution engine of a DBS can divided in a query planer and query executor subprotocols, which at the same time denote specific data management properties.

[2]https://aws.amazon.com/de/s3/pricing/.

Request	Price
PUT, COPY, POST, or LIST	\$0.005 per 1,000 requests.
GET	\$0.004 per 10,000 requests.
DELETE	Free.
Storage	**Pricing.**
First 1 TB/month	\$0.0300 per GB.
Next 49 TB/month	\$0.0295 per GB.
Next 450 TB/month	\$0.0290 per GB.
Next 500 TB/month	\$0.0285 per GB.
Next 4000 TB/month	\$0.0280 per GB.
Next 5000 TB/month	\$0.0275 per GB.
Data Transfer OUT from S3 To Internet	**Pricing**
First 1GB/month	\$0.000 per GB.
Up to 10 TB/month	\$0.090 per GB.
Next 40 TB/month	\$0.085 per GB.
Next 100 TB/month	\$0.070 per GB.
Next 350 TB/month	\$0.050 per GB.

Table 4.1: Amazon S3 pricing as of 2^{nd} February 2016 for the US East Region.

clients pay a fee for the data storage and data transfer, but also a varying fee for each Create, Read, Update and Delete (CRUD) action (see an example in Table 4.1).

With such fine-grained cost models, it is easily possible to incorporate the monetary cost into the optimization model of data management protocols. In addition to providing the desired data guarantees, we now need also to care about the costs of the guarantees and explicitly strive towards optimization of the monetary costs. Clearly, this increases the complexity of DBS. However, it fits well into the requirements of cost-aware clients, which in addition to traditional metrics (response time, throughput) require the cost in \$ to be reported [FK09].

4.1.2 Configuration and Cost Model

A *system* is defined by the set of protocols that implement a certain (set) of data management properties, together with the set of possible protocol *configurations*. The available protocol configurations define the *protocol capabilities*, and the set of all available protocols together with their configuration space define the *system capabilities* (Figure 4.4).

The set of all possible configurations defines the *configuration space* $\mathbb{C}$ (Table 4.2):

$$\mathbb{C} = \{conf_1, conf_2, conf_3, \ldots\}$$

Each protocol has its configuration space (Figure 4.4), which can be depicted as a fully connected graph $CG = (N, E)$. The nodes of the graph denote the configurations, and the edges denote *transitions* between the configurations (Figure 4.5).

Meta-protocols can be built on top of existing protocols (see `CostBased-P` in Figure 4.4). The configuration space of a meta-protocol includes all configurations of the

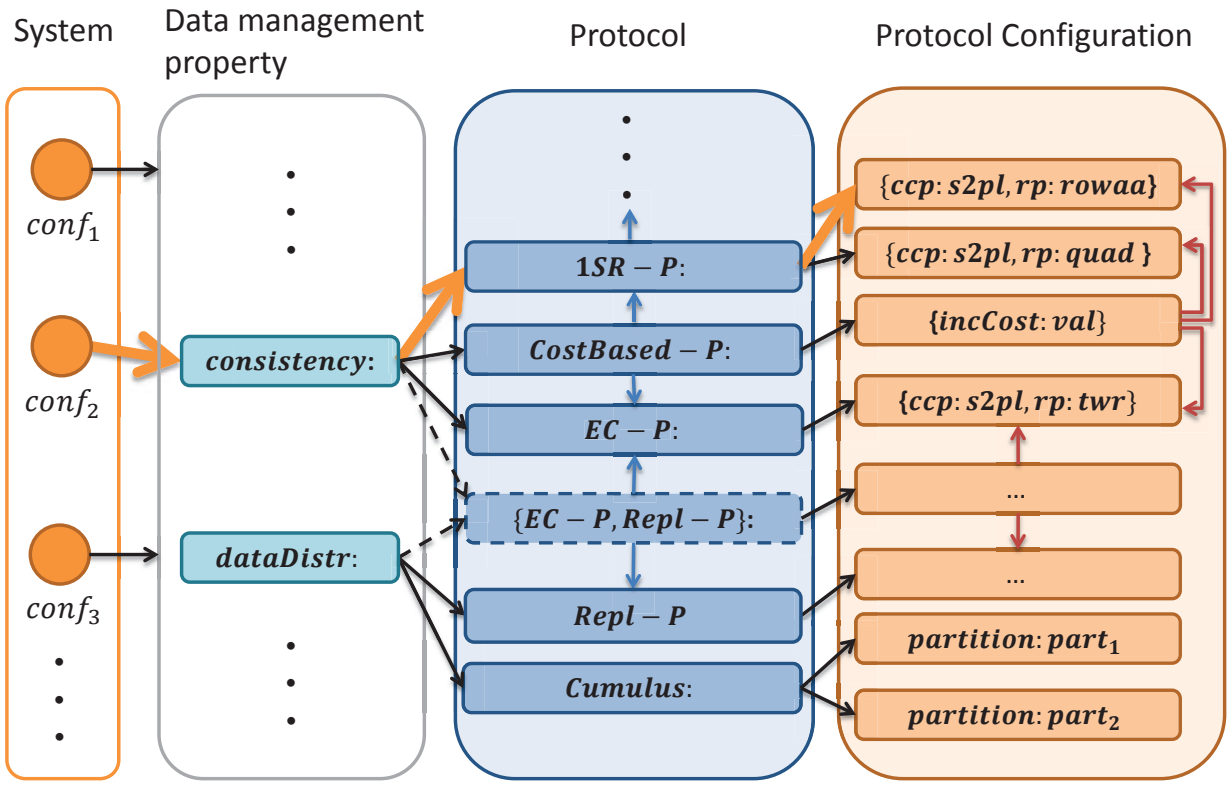


Figure 4.4: Relationship between system, data management property, protocol and protocol configuration.

protocols it consists of. Moreover, multiple data management properties can be implemented by a single *integrated* protocol (dashed box in Figure 4.4). The integrated protocols needs to also consider dependencies between the data management properties, and will choose the optimal configuration based on a holistic model that jointly considers the different data management properties (see Example 4.4).

Both the application workload and the requirements may change dynamically at runtime. We assume time to be divided in periods $p_1, p_2, ...$, and will denote with $\mathcal{R}^{p_i}$ the set of requirements valid during p_i, and $(wload^{p_i})$ the application workload during that period.

Each configuration satisfies the requirements to a certain degree:

$$\forall conf \in \mathbb{C} : satDegree(\mathcal{R}^{p_i}, wload^{p_i}, conf) \in [0,1]$$

A subset $\mathbb{C}^{sat}$ of configurations from $\mathbb{C}$ may exist that provides the desired satisfiability to the application requirements give a certain tolerance threshold. We assume that all requirements can be satisfied to a certain degree, as they have already been validated based on the interaction cycle between the client and the DBS provider (Section 2.3).

$$\forall conf \in \mathbb{C}^{sat} : satDegree(\mathcal{R}^{p_i}, wload^{p_i}, conf) \geq 1 - threshold$$

The violation of a set of requirements from $\mathcal{R}$ by a specific configuration $conf$ may generate application specific penalty costs:

Symbol	Description
$\mathbb{C}$	Denotes the configuration space.
$conf$	Denotes a configuration: $conf \in \mathbb{C}$.
$wload$	Denotes the application workload.
$\mathcal{R}$	Denotes the application requirements.
$satDegree(\mathcal{R}, wload, conf)$	Defines the satisfiability of $\mathcal{R}$ by the configuration $conf$ given the workload $wload$.
$pCost(\mathcal{R}, wload, conf)$	Denotes the penalty costs generated by the configuration $conf$ for the violation of $\mathcal{R}$ given the workload $wload$.
$opCost(\mathcal{R}, wload, conf)$	Denotes the operational costs generated by the configuration $conf$ for enforcing $\mathcal{R}$ given the workload $wload$.
$resourceCost(\mathcal{R}, wload, conf)$	Denotes the costs incurring from the resource consumption.
$activityCost(\mathcal{R}, wload, conf)$	Denotes the costs incurring from the necessary activities to be executed for enforcing $\mathcal{R}$ given the workload $wload$.
$utility(\mathcal{R}, wload, conf)$	Denotes the generated utility by the configuration $conf$ given the workload $wload$.
$totalCost(\mathcal{R}, wload, conf)$	Denotes the total costs generated by the configuration $conf$ for enforcing $\mathcal{R}$ given the workload $wload$.
$\widehat{gain}(\mathcal{R}^{p_i}, \widehat{wload}^{p_i}, conf_{current}^{p_{i-1}}, conf_{new}^{p_i})$	Denotes the expected gain of switching from the currently active configuration $conf_{current}^{p_{i-1}}$ to the new configuration $conf_{new}^{p_i}$), given the application requirements $\mathcal{R}^{p_i}$ and the expected workload $\widehat{wload}^{p_i}$.
$\widehat{benefit}(\mathcal{R}^{p_i}, wload^{p_{i-1}}, \widehat{wload}^{p_i}, conf_{current}^{p_{i-1}}, conf_{new}^{p_i})$	Denotes the expected benefit of switching from the currently active configuration $conf_{current}^{p_{i-1}}$ to the new configuration $conf_{new}^{p_i}$), given the application requirements $\mathcal{R}^{p_i}$ and the expected workload $\widehat{wload}^{p_i}$.
$\widehat{Y^{p_i}}$	Denotes the predicted time series value for period p_i.
$Y^{p_{i-1}}$	Denotes the time series value at period p_{i-1}.
θ	Denotes the autoregressive coefficient.
ϵ	Denotes the noise in the time series data.
α	Denotes order of the moving average.
$\mathcal{AP}$	Defines the set of access patterns.
ap	Defines an access pattern: $ap \in \mathcal{AP}$.
$occ(ap)$	Defines the number of occurrences (frequency) of ap.

Table 4.2: Configuration and cost model: symbols and notations.

$$pCost(\mathcal{R}^{p_i}, wload^{p_i}, conf) \begin{cases} = 0 & : satDegree(\mathcal{R}^{p_i}, wload^{p_i}, conf) \geq 1 - threshold \\ > 0 & : satDegree(\mathcal{R}^{p_i}, wload^{p_i}, conf) < 1 - threshold \end{cases} \quad (4.1)$$

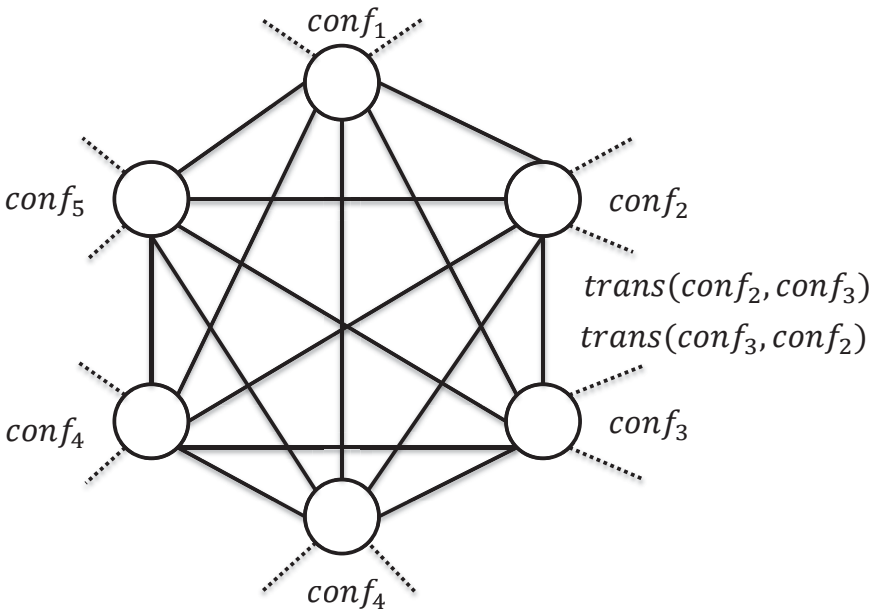


Figure 4.5: Configuration graph with the nodes depicting configurations and edges transitions between configurations.

For example, a violated upper bound of a response time requirement may lead to less sales[3]. The penalty costs are dependent on the unsatisfiability degree, and are application specific. Moreover, as the requirements are not orthogonal, the violation of one requirement may lead to an increase in satisfiability of another, or strengthen the violation of one or more other requirements.

Each configuration $conf$ incurs also operational costs that are defined as follows:

$$\begin{aligned} opCost(\mathcal{R}^{p_i}, wload^{p_i}, conf) = {} & resourceCost(\mathcal{R}^{p_i}, wload^{p_i}, conf) \\ & + activityCost(\mathcal{R}^{p_i}, wload^{p_i}, conf) \\ & - utility(\mathcal{R}^{p_i}, wload^{p_i}, conf) \end{aligned} \tag{4.2}$$

In Equation (4.2), *resourceCost* defines the costs that incur from the resource consumption (e.g., the number of sites), *activityCost* define the costs that incur from the necessary activities to be executed for satisfying the application requirements given $wload^{p_i}$, and *utility* defines the generated utility by the configuration. The utility is defined in terms of the gain generated for the satisfiability of an implicit requirement. For example, decreasing latency increases user satisfiability. It can be used to reward configurations that optimize certain implicit requirements, such as reduction of response time, and can be helpful to steer the configuration choice if more than one configurations are available that generate the same penalty, resource and activity costs.

Similar to the penalty costs, the satisfiability of one requirement may increase or decrease the operational costs for other requirements. For example, the higher the availability requirement the more resources are necessary, which also has an impact on cost

[3]According to Amazon, a slowdown of just one second per page request could cost them about $1.6 billion each year [Eat]. 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 [Eat].

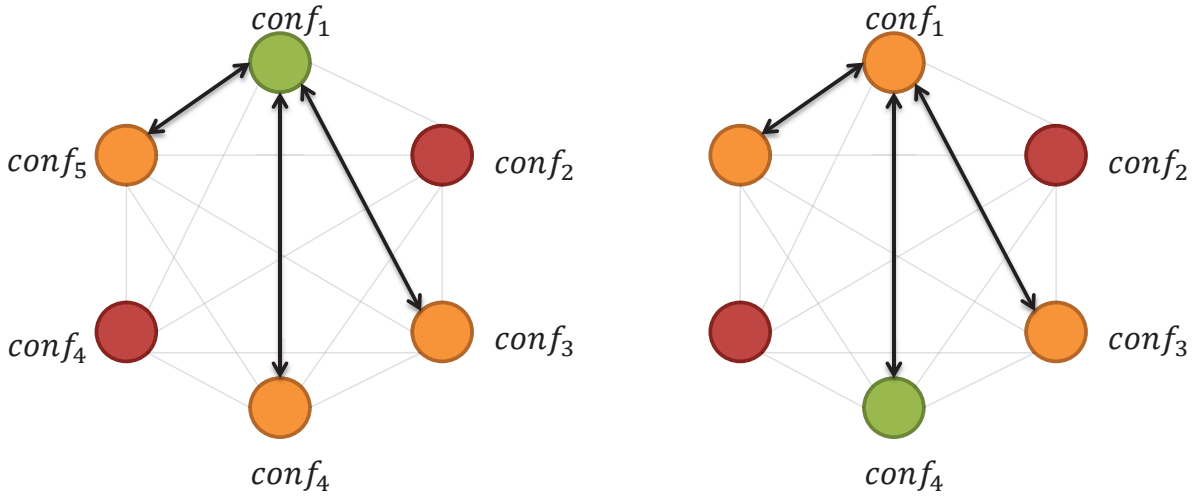


Figure 4.6: The green node denotes the currently active configuration; orange nodes denote configurations of interest. Red nodes denote configurations of no interest. Transitions of interest are depicted by dark arrows.

for guaranteeing 1SR consistency. 1SR operational costs may be measured in terms of number of messages to be exchanged over the network.

The total costs generated by a configuration $conf$ is defined as follows:

$$totalCost(\mathcal{R}^{p_i}, wload^{p_i}, conf) = pCost(\mathcal{R}^{p_i}, wload^{p_i}, conf) + opCost(\mathcal{R}^{p_i}, wload^{p_i}, conf) \tag{4.3}$$

Both application requirements and workload may change anytime. Moreover, also the infrastructure of the Cloud provider may change together with the price model. Any of these changes may necessitate a configuration adaptation, i.e., a transition to another configuration. Let $\mathcal{R}^{p_i}$ denote the application requirements that apply for p_i, $\widehat{wload}^{p_i}$ the predicted workload for p^i (we will use $\widehat{param}$ to denote the expected value of a parameter), and $conf^{p_{i-1}}_{current}$ the currently (in p_{i-1}) active configuration. Then, the expected gain in adapting the configuration to $conf^{p_i}_{new}$ is defined as follows:

$$\begin{aligned}\widehat{gain}(\mathcal{R}^{p_i}, \widehat{wload}^{p_i}, conf^{p_{i-1}}_{current}, conf^{p_i}_{new}) = \widehat{totalCost}(\mathcal{R}^{p_i}, \widehat{wload}^{p_i}, conf^{p_{i-1}}_{current}) \\ - \widehat{totalCost}(\mathcal{R}^{p_i}, \widehat{wload}^{p_i}, conf^{p_i}_{new})\end{aligned} \tag{4.4}$$

A reconfiguration may generate transition costs, which must also be considered. Hence, a transition to a new configuration generates a benefit if the cost reduction outweighs the transition costs. The expected benefit is defined as follows:

$$\begin{aligned}\widehat{benefit}(\mathcal{R}^{p_i}, wload^{p_{i-1}}, \widehat{wload}^{p_i}, conf^{p_{i-1}}_{current}, conf^{p_i}_{new}) = \\ \widehat{gain}(\mathcal{R}^{p_i}, \widehat{wload}^{p_i}, conf^{p_{i-1}}_{current}, conf^{p_i}_{new}) \\ - tCost(wload^{p_{i-1}}, conf^{p_{i-1}}_{current}, conf^{p_i}_{new})\end{aligned} \tag{4.5}$$

The goal of an adaptive DBS is to choose that configuration that maximizes the expected benefit:

$$\max_{conf \in \mathbb{C}} \widehat{benefit}(\mathcal{R}^{p_i}, wload^{p_{i-1}}, \widehat{wload}^{p_i}, conf^{p_{i-1}}_{current}, conf^{p_i}_{new}) \tag{4.6}$$

Equation (4.6) considers only the client's requirements towards the (Cloud) DBS provider. However, based on the economy of scale, the Cloud provider has an incentive in serving as many clients as possible, which may have conflicting requirements (Section 2). The provider should have the flexibility to optimize globally by finding the right balance between satisfying the requirements of all clients, and the resource usage. In certain cases, from the provider point of view, it might be more beneficial to violate the requirements and pay a penalty, than satisfy them.

The size of the configuration graph that considers all possible configurations would become extremely large. In reality, only a graph of a limited size is considered, which is defined by the concrete DBS deployment. Moreover, inside that graph, only a subset of configurations is of interest (Figure 4.6). For example, although the deployment may contain the capability of archiving the data, that functionality may not be of interest for the application. The configurations of interest, i.e., the subgraph, is defined by the application requirements.

4.2 Workload Monitoring and Prediction

The workload is one of the crucial components of our adaptive CCQ protocols. While many parameters, such as the penalty costs, resource costs (see Section 4.1.1), are predefined, the application workload is dynamic, and it determines the degree of requirement violation and the degree of resource usage.

In what follows, we will formalize the workload concept at the degree that is sufficient for the CCQ protocols.

Definition 4.1 (Workload)**.** *A workload is a set of tuples consisting of access patterns and their occurrence (frequency) (Table 4.2):*

$$\begin{aligned} wload^{p_i} &= \mathcal{AP} \times \mathbb{N}^+ \\ wload^{p_i} &= \{(ap^1, occ^{p_i}(ap^1)), \ldots, (ap^n, occ^{p_i}(ap^n))\} \\ ap &\in \mathcal{AP} \\ occ &: ap \rightarrow \mathbb{N}^+ \end{aligned} \tag{4.7}$$

$\mathcal{AP}$ defines the set of all possible access patterns: $\mathcal{AP} = \{ap^1, ap^2, \ldots\}$, and occ the number of occurrences of an access pattern in the $wload$ at time period p_i. An access pattern ap^q is a set of actions of finite size: $ap^q = \{ac^{q,1}, ac^{q,2}, \ldots, ac^{q,n}\}$ (see Section 3.1), and defines a transaction *template* (cf. class in object oriented programming). Multiple transactions can be instantiated from the same access pattern.

Actions are operations acting on a specific logical object (Section 3.1). Thus, two actions ac^k and ac^m are equal if the following condition holds:

$$ac^k = ac^m \iff ac^k.op = ac^m.op \wedge ac^k.o = ac^m.o \tag{4.8}$$

Two access patterns ap^q and ap^u are equal if they contain the same set of actions (Example 4.5):

$$ap^q = ap^u \iff (\forall ac\,[ac \in ap^q \iff ac \in ap^u]) \tag{4.9}$$

Example 4.5 (Equal and Distinct Access Patterns)

$$\begin{aligned}
ap^1 &= \{r^{1,1}(o_3), r^{1,2}(o_5), r^{1,3}(o_7)\}\\
ap^2 &= \{r^{2,1}(o_3), r^{2,2}(o_5), r^{2,3}(o_7)\}\\
ap^3 &= \{r^{3,1}(o_3), w^{3,2}(o_5), r^{3,3}(o_7)\}\\
ap^4 &= \{r^{4,1}(o_3)\}
\end{aligned}$$

$$\begin{aligned}
&ap^1 = ap^2\\
&ap^1 \neq ap^3 \wedge ap^2 \neq ap^3\\
&ap^1 \neq ap^4 \wedge ap^2 \neq ap^4 \wedge ap^3 \neq ap^4
\end{aligned}$$

Notice that the equality of access patterns is defined in terms of a set equality. Hence, it does not require that actions appear in the same order, and this is sufficient for the CCQ protocols, as their models do not consider the ordering of actions. In certain cases, however, it might be necessary to consider also the ordering of actions in an access pattern. In that case, access patterns are defined as sequences instead of sets. As access patterns define templates, the equality of two access patterns does not imply their membership to the same transactions.

The workload size is defined as follows:

$$|wload| = \sum_{ap \in wload} occ(ap) \tag{4.10}$$

Two workloads $wload_1$ and $wload_2$ are equal if they contain the same access patterns in the same frequency:

$$wload_1 = wload_2 \Leftrightarrow (\forall ap[ap \in wload_1 \Leftrightarrow ap \in wload_2] \wedge occ(ap, wload_1) = occ(ap, wload_2)) \tag{4.11}$$

4.2.1 Workload Prediction

The workload is dynamic and may change during the lifetime of an application. The CCQ protocols follow the approach of deploying the most cost-optimal configuration

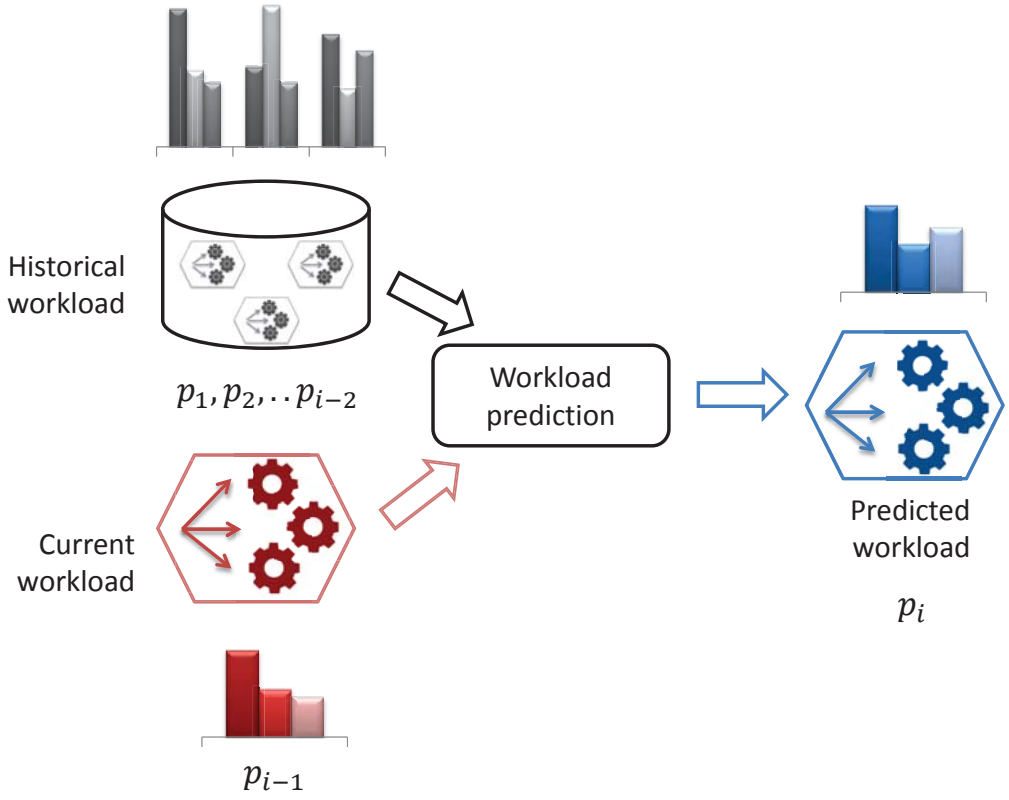


Figure 4.7: Workload prediction for the period p_i based on historical ($p_1, p_2, \ldots p_{i-2}$), and current workload (p_{i-1}).

based on the expected workload for the future. For that, a workload prediction component is needed. Its goal is to asses expected workload by considering historical (workload) data and current (workload) data (Figure 4.7). If the predicted workload significantly varies from the current workload, the currently active configuration may be invalidated, and based on the benefit, a transition to a new one may be initiated (see Equation (4.6)).

Workload prediction consists of two main tasks, namely of the *workload monitoring* also known as tracking, and the *prediction model*. The goal of the first task is to continuously monitor the resource usage and to produce a representation of this usage data. The prediction model will then anticipate the future workload based on the current and historical resource usage data.

Different prediction models exist, such as EMA, Auto-Regressive Integrated Moving Average (ARIMA), cubic splines, etc., that differ in accuracy and cost [AC06]. In the context of this thesis, the workload prediction is just a tool to feed the configuration models of the CCQ protocols for choosing a suitable configuration and is not considered as a contribution. We decided to use EMA for predicting the workload due to its simplicity and ability to accurately predict certain workload types [AC06]. In what follows, we will provide an overview of time series prediction based on moving averages.

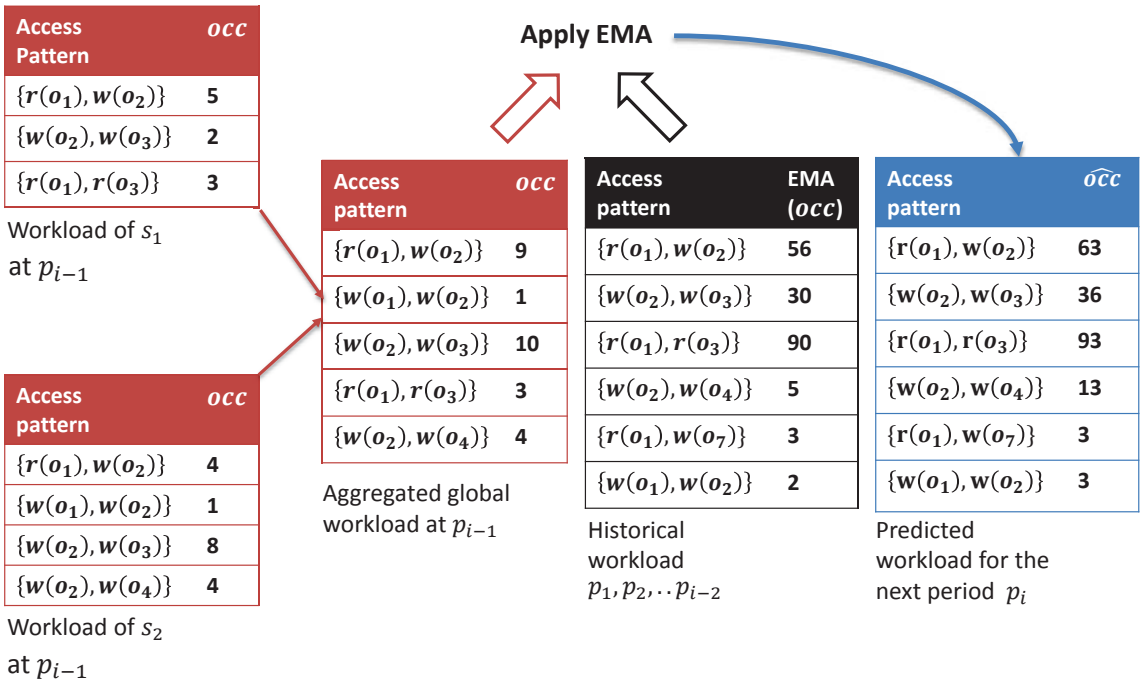


Figure 4.8: Workload prediction with EMA.

4.2.2 Time Series Prediction

A *time series* is a sequence of data points measured over time [AA13]. There are two widely used models for time series prediction, namely Autoregressive Model (AR) and Moving Average (MA). Their combination leads to Autoregressive Moving Average (ARMA), which are mathematical models for autocorrelation in time series, that allow the prediction of future values based on observed values in the past [SS11]. The basic idea is to predict future values based on one or more recent values, and/or one or more recent values of the errors [Nau].

An $ARMA(h, q)$ model is a combination of $AR(h)$ and $MA(q)$. $AR(h)$ assumes the predicted value $\widehat{Y^{p_i}}$ for period p_i to be a linear combination of h past values (Y):

$$\widehat{Y^{p_i}} = \theta_1 \cdot Y^{p_{i-1}} + \cdots + \theta_h \cdot Y^{p_{i-h}} + \epsilon^{p_i} \tag{4.12}$$

In Equation (4.12), $Y^{p_{i-1}}$ denotes the actual value at period p_{i-1}, θ the autoregressive coefficient, and ϵ^{p_i} the noise. The noise is also known as the error or the residual.

The *moving average (MA)* is a model that defines time series as a moving weighted average. The MA of order q is defined as follows:

$$\widehat{Y^{p_i}} = \epsilon^{p_i} + \alpha_1 \cdot \epsilon^{p_{i-1}} + \cdots + \alpha_q \cdot \epsilon^{p_{i-q}} \tag{4.13}$$

In Equation (4.13), $\epsilon^{p_{i-1}}$ denotes the residual for period p_{i-1}, and α denotes the order of the moving average.

$AR(h)$ and $MA(q)$ can be combined to an $ARMA(h, q)$ model as follows:

$$\begin{aligned} \widehat{Y^{p_i}} &= \theta_1 \cdot Y^{p_{i-1}} + \cdots + \theta^{p_{i-h}} \cdot Y^{p_{i-h}} + \epsilon^{p_i} \\ &\quad + \alpha_1 \cdot \epsilon^{p_{i-1}} + \cdots + \alpha_q \cdot \epsilon^{p_{i-q}} \\ \epsilon &\sim \ IID(0, \sigma^2) \end{aligned} \tag{4.14}$$

Different values of h and q lead to different models [SS11]. $ARMA(0,1)$ leads to a forecasting based on EMA, which is defined as follows:

$$\begin{aligned} \widehat{Y^{p_i}} &= \widehat{Y^{p_{i-1}}} + \alpha \cdot \epsilon^{p_{i-1}} \\ \epsilon^{p_{i-1}} &= Y^{p_{i-1}} - \widehat{Y^{p_{i-1}}} \\ \widehat{Y^{p_i}} &= \alpha \cdot Y^{p_{i-1}} + (1-\alpha) \cdot \widehat{Y^{p_{i-1}}} \end{aligned} \tag{4.15}$$

In Equation (4.15), $\widehat{Y^{p_{i-1}}}$ denotes the predicted value for p_{i-1}, which is based on all previous periods, and α the smoothing factor with $0 < \alpha < 1$. The predicted value is a weighted average of all past values of the series, with exponentially decreasing weights as we move back in the series. The value of α defines the decay rate of old values: a high value of α discounts older observations faster. It means that the higher the value, the more responsive the prediction and vice-versa. The effects of exponential smoothing can be better seen if we replace $\widehat{Y^{p_{i-1}}}$ in Equation (4.15) with its components:

$$\begin{aligned} \widehat{Y^{p_i}} &= \alpha \cdot Y^{p_{i-1}} + (1-\alpha) \cdot (\alpha \cdot Y^{p_{i-2}} + (1-\alpha) \cdot \widehat{Y^{p_{i-2}}}) \\ \widehat{Y^{p_i}} &= \alpha \cdot Y^{p_{i-1}} + \alpha \cdot (1-\alpha) \cdot Y^{p_{i-2}} + \mathbf{(1-\alpha)^2} \cdot \widehat{Y^{p_{i-2}}} \end{aligned}$$

4.2.3 Workload Prediction with EMA

The CCQ protocols use a workload prediction model based on EMA, which is a simple yet effective model able to accurately predict certain types of workloads [AC06].

The workload prediction based on EMA is depicted in Figure 4.8. CCQ protocols will periodically collect the workload of each site and add them to the aggregated workload. Once the aggregated workload has reached a specific size, as defined by a threshold, the workload for the new interval p_i is predicted, the old interval p_{i-1} is closed, and its workload is added to the set of historical workloads. Notice that the historical workload contains only the predicted workload for the last interval as the predictions are recursively based on the past, i.e., a prediction for the interval p_{i-1} includes and is based on the prediction of previous intervals $\{p_{i-2}, ..., p_1\}$ (Equation (4.15)):

$$\widehat{wload^{p_i}} = \alpha \cdot wload^{p_{i-1}} + (1-\alpha) \cdot \widehat{wload^{p_{i-1}}} \tag{4.16}$$

4.3 CCQ Configuration and Cost Model

The configuration spaces of the CCQ protocols are depicted in Figure 4.9. Each CCQ protocol targets the reduction of cost for certain data management properties, that are

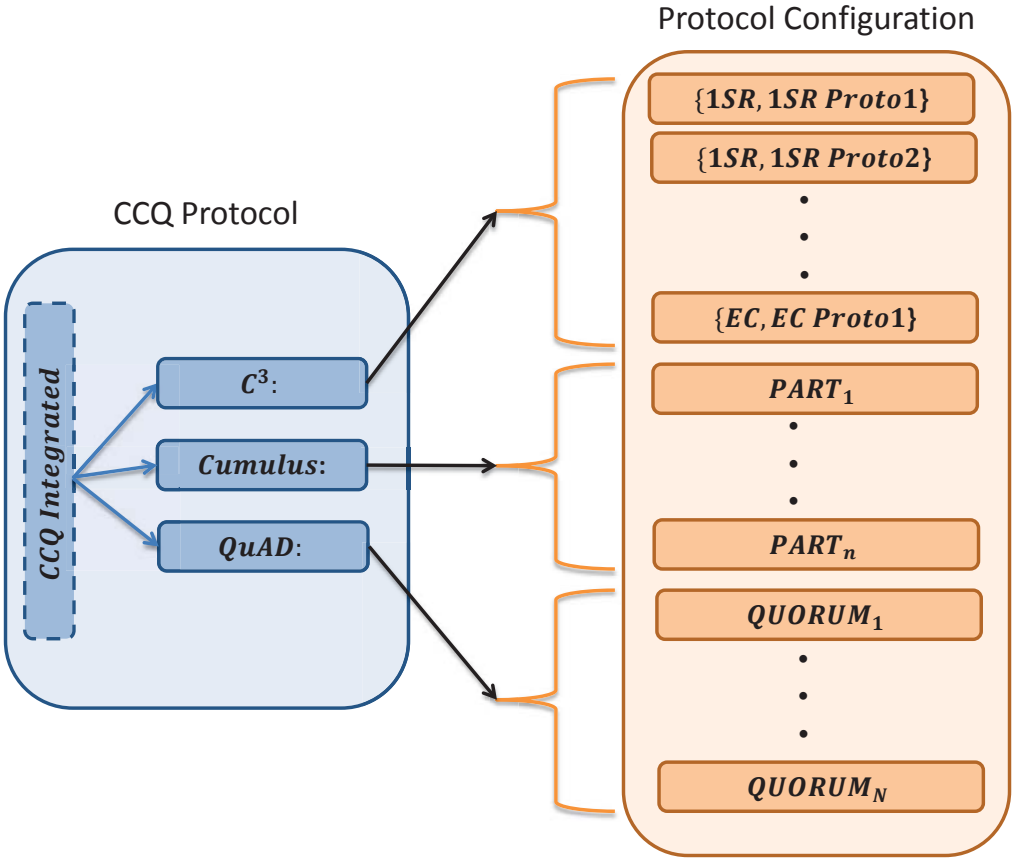


Figure 4.9: CCQ configuration space.

specified by application requirements. It is very challenging, if not impossible, to consider the full spectrum of data management properties and the dependencies between them. Therefore, the CCQ protocols make certain assumptions, which are defined as follows.

4.3.1 C^3

- The configuration space of C^3 is defined by the available $\langle Model, Protocol \rangle$ combinations (configurations). The *Model* denotes the consistency model, and *Protocol* the protocol implementing that model. Each model can be implemented by more than one protocol.

- The application provider specifies a certain availability requirement (e.g., $availability \geq 0.99999$) that must not violated, i.e., its violation will incur infinite costs. The availability requirement determines the number of replica sites and their location. This aspect is predefined and outside the control of C^3.

- The application provider defines penalty costs for the violation of consistency, i.e., each generated inconsistency incurs a certain penalty.

- C^3 targets the optimization of transaction costs by choosing the most suited configuration given the application workload, the number of sites and their locations. The transaction costs are determined by the inconsistency (*pCost*) and consistency (*opCost*) costs.
- The *pCost* are determined by the consistency model, the workload (number of inconsistencies), and the cost of a single inconsistency.
- The *opCost* are determined by the consistency model, the workload, number of sites, and their locations.

4.3.2 Cumulus

- The configuration space of Cumulus is defined by the set of possible partitions.
- The application provider specifies that 1SR must not be violated.
- Cumulus targets the optimization of operational costs by choosing the most optimal partition configuration.
- The *opCost* is determined by the number of distributed transactions in a workload. Cumulus will partition the data horizontally (see Section 3.2.1), so that the number of distributed transactions is reduced or completely avoided.

4.3.3 QuAD

- The configuration space of QuAD is defined by the set of possible quorum configurations.
- The application provider has specified that 1SR consistency and a certain availability level must be guaranteed, as their violation would incur infinite penalty costs. The availability requirement applies for all data, which, as a consequence, leads to a fully replicated DDBS.
- QuAD targets the optimization of operational costs by constructing the quorums in such a way so that the transaction response time is reduced as far as possible. The lower the response time the higher the generated utility for the application (see Equation (4.2)).
- The dynamic replication, i.e., the provisioning and deprovisioning of sites either due to load changes or site failures, is outside the control of QuAD.

4.3.4 Integrated CCQ

Although each CCQ protocol considers specific data management properties, and has a specific configuration space, they are not independent of each other. Moreover, they can be integrated to a meta-protocol called *Integrated CCQ* (I-CCQ) based on a holistic model. I-CCQ considers also the dependencies between the CCQ protocols, and is

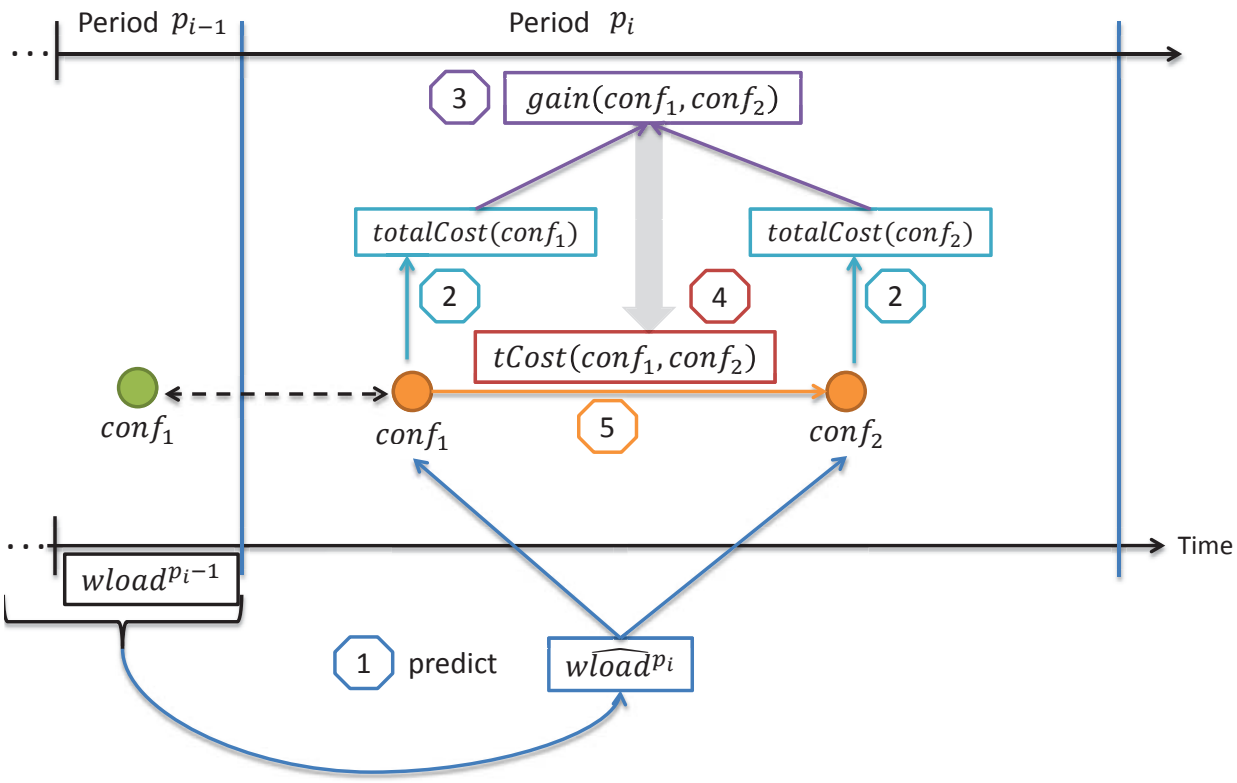


Figure 4.10: Cost and workload-driven reconfiguration.

able to fully exploit the entire CCQ configuration space. Furthermore, the configuration space of the meta-protocol can be enhanced by incorporating further protocols, that implement additional data management properties, or by incorporating further configurations into the existing protocols. In what follows we will provide two examples that depict the behavior of the I-CCQ meta-protocol (see also Example 4.4).

Example 4.6 (I-CCQ: From Partitioned to Partially Replicated Database)

Similar to the Example 4.4, let us assume that the application does not have any availability requirements towards the DBS, and that it has only specified how much an inconsistency would cost. As it is not necessary to provide any availability guarantees, *I-CCQ* will choose to partition the data by using the capabilities of Cumulus. Based on the predicted workload, Cumulus will determine the most optimal partition configuration, which in the optimal case, will completely avoid distributed transactions.

1SR and EC generate the same costs for workloads without distributed transactions. Thus, it is obvious that the *I-CCQ* will provide 1SR consistency to the application. For that, *I-CCQ* will use C^3 and ask it to enforce 1SR correctness. C^3 has the freedom to choose the most optimal 1SR protocol.

The application may later decide to increase the availability for certain partitions. This decision may be based on either changed business constraints, or changed end-user behavior. *I-CCQ* may use the Accordion protocol [SMA+14], that will replicate the affected partitions. In that case, the distribution model changes from partitioned to

partially replicated (Section 3.2.1). As now certain partitions are replicated, and as the application has specified the cost for a single inconsistency, *I-CCQ* will hand-in the control for data consistency to C^3 for the replicated partitions. C^3 will run in the adaptive mode, i.e., it will choose the most optimal consistency model and protocol based on the workload, and will continuously adapt its configuration if the workload shifts.

Example 4.7 (I-CCQ: ROWAA vs. Quorum-Protocol for 1SR Consistency)

In this scenario, we assume that the application has specified the desired availability, which leads to a fully replicated database (Section 3.2.1). Moreover, the application demands 1SR as the cost for an inconsistency is extremely high, possibly infinite. *I-CCQ* will forward control to C^3, which will initiate the workflow for determining the optimal configuration. As the consistency level is determined by the application, C^3 will search for the most optimal protocol based on the workload. In case of a read-heavy workload, C^3 will use ROWAA as a replication protocol. However, ROWAA generates considerable overhead for update transactions, as all available sites must be eagerly committed. Therefore, at some point in time, when the proportion of the update transactions in the workload is above a specific threshold, C^3 will initiate a reconfiguration, and will switch to a quorum-based RP, such as QuAD. QuAD is able to adjust the quorums in such a way so that 'weak' (e.g., slow, expensive, distant) sites, that are the main source of performance degradation, are avoided from the read and commit paths of transactions.

4.3.5 CCQ Adaptive Behavior

In what follows we will provide a step-by-step description of the CCQ adaptive behavior. Let us assume that the configuration space consists of two configurations: $\mathbb{C} = \{conf_1, conf_2\}$ (Figure 4.10), with $conf_1$ being the active configuration at p_{i-1}.

- In the first step, before p_{i-1} is finished[4], the workload prediction for p_i is initiated based on the EMA model (Algorithm 1).
- In the second step, the predicted workload $\widehat{wload}^{p_i}$ is used as a basis for calculating (predicting) the costs of each configuration (Equation (4.3)).
- There are two choices, namely to remain with the currently active configuration $conf_1$, or to switch to $conf_2$. The decision is taken by considering the expected gain (step three as defined by Equation (4.4)), but also the transition costs (step four as defined by Equation (4.5)). The later is determined mainly by $wload^{p_{i-1}}$ and the requirements of $conf_2$.
- If the expected gain outweighs the transition costs than, by moving to $conf_2$, a benefit is generated for applications (Equation (4.6)). In that case, a transition to $conf_2$ is initiated (step five).

[4] A period is finished if the size of the current workload is above a specified threshold.

The workload prediction may impose a considerable challenge with regards to the size of the data that needs to be transferred between the sites. The CCQ protocols use a centralized approach in which a centralized component collects the workload data from each site so that it can assess the future workload. The centralized component decreases the transfer overhead compared to fully distributed workload prediction approaches, in which each site has to send its workload to all other sites. However, the existence of a centralized component makes the system less resilient to failures. The transfer of the local workloads can be further reduced, or even completely avoided, by allowing each site to asses global workload based on local data. Possible approaches include the hypothesis testing in a fully distributed manner and statistical inference with partial data [TSJ81, Vis93].

Algorithm 1: Algorithm for collecting and predicting the application workload. A workload prediction is initiated once the size of the collected workload is above a threshold. The predicted workload is used as a basis for determining the optimal CCQ configuration. The size of the managed data may grow with time. However, as only those access patterns are maintained that have an $occ > 0$, infrequent access patterns diminish with time and are removed from the workload.

Input: *collFreq* : Collection Frequency, *threshold* : Workload Threshold, S : Set of Sites

while *true* **do**
- `sleep(collFreq);`
- `/* Current workload at` p_{i-1} `*/`
- $wload^{p_{i-1}} \leftarrow \{\}$;
- **foreach** $s \in S$ **do**
  - $wload^{p_{i-1}}$`.add(`s`.getWorkload());`
- **if** $wload^{p_{i-1}}$`.size()` $\geq$ threshold **then**
  - $\widehat{wload^{p_i}} \leftarrow$ `predict(predict,` $\widehat{wload^{p_{i-1}}}$`,` $wload^{p_{i-1}}$`);`
  - `// The function for determining the optimal configuration is implemented by each CCQ protocol.`
  - `determineConfiguration(`$\widehat{wload^{p_i}}$`);`
  - $i++$;

Procedure `predict(`α`,` $\widehat{wload^{p_{i-1}}}$`,` $wload^{p_{i-1}}$`)`
- **foreach** $ap \in \widehat{wload^{p_{i-1}}}$ **do**
  - `// The lookup for an access pattern is an` $\mathcal{O}(1)$ `operation, as each access pattern has an unique hash-code.`
  - $\widehat{occ(ap)}^{p_{i-1}} \leftarrow \widehat{wload^{p_{i-1}}}$`.getOccOfAccessPattern(`$ap$`);`
  - $occ(ap)^{p_{i-1}} \leftarrow wload^{p_{i-1}}$`.getOccOfAccessPattern(`$ap$`);`
  - `// Remove` ap `from the current workload, so that at the end the access pattern that are new in the current workload remain.`
  - $wload^{p_{i-1}}$`.removeAccessPattern(`ap`);`
  - $\widehat{occ(ap)}^{p_i} \leftarrow \alpha \cdot occ(ap)^{p_{i-1}} + (1-\alpha) \cdot \widehat{occ(ap)}^{p_{i-1}}$;
  - **if** $\widehat{occ(ap)}^{p_i} > 0$ **then**
    - $\widehat{wload^{p_i}}$`.add(`ap`,`$\widehat{occ(ap)}^{p_i}$`);`
- `// Now predict occurrence of the remaining access patterns that are new in the current workload.`
- **foreach** $ap \in wload^{p_{i-1}}$ **do**
  - $occ(ap)^{p_{i-1}} \leftarrow wload^{p_{i-1}}$`.getOccOfAccessPattern(`$ap$`);`
  - `/* As new patterns were not part of the last prediction, simply take over their occurrence. */`
  - $\widehat{occ(ap)}^{p_i} \leftarrow occ(ap)^{p_{i-1}}$;
  - **if** $\widehat{occ(ap)}^{p_i} > 0$ **then**
    - $\widehat{wload^{p_i}}$`.add(`ap`,`$\widehat{occ(ap)}^{p_i}$`);`
- **return** $\widehat{wload^{p_i}}$;

5

Design of the Cost- and Workload-driven CCQ Protocols

IN THIS CHAPTER, we will describe the design of the CCQ protocols, that target the reduction of application costs by choosing a suitable configuration. This choice is driven by application defined cost parameters and workload, as well as by infrastructure properties. All CCQ protocols follow the same design schema, which is defined as follows. In the first step, specific information, such as conflicts between transactions or number of distributed transactions, is derived from the predicted workload as required by the configuration and cost model of the concrete protocol. In the next step, this information, together with the cost parameters defined by the application and infrastructure properties, are provided as input to the configuration and cost model that decides on the cost-optimal configuration. Finally, the reconfiguration process will ensure that a transition from one configuration to another is done in a safe manner, without violating the system correctness.

The purpose of this chapter is to zoom-in into the configuration space of each CCQ protocol, and describe their cost models that steer the runtime configuration with the goal of reducing application costs.

5.1 C^3: Cost and Workload-Driven Data Consistency in the Cloud

C^3 is a cost and workload-driven consistency protocol for the Cloud, that is able to adjust the consistency level at runtime by considering the operational – *consistency*, and the penalty – *inconsistency* – costs. C^3 is based on the consistency-cost trade-off described in Section 3.4 (Figure 5.1). The stronger the consistency model, the higher the consistency costs. Relaxing consistency decreases the operational costs, but might generate inconsistency costs.

The consistency costs are defined in terms of resources and activities needed for guaranteeing a certain consistency level, whereas the inconsistency costs are deter-

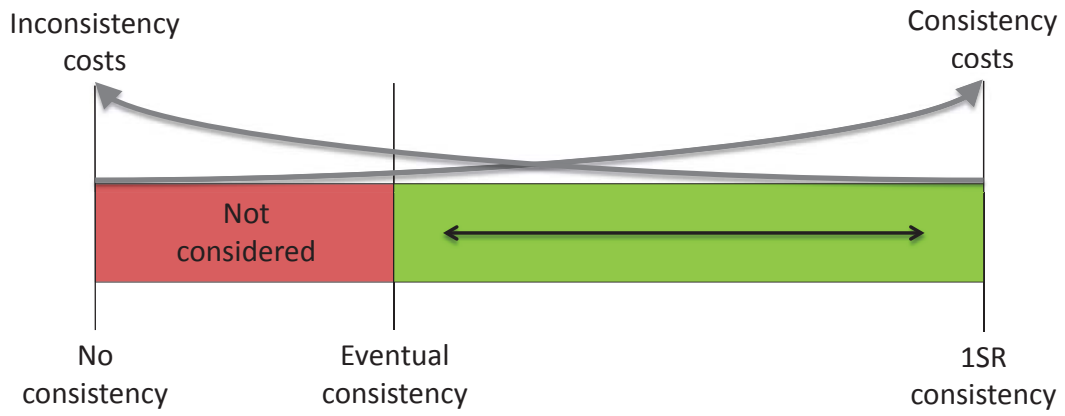


Figure 5.1: Consistency vs. inconsistency costs. The stronger the consistency level the lower the consistency costs and vice-versa. The inconsistency costs increase with decreasing consistency level.

mined by the overhead generated for compensating the effects of inconsistencies, when relaxed consistency levels are used. The goal of C^3 is, given a certain workload, to choose that consistency level that incurs the lowest total costs, which are defined as the sum of consistency and inconsistency costs.

C^3 considers only the spectrum of consistency models that lay in the ranged defined by EC and 1SR. All consistency models weaker than EC are of no interest despite the fact that they generate low or no consistency costs. The main reason is their weak correctness semantics, which makes them difficult to reason about. EC is the "bare minimum" of correctness to be provided [BGHS13].

The C^3 results have been published in [FS12].

5.1.1 C^3 Overview

The idea of C^3 is to add a cost and workload-driven layer on top of existing consistency models and protocols implementing these models (Figure 5.2). A configuration is defined by the $\langle model, protocol \rangle$ combination, which means that the configuration space of C^3 is determined by the available consistency models, and protocols implementing these models. The combination of the C^3 meta-consistency model and the C^3 meta-protocol defines a new *meta-configuration* (Figure 4.4). The consistency model defines the observable correctness, whereas the protocol its implementation.

The goal of C^3 is to navigate at runtime between the different configurations so that transactions costs, given a certain workload and application defined inconsistency costs, are minimized. Each configuration may generate different costs for a the same workload (see Example 5.1).

Example 5.1 (Cost and Workload-driven Configuration)

In a concrete scenario the application developers may specify that the 1SR consistency level must be enforced, in order to avoid any inconsistencies. This can be achieved by

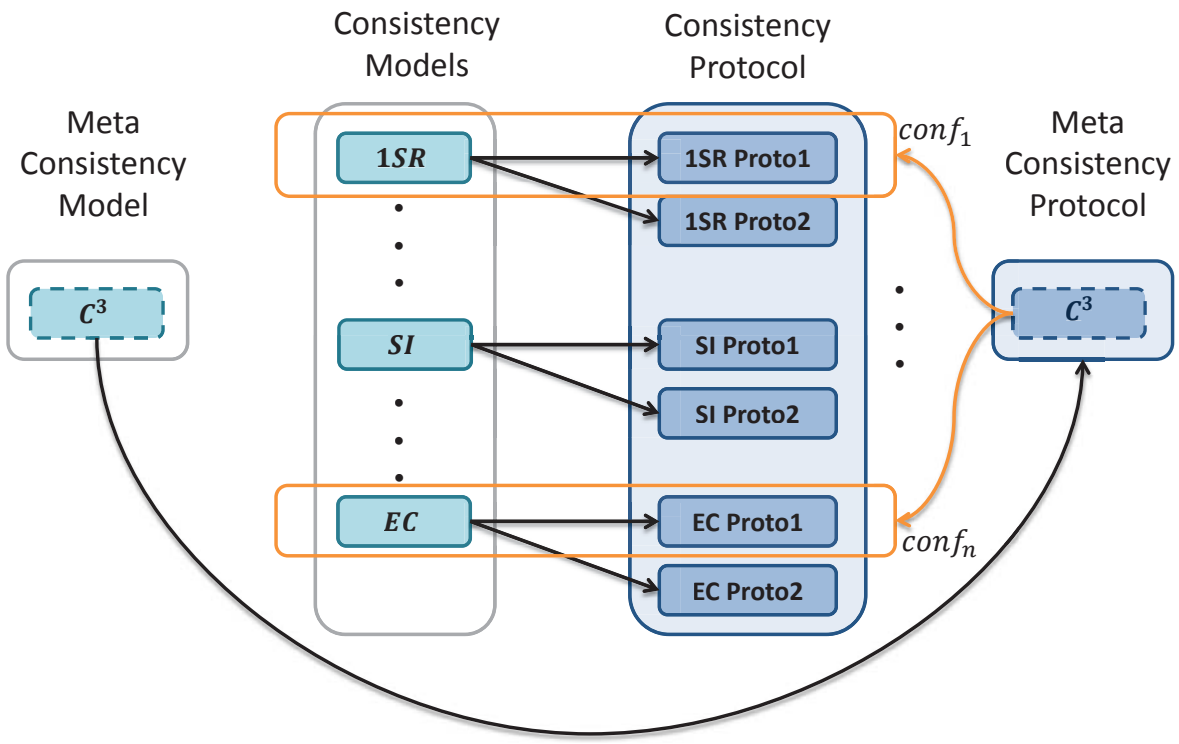


Figure 5.2: Relationship between consistency models and protocols. A configuration is determined by the $\langle model, protocol \rangle$ combination.

setting the inconsistency costs to infinity. The decision may be a consequence of business or legal constraints. In this case, the C^3 configuration space is narrowed down to the 1SR consistency model. However, different 1SR protocols may exist, so that C^3 will start an evaluation process to determine the most appropriate protocol based on the workload (see Negotiation Process in Section 2.3). For read-only workloads, a protocol based on S2PL and ROWAA may be the most optimal implementation in terms of generated costs, whereas for update-heavy workloads a quorum RP may be more suitable.

The C^3 configuration space can be extended by adding new configurations, i.e., combinations of consistency models and protocols. The provider of a configuration must define a model for calculating the costs for a given workload. A transition from one configuration to another configuration of the same consistency model does not generate any costs, as configurations of the same consistency model deliver the same correctness guarantees. Transition costs incur only during a transition between configurations of different models. However, it is necessary to define these costs only once, as they apply for all protocols of a consistency model. The applicability of the transition cost models to all protocols means that it should be incorporated into C^3, in order to relief the configuration providers from the burden of defining the transition costs.

In its current version, C^3 implements the 1SR and EC consistency models based on well-known protocols (Figure 5.3). It assumes a fully replicated DDBS with a set of

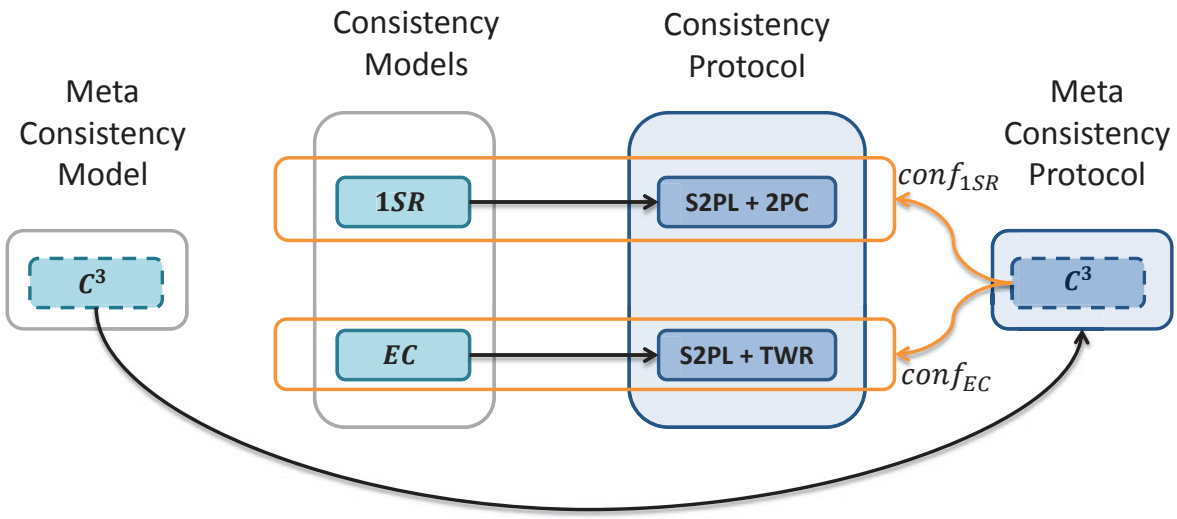


Figure 5.3: C^3 Configuration Space: $\mathbb{C}_{C^3} = \{1SR, EC\}$.

sites at specific locations. Both, the number of sites and their location are determined by the availability requirement of applications that must not be violated (Section 4.3). C^3 can be run in the adaptive or in the traditional mode. In the adaptive mode application providers specify inconsistency costs and C^3 will choose that configuration that minimizes the overall costs. In the traditional mode, a certain correctness model is enforced. In the later case, C^3 will determine the most optimal protocol that implements the enforced consistency model.

In what follows, we will describe in detail the protocols that are used to implement the 1SR and EC consistency models, and we will define their cost and configuration models.

5.1.2 1SR and EC Protocol Definition

Both the 1SR and EC protocols use S2PL as a CCP. Once the shared and exclusive locks are acquired, the transaction execution is initiated. While the protocols behave the same with regards to the CCP, they differ when it comes to the RP. In contrast to the 1SR protocol that uses 2PC for the eager synchronization of replica sites, EC is based on a lazy RP (Section 3.3.3).

Transaction execution in C^3 is asymmetric [JPAK03], which means that a transaction is executed at the local site and the generated values are propagated to other sites. This is in contrast to log-based approaches, in which the transaction logs, i.e., its actions, are propagated. In that case, the transaction needs to be re-executed at the remote sites. The propagated data consists of a set of triples for each of the modified object:

$$\{\langle o_{id}, val, \tau(o_{id})\rangle, \dots\}$$

In case of 1SR the updates are propagated during the exchange of the `prepare` messages as part of 2PC, whereas in case of EC the updates are propagated through

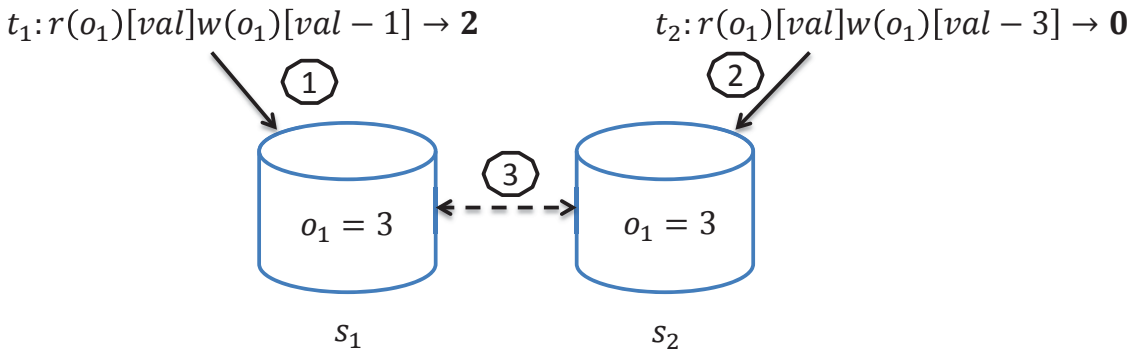


Figure 5.4: Execution of transactions with the EC consistency level.

dedicated `synchronization` transactions. The uncoordinated transaction execution in EC, which is a consequence of the lazy synchronization, necessitates the existence of a mechanism that allows to agree on final values for the modified objects (see Example 5.2).

Example 5.2 (Lazy Synchronization)

Let us consider the scenario depicted in Figure 5.4, in which two transactions t_1 and t_2 are executed with the EC consistency level, and both modify the same object (o_1). The final result of t_1 is 2 and that of t_2 is 0. Although there is a conflict between t_1 and t_2, as there is no coordination during there execution, both transactions will successfully commit. Later, as part of step 3, each site will propagate the updates to the other sites in the system. During the synchronization, a means should be available to agree on the final value of o_1, which in this case can either be 2 or 0. Independently of the final agreement, one of the updates will be lost.

C^3 uses the TWR [Tho79,BLFS12,TTP+95] for reaching an agreement during the synchronization of sites (see Definition 3.36), and guarantees that all sites will eventually reflect the same values .

TWR requires each transaction to be assigned a system wide unique timestamp. At transaction commit, that timestamp is attached to each modified object, which needs to be also propagated together with the object values during the replica synchronization. Although TWR guarantees that sites will eventually reflect the same values, it does not ensure that the reflected state is free of inconsistencies (Example 5.3).

Example 5.3 (Inconsistencies in an Eventually Consistent Database)

Let us consider again the scenario depicted in Figure 5.4. In a DDBS that provides 1SR transactions are serialized either as $t_1 t_2$ or $t_2 t_1$. In both cases the final value of o_1 would be -1. In an eventually consistent system, one of the updates would be lost. So, in case the timestamp of t_1 is higher than that of t_2 the end value would be 2, and in the opposite case it would be 0.

Symbol	Description
$\mathbb{C}_{C^3}$	Denotes the configuration space of C^3.
$totalCost(wload, CL)$	Denotes the total costs generated by the consistency level $CL \in \{1SR, EC\}$ given the specified workload $wload$.
$pCost(wload, CL)$	Denotes the inconsistency costs generated by the consistency level CL given the specified workload $wload$.
$incCost$	Denotes the cost of a single inconsistency.
$nrInc(wload, CL)$	Denotes the number of inconsistencies generated by CL given $wload$.
$opCost(wload, CL)$	Denotes consistency costs generated by CL given $wload$.
$opCost(wload, ccp_{CL})$	Denotes the operational costs generated by the CCP of CL.
$opCost(wload, rp_{CL})$	Denotes the operational costs generated by the RP of CL.
$cost_{2pcmess}$	Denotes the cost of a single 2PC message.
$nrT_u(wload)$	Denotes the number of update transactions in $wload$.
$conflictRate(s_i, s_j)$	Denotes the w/w conflict rate between the workloads of s_i and s_j.
$s.wactions$	Denotes the set of write actions executed at the site s.
$s.card(o)$	Denotes the number of distinct transactions that execute a write action on object o at site s.
$conflictset(s_i, s_j)$	Denotes the set of common objects in s_i and s_j that are accessed by write actions.
$\widehat{gain}(\widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new})$	Denotes the expected saving in costs when the predicted workload $\widehat{wload^{p_i}}$ is executed with $CL^{p_i}_{new}$ compared to the costs generated by $CL^{p_{i-1}}_{current}$.
$tCost(wload^{p_{i-1}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new})$	Denotes the costs of a transition from $CL^{p_{i-1}}_{current}$ to $CL^{p_i}_{new}$ given the executed $wload^{p_{i-1}}$ with $CL^{p_{i-1}}_{current}$.
$nrModObj(wload, CL)$	Denotes the number of modified objects in $wload$ by transactions running with the CL consistency level.
$\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new})$	Denotes the expected benefit when the predicted workload $\widehat{wload^{p_i}}$ is executed with $CL^{p_i}_{new}$ compared to that of $CL^{p_{i-1}}_{current}$.

Table 5.1: C^3 symbols and notations.

If there is constraint $o_1 \geq 0$ set on o_1, then in case of 1SR one of the transactions would be aborted. With EC both transactions would successfully commit, leading to, for example, an oversell in an online shop. This oversell generates administrative cost incurring from the business process to be executed for compensating the inconsistency (e.g., contacting the customer and canceling the order).

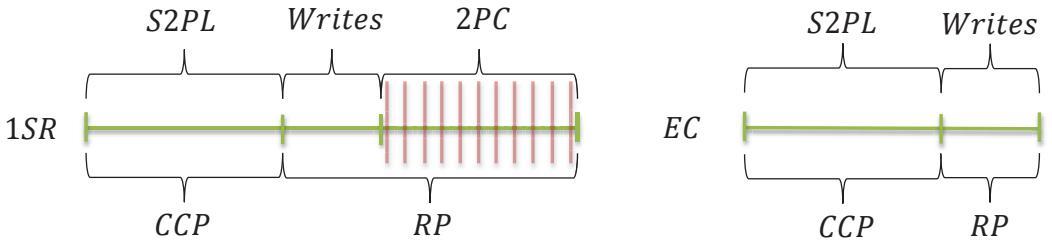


Figure 5.5: Consistency costs of 1SR and EC.

5.1.3 Cost Model

In what follows we will describe the cost models for the 1SR and EC consistency models based on the protocols described above. The list of symbols is summarized in Table 4.2 and 5.1.

The total costs of a consistency configuration CL is defined as the sum of its consistency and inconsistency cost given a certain workload $wload$:

$$totalCost(wload^{p_i}, CL) = \underbrace{pCost(wload^{p_i}, CL)}_{\text{inconsistency cost}} + \underbrace{opCost(wload^{p_i}, CL)}_{\text{consistency cost}} \tag{5.1}$$

The inconsistency costs are determined by the number of inconsistencies ($nrInc$) and the application specific cost for one consistency ($incCost$):

$$pCost(wload^{p_i}, CL) = incCost \cdot nrInc(wload^{p_i}, CL) \tag{5.2}$$

As 1SR does not generate any inconsistencies, its $pCost(wload^{p_i}, 1SR) = 0$. The consistency costs capture the overhead generated in terms of resources and activities by a certain consistency protocol given the workload $wload^{p_i}$. A protocol consists of a CCP and RP. Thus, its overall overhead is defined by the overhead of the CCP and RP:

$$opCost(wload^{p_i}, CL) = opCost(wload^{p_i}, CCP_{CL}) + opCost(wload^{p_i}, RP_{CL}) \tag{5.3}$$

As we described above, the 1SR and EC protocols use the S2PL as a CCP: $opCost(wload^{p_i}, ccp_{1SR}) = opCost(wload^{p_i}, ccp_{EC})$. The main difference between them lies in the generated costs for the RP (see Figure 5.5), and the inconsistency costs. In what follows we specify the cost model for 1SR and EC, and consider only the additional overhead generated by one protocol compared to the other.

1SR Costs

The total costs generated by 1SR is defined as follows:

$$totalCost(wload^{p_i}, 1SR) = opCost(wload^{p_i}, 1SR) = opCost(wload^{p_i}, 2pc) \tag{5.4}$$

As defined by Equation (5.4), the costs generated by the 2PC also determines the overall 1SR costs. Each 2PC message generates costs according to the pricing model defined in Section 4.1.1:

$$opCost(wload^{p_i}, 2pc) = cost_{2pcmess} \cdot nrT_u(wload^{p_i}) \cdot (|S| - 1) \tag{5.5}$$

Equation (5.5) defines the 2PC cost by considering the number of update transactions (nrT_u) in the workload, cost of each message ($cost_{2pcmess}$), and the number of sites (notice that only the number of agents accounts for the costs).

EC Costs

As EC does not use 2PC for the synchronization of replica sites, its $opCost(wload^{p_i}, EC) = 0$. This, however, does not mean that EC does not generate any operational costs, as EC also uses S2PL for the synchronization of concurrent transactions (see Figure 5.5). We simply consider the difference in the cost components, i.e., the costs components that are available in 1SR and not in EC. Adding new consistency models and protocols into C^3 may lead to modifications of the cost model in order to consider their cost components.

In contrast to 1SR, transactions run with the EC consistency level may lead to inconsistent data (see Example 5.3). Thus, $pCost(wload^{p_i}, EC) \geq 0$ (Equation (5.2)). It follows that the overall costs of EC are determined by the inconsistency costs given a certain workload:

$$totalCost(wload^{p_i}, EC) = pCost(wload^{p_i}, EC) \tag{5.6}$$

Calculation of EC Inconsistency Costs

C^3 defines inconsistencies based on `w/w` conflicts that occur between transactions running with EC at different sites. More concretely, we have considered the lost-update inconsistency that can occur as a consequence of `w/w` conflicts. Lost-updates occur when the updates of one transaction overwrite the updates of another transaction (Example 5.2). However, the number of lost updates is dependent on the ordering of transactions, i.e., the choice of the *last-committer* as defined by the TWR (Example 5.4).

Example 5.4 (Calculation of Inconsistencies for EC Transactions)

Let us consider the scenario depicted in Figure 5.6 with the given workload at each site. Further, let us assume that transactions executed at s_1 have a higher timestamp than all other transactions executed by s_2 and s_3. In that case, updates of t_3 t_4 and t_5 will be lost after the synchronization of the replica sites. Concretely, four updates will be overwritten by the updates of s_1. Figure 5.6 depicts only write actions for reasons of readability. C^3, however, does not impose any limitations on the structure of transactions. Furthermore, as local transactions are synchronized according to the S2PL, locally at the sites no lost-updates can occur, except in the case when a transaction contains blind writes.

Predicting the time-based ordering of transactions in a distributed system is very challenging, if not impossible. Therefore, an approximation of the ordering is necessary. In C^3, this is done by declaring the site with the greatest workload as the last-committer.

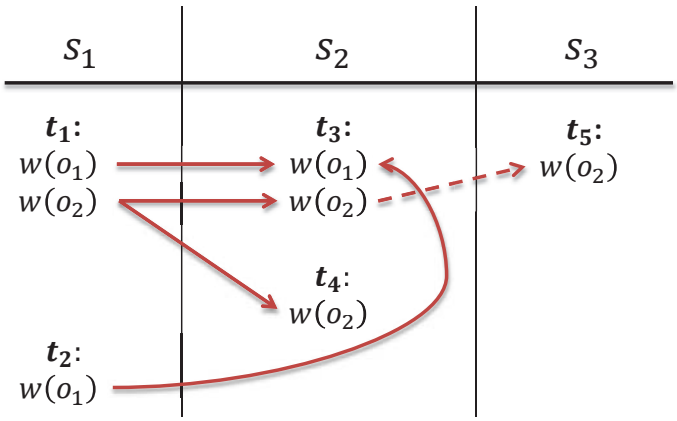


Figure 5.6: Calculation the number of lost-updates given a certain workload.

The decision is based on the assumption that the site with the greatest workload generates also the highest inconsistency impact on the transactions executed by other sites, i.e., generates the highest number of lost-updates. Such an approach avoids the necessity of an expensive pairwise calculation of the impact each site has on all other sites. Thus, a site s_i is declared as a last-committer if the following condition holds:

$$s_i : last - committer \Leftrightarrow \forall s_j \in S \wedge i \neq j : |wload(s_i)| \geq |wload(s_j)| \tag{5.7}$$

Based on Equation (5.7) we assume that the transactions of the site with the greatest workload have the highest timestamp, and will be the last-committer according to the TWR. In scenario depicted Figure 5.6, either s_1 or s_2 is declared as a last-committer. The calculation of inconsistencies is then as follows.

Let $conflictRate(s_i, s_j)$ denote the number of transactions executed at s_j that are in w/w conflict with those executed at s_i. $conflictRate(s_i, s_j)$ defines at the same time the impact of s_i's transactions to those of s_j, if s_i would be declared the last-committer according to the TWR. The transactions of the last-committer would overwrite the updates conducted by the transactions of at the other sites. Thus, the number of lost-updates, i.e., inconsistencies in case of s_i being the last-committer is defined as follows (see Example 5.4):

$$nrInc(wload^{p_i}, EC)^{s_i} = \sum_{s_j \in S \wedge j \neq i} conflictRate(s_i, s_j) \tag{5.8}$$

The conflict rate between two sites is calculated as follows (see Algorithm 2). Let $s.wactions$ denote the set of objects accessed by write operations at s. Further, let $s.card(o)$ denote the cardinality, i.e., the number of distinct transactions that execute a write action on object o, and $conflictset(s_i, s_j) = s_i.wactions \cap s_j.wactions$. Then:

$$conflictRate(s_i, s_j) = \sum_{o \in conflictset(s_i, s_j)} s_j.card(o) \tag{5.9}$$

For the calculation of the lost-updates only the write actions need to be considered.

Algorithm 2: Algorithm for calculating the inconsistencies given the workload of two sites. The complexity is $\mathcal{O}(|s_2.wactions|)$.

Input: $s_1.wactions : \{< o_{id}, o_{id}.card >\}, s_2.wactions : \{< o_{id}, o_{id}.card >\}$
Output: Number of updates lost at s_2

```
Function calculateInconsistencies is
    numberOfLostUpdates ← 0 ;
    if isEmpty(s1.wactions) || isEmpty(s2.wactions) then
        return numberOfLostUpdates;
    foreach o ∈ s2.wactions do
        // contains(o) and card(o) are O(1) operations.
        if s1.wactions.contains(o) then
            numberOfLostUpdates ← numberOfLostUpdates + s2.card(o);
    return numberOfLostUpdates;
```

5.1.4 Configuration Model

In what follows we will describe the configuration model that allows C^3 to determine the most optimal configuration, and continuously adapt its configuration based on the application workload.

The total costs of a consistency level is normalized to a range $[0..1]$ as follows ($CL \in \{1SR, EC\}$):

$$\overline{totalCost(wload^{p_i}, CL^{p_i})} = \frac{totalCost(wload^{p_i}, CL^{p_i})}{\sum_{CL \in \mathbb{C}_{C^3}} totalCost(wload^{p_i}, CL)} \tag{5.10}$$

Given a currently active consistency level $CL^{p_{i-1}}_{current}$ at p_{i-1} and the predicted workload $\widehat{wload^{p_i}}$, the expected gain switching to another $CL^{p_i}_{new}$ is defined as follows:

$$\begin{aligned} \widehat{gain}(\widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new}) = \widehat{totalCost}(\widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}) \\ - \widehat{totalCost}(\widehat{wload^{p_i}}, CL^{p_i}_{new}) \end{aligned} \tag{5.11}$$

As already described, a transition from one consistency level (configuration) to another, may generate costs (Figure 5.7). While a transition from 1SR does not generate transition costs, a transition from EC necessitates a reconciliation of sites, so that 1SR transactions observe the same data independently on the site. The reconciliation is done based on the TWR as described in Section 5.1.2. Although at some point the reconciliation would anyways take place, C^3 penalizes a transition from EC to 1SR, as the forced reconciliation may happen at an inappropriate time (e.g., when the system load is very high). This in order to account for the possibility of the lazy protocol to choose the most suitable timepoint for reconciliation (see Section 3.3.3). The transition costs are defined as follows:

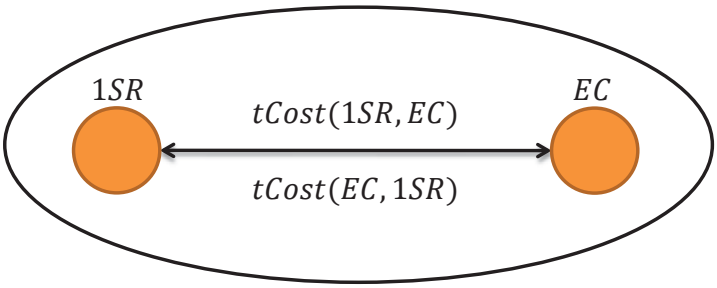


Figure 5.7: Transitions between consistency configurations.

$$
\begin{aligned}
tCost(wload^{p_{i-1}}, 1SR, EC) &= 0 \\
tCost(wload^{p_{i-1}}, EC, 1SR) &= \frac{nrModObj(wload^{p_{i-1}}, EC)}{|LO|} \cdot exp^{(1-\frac{1}{load(S)})} \\
load(S) &= \frac{\sum_{s\ inS}^{load(s)}}{|S|} \\
load(s) &\in]0, 1]
\end{aligned}
\tag{5.12}
$$

Equation (5.12) penalizes a transition from EC to 1SR by considering the number of modified objects during the EC period, and the average load in the system. The penalty would be maximized if all objects need to be updated at the maximum average load.

The consistency configuration $CL^{p_i}_{new}$ generates a benefit, if its expected gain outweighs the transition costs from the current consistency configuration $CL^{p_{i-1}}_{current}$ to $CL^{p_i}_{new}$.

$$
\begin{aligned}
\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new}) = \\
\widehat{gain}(\widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new}) \\
- tCost(wload^{p_{i-1}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new})
\end{aligned}
\tag{5.13}
$$

C^3 is an adaptive protocol that adjusts consistency dynamically so that the benefit is maximized:

$$
\max_{CL \in \{1SR, EC\}} \widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new}) \tag{5.14}
$$

Adaptive Thresholding

In order to account for errors in the workload prediction, C^3 incorporates an additional verification step, which is based on the probability that the expected total costs of one consistency level will be smaller/greater than the expected costs of the another consistency level. A transition from the currently active consistency level to the new one is conducted only if the following condition holds:

$$\begin{aligned}
&\textbf{(1)}\ P[(\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new})) > \\
&\quad (\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_{i-1}}_{current}))] \\
&\quad > \\
&\textbf{(2)}\ P[(\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_i}_{new})) < \\
&\quad (\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, CL^{p_{i-1}}_{current}, CL^{p_{i-1}}_{current}))]
\end{aligned} \tag{5.15}$$

The first part of Equation (5.15) denotes the probability that the new configuration generates a higher benefit compared to the old one for the expected workload. The second part denotes the opposite, i.e., the probability that the old configuration generates a higher benefit compared to the new configuration. If the the probability of the first part is higher than that of the second part, then C^3 will adjust the consistency configuration, i.e., will initiate a transition to $CL^{p_i}_{new}$.

C^3 assumes that the benefit generated by a consistency configuration is normally distributed. The probability $P[X < Y]$, with X and Y being random variables from two different normal distributions, can be calculated as follows [Coo]:

$$\begin{aligned}
P[X < Y] &= 1 - P[X > Y] \\
P[X > Y] &= \Psi\left(\frac{\mu_x - \mu_y}{\sqrt{\sigma_x^2 + \sigma_y^2}}\right)
\end{aligned} \tag{5.16}$$

In the Equation above, Ψ denotes the Cumulative Distribution Function (CDF) of a normal distribution.

5.1.5 Consistency Mixes

New correctness models can be built as combinations of existing models [FS12, IH12, KHAK09, YV02]. For example, different transactions or transaction classes can be executed with different consistency models and/or protocols resulting in a *consistency mix*.

The concept of consistency mixes has been around for a while and is based on the observation that the more inconsistencies are to be prevented the higher the overhead for transactions [Fek05]. Existing databases, such as Oracle and MySQL, have ever since provided a range of consistency models, also known as isolation levels. Application developers can assign the most suitable isolation level to transactions with the goal of finding the right balance between correctness, availability, performance and costs.

The following are possible semantics in case of consistency mixes.

1. The resulting correctness guarantees correspond to that of the strongest model in the mix.

2. The resulting correctness guarantees correspond to that of the weakest model in the mix.

3. The mix results in a new correctness model that needs to be formally defined together with the possible inconsistencies.

Independently on the semantics, it is desirable to define the mix-ability of consistency models and protocols. This would allow C^3 to consider only those consistency configurations that are able to achieve the desired correctness. Moreover, if the mix results in a new correctness model, then C^3 can consider only configurations, which lead to an understandable semantics. In theory, models can be mixed in any arbitrary way, which may lead to a correctness that applications cannot reason about. This is however not the goal of C^3, which does not consider any correctness below that provided by the EC consistency model.

There is a growing research interest in providing such models. For example, in [Fek99] the authors analyze possibilities to prove serializability when all transactions use SI, i.e., provide correctness stronger than the model used by transactions. Along the lines of that work, in [SW00] a set of criteria is defined with the goal of ensuring serializability in federated databases when the sites locally use different models or protocols. The work in [Fek05] provides a theory allowing developers to assign concrete isolation levels to transactions so that the overall correctness guarantees correspond to a serializable execution by considering S2PL and lock-based implementation of SI.

5.1.6 Handling of Multi-Class Transaction Workloads in C^3

In C^3 different inconsistencies may incur different inconsistency cost (*incCost*). Applications can annotate transactions with the *incCost*, and C^3 will cluster transactions in so-called *classes*. A class subsumes all transactions that incur the same inconsistency cost. For example, in an online shop, transactions that insert a new item (e.g., book) and those that read item details, incur a different inconsistency cost than buy and payment transactions. Clustering transactions in different classes allows C^3 to determine the most suitable consistency configuration for each of the classes separately. This does not require any change to the cost and configuration model. It just a matter of clustering the access patterns from the workload. C^3 assumes that each access pattern can exist in exactly one class.

Handling consistency on per-class basis may lead to a situation in which different consistency configurations apply for each of the classes. In that case, applications would observe different correctness for each of the classes. However, usually, the classes are not free of conflicts, i.e., the same objects may be accessed by conflicting actions of transactions belonging to different classes. In that case, the classes running weak consistency models would influence the correctness of those classes that are executed with stronger models. Such an influence can lead to inconsistency costs generated by transactions that run with consistency models, which do not permit any inconsistencies.

C^3 avoids such situations by detecting conflicts between transactions of different classes. In the case of (w/w) conflicts, transactions belonging to different classes will be outsourced to a new class, and the highest inconsistency cost (*incCost*) will be assigned to all transactions of which the new class consists. Its consistency configuration will be then determined based on the usual cost and configuration model. Clearly, in the worst case, all transaction may end up in the same single class, which corresponds to the default C^3 behavior.

5.1.7 Adaptive Behavior of C^3

Algorithm 3: The Algorithm for determining the most optimal consistency configuration. The *determineConfiguration* function is invoked by Algorithm 1.

Input: $wload^{p_{i-1}}$,$\widehat{wload^{p_i}}$, $CL_{current}^{p_{i-1}}$
Output: Configuration for the next period

Function *determineConfiguration* **is**
- maxBenefitConf $\leftarrow CL_{current}^{p_{i-1}}$;
- maxBenefit $\leftarrow 0$;
- totalCosts $\leftarrow$ `calculateTotalCosts(`$\widehat{wload^{p_i}}$,$CL_{current}^{p_{i-1}}$`)` ;
- **foreach** $CL^{p_i} \in \{\mathbb{C} \setminus CL_{current}^{p_{i-1}}\}$ **do**
  - tmpTotalCosts $\leftarrow$ `calculateTotalCosts(`$\widehat{wload^{p_i}}$, CL^{p_i}`)` ;
  - **if** tmpTotalCosts > totalCosts **then**
    - **continue**;
  - tmpTCosts $\leftarrow$ `calculateTransCosts(`$wload^{p_{i-1}}$, $\widehat{wload^{p_i}}$, $CL_{current}^{p_{i-1}}$, CL^{p_i}`)` ;
  - tmpBenefit $\leftarrow$ totalCosts- tmpTotalCosts- tmpTCosts;
  - **if** tmpBenefit > maxBenefit **then**
    - maxBenefit $\leftarrow$ tmpBenefit;
    - maxBenefitConf $\leftarrow CL^{p_i}$;
- `/* The adaptive threshold is checked by the caller of the` *determineConfiguration* `function. */`
- **return** maxBenefitConf;

In what follows we will provide a step-by-step description of the adaptive behavior of C^3 based on the scenario depicted in Figure 4.10. The choice of the most optimal configuration is described in Algorithm 3.

1. In the first step, before p_{i-1} is finished, the workload prediction for p_i is initiated (Algorithm 1).

2. In the second step, the predicted workload $\widehat{wload^{p_i}}$ is used as a basis for calculating the consistency and inconsistency costs for all available configurations based on the Equations 5.1, 5.2, and 5.3.

3. In the third step, the expected gain for the predicted workload will be calculated by comparing the expected costs of the current configuration to that of all other configurations (Equation 5.11).

4. In the forth step, the costs are calculated for each possible transition from the current configuration to any other configuration (Equation 5.12).

5. The configuration that has the highest expected benefit (Equations 5.14 and 5.15) is chosen as the active configuration for period p_i.

6. If the new configuration is not the same as the old one, then a reconfiguration is initiated (step five), which will execute all necessary steps required by the new configuration.

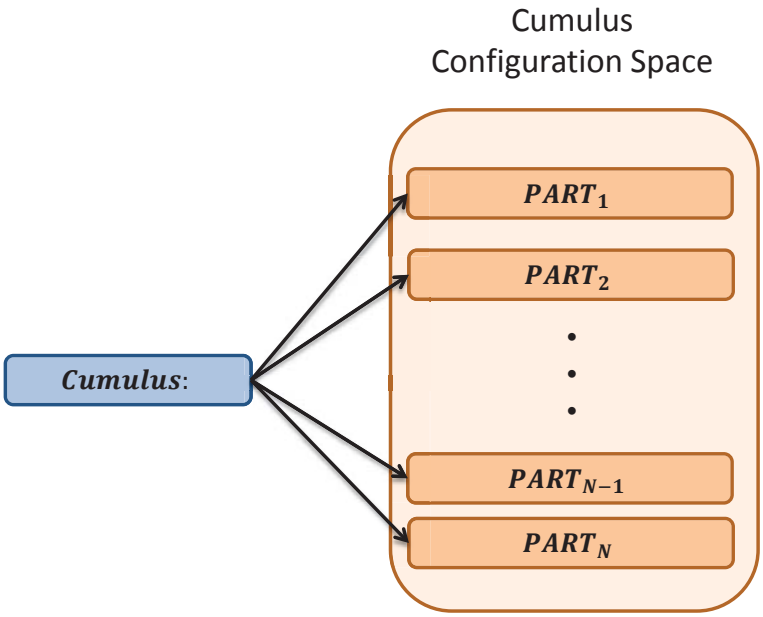


Figure 5.8: Configuration space of Cumulus.

5.2 Cumulus: A Cost and Workload-Driven Data Partitioning Protocol

Distributed transactions are expensive in terms of both performance and monetary cost. This overhead can be considerable especially if strong consistency such as 1SR is required (see Section 3.4), as in that case distributed coordination over the network is necessary. Data partitioning is an approach that considers the collocation of objects accessed together so that, in the best case, distributed transactions are completely avoided. The simplest approach to avoid distributed transactions is to collocate all objects to the same site. However, that would not be the most optimal approach in terms of load distribution.

Cumulus is a cost and workload-driven data partitioning protocol that jointly addresses both the reduction of distributed transactions and load distribution. It is able, given the expected application workload (Section 4.2.3), to derive significant access patterns from that workload at runtime and partition the data so that the number of distributed transactions is reduced. Cumulus uses the horizontal partitioning approach based on the object id (o_{id}) as defined in Section 3.2.1.

Cumulus is tailored to applications that demand 1SR consistency and assumes that data do not need to be replicated, and that latency is the main application concern (see Section 4.3). Approaches such as the one defined in [SMA+14] can be used to handle the replication of partitions if applications demand certain availability guarantees.

The Cumulus configuration space ($\mathbb{C}_{cumulus}$) is defined by the *partition sets*:

$$\mathbb{C}_{cumulus} = \{PART_1, \ldots, PART_N\}$$

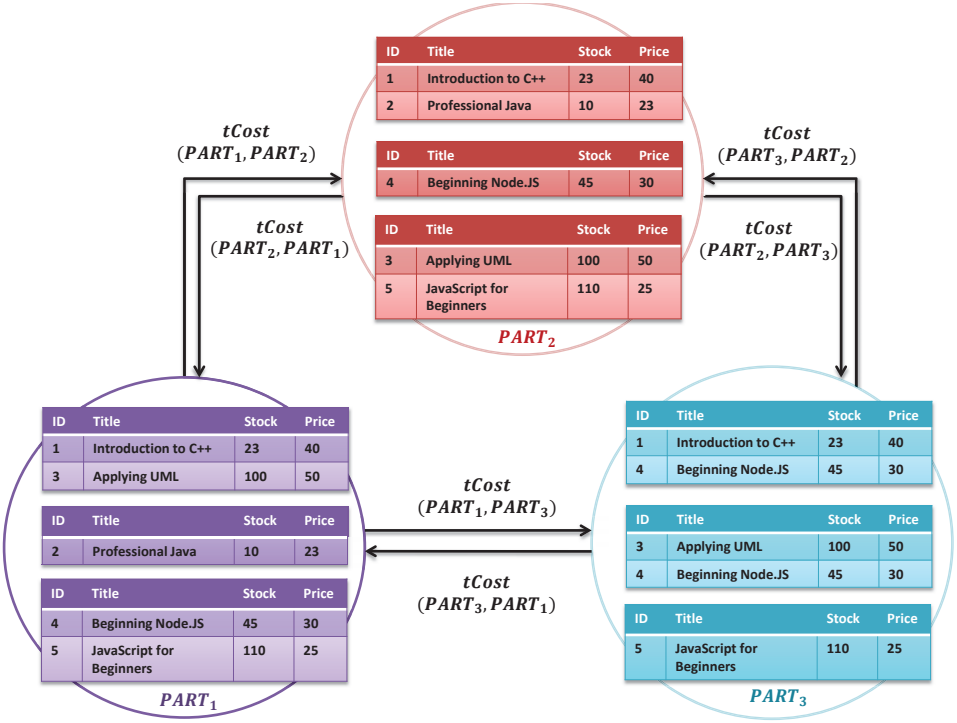


Figure 5.9: A sample configuration space consisting of three partition sets: $\mathbb{C}_{cumulus} = \{PART_1, PART_2, PART_3\}$. Each partition set consists of one or more partitions as defined in Section 3.2.1 (see Definition 3.26).

A partition set ($PART$) is a unique mapping of objects to partitions based on the object id (Section 3.2.1). If we assume that there is one-to-one mapping of partitions to sites, then $|PART| = |S|$. The size of the configuration space of Cumulus is determined by the number of unique mappings of objects to partitions. Given $|LO|$ objects, there are $|PART|^{|LO|-1} - 1$ ways to distribute the objects to partitions, so that each partition gets at least one object [Qia]. For example, given the objects o_1 and o_2 and the partitions $part_1$ and $part_2$ there is one possible way to distribute the objects, namely $\langle part_1 : o_1\rangle, \langle part_2 : o_2\rangle$;$\langle part_1 : o_1\rangle, \langle part_2 : o_2\rangle \equiv \langle part_1 : o_2\rangle, \langle part_2 : o_1\rangle$, as they generate the same number of distributed transactions.

The goal of Cumulus is to choose that $PART$ from $\mathbb{C}_{cumulus}$ that minimizes the application costs (Figures 5.8 and 5.9). The adaptability property allows Cumulus to continuously readjust the partitions, i.e., switch to another partition set, if the workload shifts.

The Cumulus results have been published in [FMS15].

5.2.1 Data Partitioning

Distributed transactions are expensive and should be avoided [CZJM10]. In a DDBS with replication, the percentage of distributed transactions is determined by the percentage of update transactions in the workload. The goal of data partitioning is to organize the data in disjoint partitions that reside in different sites so that, in the best case,

each partition can serve both read-only and update transactions locally without any coordination with other partitions. In what follows, we briefly characterize different requirements towards data partitioning protocols.

Distributed Transactions and Load Distribution Distributed transactions are expensive due to the necessary commit coordination [CZJM10]. A partitioning protocol should minimize the number of – or in the best case completely avoid– distributed transactions. In addition to distributed transactions, a data partitioning protocol, should also avoid so-called *hotspots* which are objects or partitions that are very popular. This can be done by creating partitions, which avoid bottlenecks. Graph-based algorithms [CZJM10], for instance, can be used to target both goals, i.e., the minimization of distributed transactions and load distribution.

Rigid vs. Elastic Partitioning Elasticity is a crucial requirement for applications deployed in the Cloud [Aba09]. It denotes the ability of a system to provision or deprovision resources based on the application load. In the context of partition protocols, the elasticity defines the ability to deploy new sites in case the available resources can not handle the load. These additional sites can be used to create replicas for overloaded partitions and use them for distributing the load, or a reconfiguration can be initiated by incorporating new sites into the distribution process of partitions to sites. Both approaches can be used to address the hotspot challenge, to avoid that a partition is more frequently accessed by transactions than others and become a bottleneck. However, by replicating hotspots and distributing the load between sites, expensive distributed transactions are introduced. Even though only the hotspot data objects are involved in these distributed transactions, the performance of this approach may still degrade to the performance of a fully replicated system due to the popularity of the hotspots. The de-provisioning (un-deployment of sites) in case the load decreases again is equally important to the provisioning, as unneeded resources generate unnecessary costs.

Elastic protocols, in contrast to rigid protocols, are able to provision and deprovising sites based on the application workload by also considering the incurring monetary costs [SMA+14].

Static vs. Adaptive Partitioning As described in Example 4.3, the workload of an application may shift as a consequence of changed user behavior. In that case, the previously deployed partitions may suddenly become unsuitable and lead to a high ratio of distributed transactions. Static protocols are not able to adapt at runtime, and would require the system must be taken off-line in order to reconfigure it. However, this is in a sharp contrast to the high availability and always-on requirements of applications deployed in the Cloud [ADE12].

Adaptive protocols are able to reconfigure the DBS at runtime. However, care must be taken with regards to the reconfiguration cost, but also with regards to consistency (safety) as the reconfiguration is usually a distributed activity.

Manual vs. Automatic Partitioning Manual protocols require the involvement of a (human) expert, while automatic approaches can re-partition the data without any expert involvement. Hybrid protocols can propose partitions that can be further refined by an expert, or refine partitions recommended by an expert.

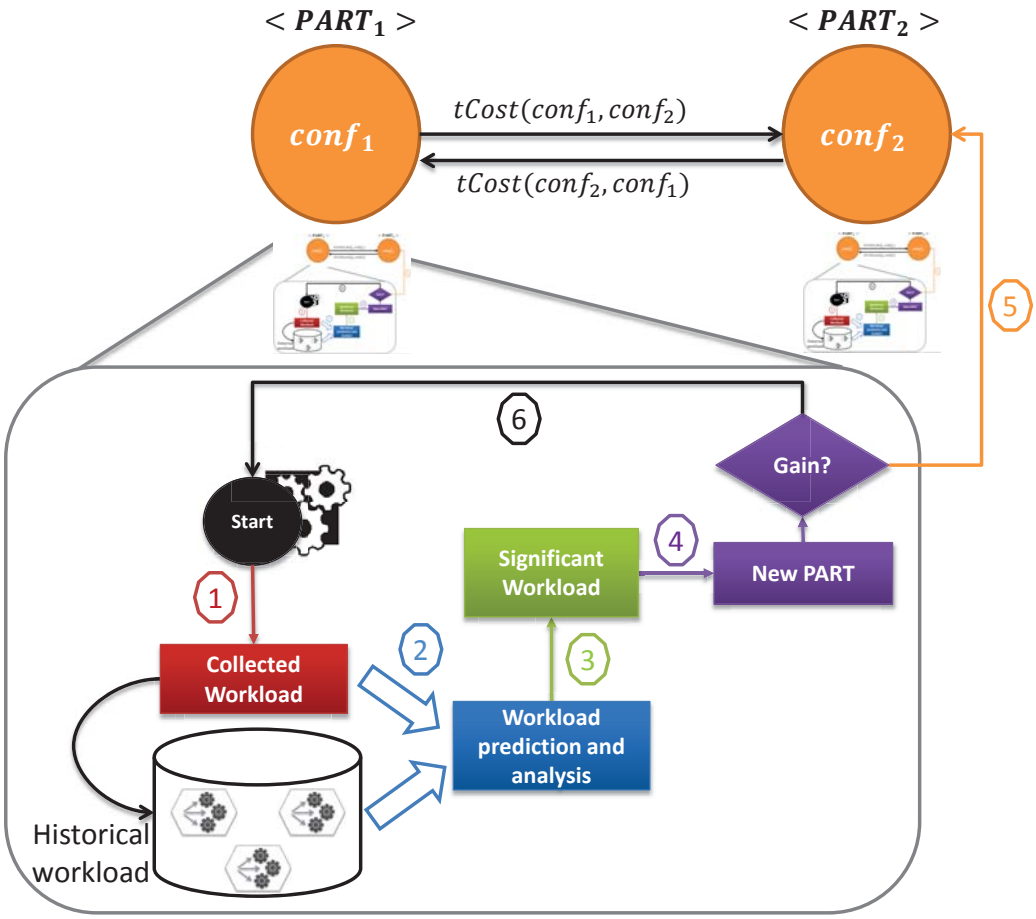


Figure 5.10: Cumulus partitioning workflow.

Routing Transparency vs. Manual Site Selection Compared to a fully replicated system, the location of data in a partitioned system is mandatory information needed to route transactions to those sites that contain the required data. Protocols that provide routing transparency decouple the client from the data location and give the system the flexibility to reconfiguring the DDBS without affecting applications.

5.2.2 The Cumulus Data Partitioning Approach

Cumulus targets three main aspects that considerably impact the quality of the generated partitions, the partitioning overhead, and the correctness of the system behavior:

1. Cumulus incorporates a workload analysis approach that aims at anticipating future access patterns for the generation of partitions to match best the expected workload and thus reduce or completely avoid expensive distributed transactions.

2. Cumulus is based on a cost model, which targets the optimization of application costs, and configuration model which ensures that a reconfiguration is conducted only if it leads to a benefit, i.e., the gain outweighs the reconfiguration costs (Equation (4.5)).

3. Cumulus implements an on-the-fly and on-demand reconfiguration approach that compared to stop-and-copy approaches [TMS+14], considerably reduces the system unavailability (interruption duration). It exploits locking and 2PC to ensure correctness (safety) even in case of failures and provides full routing transparency to applications.

The partitioning workflow of Cumulus is depicted in Figure 5.10 and is defined as follows. Once there is a significant amount of current workload collected, the workload prediction and analysis activity is initiated. The goal of the workload prediction is to anticipate the expected workload in the future, whereas the workload analysis determines the most significant subset of access patterns in the workload that should determine the partitions. It is difficult, if not impossible, to generate partitions that satisfy the entire workload. Moreover, considering all access patterns, depending on the workload size, may lead to considerable partitioning overhead, which is not inline with an online protocol. The Cumulus' strategy is based on the idea of tailoring the partitions to those access patterns that will occur with high frequency, as optimizing for infrequent (*noisy*) patterns generates low or no benefit at all.

A new partition set is proposed based on the most significant subset of access patterns. Based on the cost model which finds the right balance between reconfiguration cost and the gain of the new configuration, Cumulus will decide if the new partitions are to be applied. If so, it will initiate the transition to the new configuration using the on-demand and on-the-fly approach, which is driven by user transactions (i.e., only objects that are accessed will be redistributed on the fly) such that the system interruption remains low.

Once a decision has been reached, the entire workflow is restarted independently on the current configuration. At workflow restart, the previously collected workload is added to the set of historical workloads, the current workload is reset, and the workload monitoring and collection is resumed. In what follows, we will describe in greater detail the different steps of the workflow.

5.2.3 Workload Prediction and Analysis

In the first step of the workflow, the workload for the next period p_i will be predicted using the EMA model on the basis of the current workload that considers the entire database. Notice that Cumulus slightly deviates with regards to the workload definition in Section 4.2, as it considers only objects and ignores the actions of the access patterns. Equation (5.2.3) defines the equality of access patterns in Cumulus.

$$ap^q = ap^u \iff (\forall o \, [o \in ap^q \Leftrightarrow o \in ap^u])$$

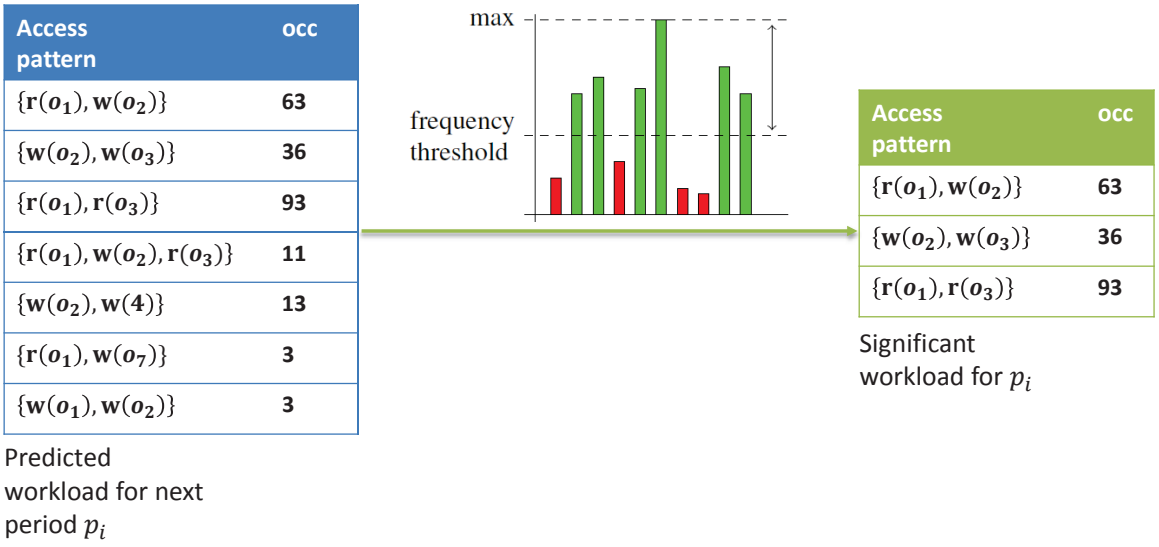


Figure 5.11: Workload prediction and analysis.

Example 5.5 (Equal and Distinct Access Patterns in Cumulus)

$$
\begin{aligned}
ap^1 &= \{r^{1,1}(o_3), r^{1,2}(o_5), r^{1,3}(o_7)\} \\
ap^2 &= \{r^{2,1}(o_3), r^{2,2}(o_5), r^{2,3}(o_7)\} \\
ap^3 &= \{r^{3,1}(o_3), w^{3,2}(o_5), r^{3,3}(o_7)\} \\
ap^4 &= \{r^{4,1}(o_3)\}
\end{aligned}
$$

$$
\begin{aligned}
ap^1 &= ap^2 = ap^3 \\
ap^1 &\neq ap^4 \wedge ap^2 \neq ap^4 \wedge ap^3 \neq ap^4
\end{aligned}
$$

As depicted in Example 5.5, in contrast to the definition in Section 4.2, two access patterns are equal if they access the same set of objects independently on the operations that act on the objects. However, since distributed transactions that include writes imply the usage of 2PC, read and write actions do not generate the same costs. Considering the operations in addition to the objects would allow for a more fine-grained optimization at the price of increased graph size, as in that case the graph nodes represent actions and not objects. Two actions $r(o)$ and $w(o)$ would be represented by two graph nodes, instead of one in Cumulus. In the worst case the size of the graph increases by a factor of two, i.e., from $|wload.LO|$ nodes to $2 \cdot |wload.LO|$, with $wload.LO$ denoting the set of objects in the workload. The worst case incurs if each object is accessed by a write action, as we assume that a read action always precedes a write action. Therefore, Cumulus neglects the operations to reduce the partitioning overhead, which correlates with graph size.

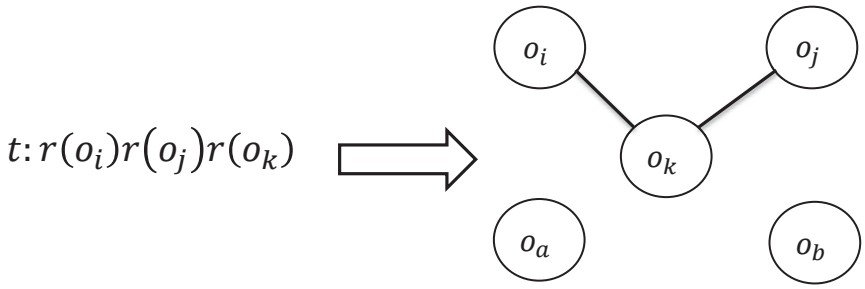
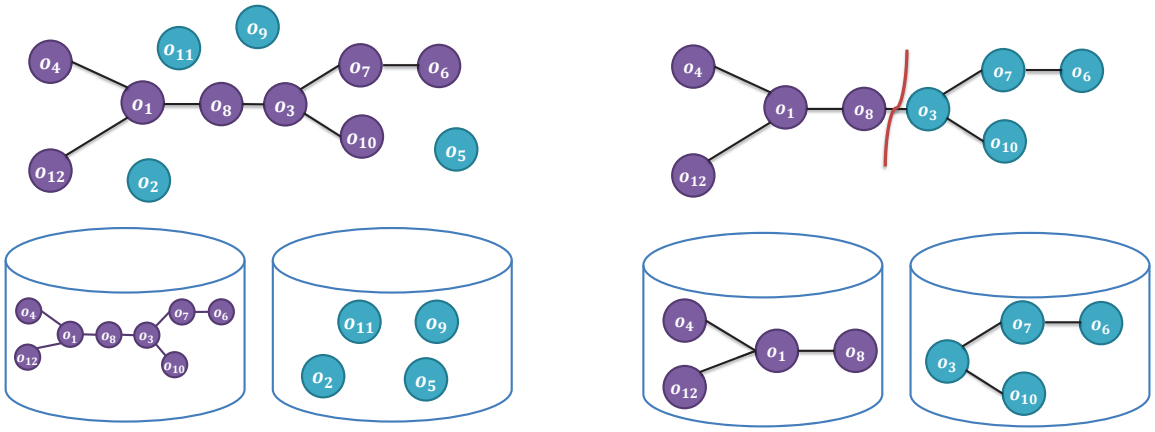


Figure 5.12: Workload graph.

The smoothing property of EMA is crucial for the adaptive behavior of Cumulus, as high frequency access patterns that were added as part of previously triggered re-partitioning activities would remain indefinitely in the workload if no smoothing was applied. Over time, they might superimpose and dominate the current and expected access patterns and would behave similar to noisy transactions, preventing the system to adapt to shifting access patterns. A naïve approach would be to construct the workload from scratch each time a partitioning is triggered. However, this memory-less approach would disallow the system to recognize long-term access patterns, and would lead to a highly unstable system that will steadily re-partition in reaction to short-term access pattern changes.

In the second step, the workload analysis is initiated, which applies a high-pass filter to the predicted frequencies with the goal of removing infrequent (noisy) patterns from the workload. The filter only preserves access patterns that have a predicted frequency above a certain threshold (Figure 5.11). Each access pattern from the predicted workload represents a constraint to the problem of determining the optimal partition configuration. On one hand, the more constraints are available, the more difficult it is to satisfy all of them leading to a sub-optimal partition configuration. On the other hand, optimizing for noisy transactions does not generate any benefit for future transactions. The choice of the applied filter can alter the partitioning behavior significantly. The more aggressive the high pass filter is tuned, the more stable already established patterns will be. On the other hand, new patterns will need a significant frequency before they are recognized. It is thus advised to tune the filter according to the expected access pattern, i.e., a high threshold for stable patterns and a lower threshold for volatile access patterns.

The resulting significant workload is maintained as a graph, similar to the approach presented in [CZJM10]. The graph consists of nodes that denote data objects, and edges between o_i and o_j if both objects are accessed by the same transaction (Figure 5.12). Edge weights improve the partition step by giving a higher weight to re-occurring transaction edges. The problem of finding the optimal partition configuration is thus equivalent to the problem of finding optimal graph partitions, and this can be done by graph partitioning libraries such as Metis [KK98]. The configuration space is exploited by the graph algorithm, i.e., the configurations are implicit in that case. The algorithm will



(a) Partitions leading to a site becoming a bottleneck

(b) Partitions that avoid bottlenecks

Figure 5.13: The impact of considering all database objects vs. workload objects only to the quality of the generated partitions.

simply propose the best configuration[1], and Cumulus will decide based on its cost and configuration model if there is a benefit in moving to that proposed configuration.

Since Cumulus takes into account only objects from the significant workload when determining the partitions, objects not available in the workload are stripped to the available sites based on a round-robin approach. This is done to avoid problems that would occur when considering all database objects when constructing the workload graph. If the set of workload objects is restricted to a subset of the entire database objects, then the resulting graph will contain a low edge to node ratio and many nodes will have no connections at all. In this case, the graph partitioning may lead to all graph nodes corresponding to data objects being located at one site and the rest will be randomly distributed to the other sites. This approach will lead to a low ratio of distributed transactions. However, it is likely to create hotspots (Figure 5.13).

5.2.4 Cost Model

As already described, the task of Cumulus is to choose those partitions that reduce the number of distributed transactions with the goal of satisfying application requirements towards performance. Although not the primary concern of Cumulus, the activities for coordinating a distributed transaction can be mapped to monetary costs. In that case, Cumulus would target the reduction of monetary costs. In what follows, we will describe the cost model of Cumulus. The list of symbols is summarized in Table 5.2.

The total costs of a partition configuration are defined by the operational cost, as no penalty cost can occur in Cumulus. The reason is that Cumulus targets the optimization of an implicit requirement, such as the performance impact of distributed transactions.

$$totalCost(wload^{p_i}, PART) = opCost(wload^{p_i}, PART) \tag{5.17}$$

[1]The problem of finding an optimal graph partition is NP-complete. Thus, the "best" corresponds to an approximate solution based on the heuristics used by the graph algorithm.

Symbol	Description
$totalCost(wload, PART)$	Denotes the total costs generated by the partition configuration $PART \in \mathbb{C}_{cumulus}$ given the specified workload $wload$.
$opCost(wload, PART)$	Denotes operational costs generated by $PART$ given $wload$.
$\widehat{gain}(\widehat{wload^{p_i}}, PART^{p_{i-1}}_{current}, PART^{p_i}_{new}$	Denotes the expected saving in costs when the predicted workload $\widehat{wload^{p_i}}$ is executed with $PART^{p_i}_{new}$ compared to the costs generated by $PART^{p_{i-1}}_{current}$.
$tCost(wload^{p_{i-1}}, PART^{p_{i-1}}_{current}, PART^{p_i}_{new})$	Denotes the costs of a transition from $PART^{p_{i-1}}_{current}$ to $PART^{p_i}_{new}$ given the executed $wload^{p_{i-1}}$ with $PART^{p_{i-1}}_{current}$.
$DTxns(wload, PART)$	Denotes the set of distributed transactions given the workload $wload$ and partition configuration $PART$.
$costDTxn$	Denotes the cost of a distributed transaction.
$dCost(wload, PART)$	Denotes the costs of distributed transactions given the workload $wload$ and partition configuration $PART$.
$dCost(t, PART)$	Denotes the cost of distributed transaction t given configuration $PART$.
$dDeg(t, PART)$	Denotes the distribution degree of transaction t given $PART$.
$\#accPartitions(t, PART)$	Denotes the number of accessed partitions by t given $PART$.
$maxDDeg(PART)$	Denotes the maximum distribution degree given $PART$.
$tCost(wload^{p_{i-1}}, PART^{p_{i-1}}_{current}, PART^{p_i}_{new})$	Denotes the transition costs given the workload $wload^{p_{i-1}}$, current configuration $PART^{p_{i-1}}_{current}$, and the new configuration $PART^{p_i}_{new}$.
otM	Denotes the set of objects to be relocated as consequence of a transition to a new configuration.
$relocCost$	Denotes the cost for the relocation of a single object.
$\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, PART^{p_{i-1}}_{current}, PART^{p_i}_{new})$	Denotes the expected benefit when the predicted workload $\widehat{wload^{p_i}}$ is executed with $PART^{p_i}_{new}$ compared to that of $PART^{p_{i-1}}_{current}$.

Table 5.2: Cumulus symbols and notations.

The operational cost are determined by the number of distributed transactions (Equation (5.18)):

$$\begin{aligned} opCost(wload^{p_i}, PART) &= dCost(wload^{p_i}, PART) \\ dCost(wload^{p_i}, PART) &= \frac{\sum_t^{t \in DTxns} dCost(t, dDeg(t, PART))}{\sum_t^{t \in wload^{p_i}} dCost(t, maxDDeg(PART))} \end{aligned} \tag{5.18}$$

In Equation (5.18), $DTxns$ denotes the set of distributed transactions given the workload $wload^{p_i}$ and configuration $PART$, and $dCost(t, maxDDeg(PART))$ the cost of the distributed transaction t. The denominator of the fraction is used to normalize the cost by simply summing up the cost of all transactions in the workload with the highest possible distribution degree.

Cumulus assumes that distributed transactions need to execute more activities than non-distributed transactions, and are thus more expensive. The cost of a distributed transaction is influenced by and increases in non-linear manner with its distribution degree:

$$\begin{aligned} dCost(t, PART) &= \left(\frac{dDeg(t, PART)}{maxDDeg(PART)}\right)^2 \cdot costDTxn \\ dDeg(t, PART) &= \#accPartitions(t, PART) - 1 \\ maxDDeg(PART) &= |PART| - 1 \text{ with } |PART| > 1 \end{aligned} \tag{5.19}$$

In Equation (5.19), $dDeg(t, PART)$ denotes the distribution degree of t which is determined by the number of accessed partitions $\#accPartitions$, $maxDDeg(PART)$ denotes the maximal possible distribution degree, and $costDTxn$ is fixed cost for a distributed transaction.

5.2.5 Configuration Model

The expected gain of adapting the configuration from the currently active one ($PART^{p_{i-1}}_{current}$) to a new one ($PART^{p_i}_{new}$) given the expected workload $\widehat{wload^{p_i}}$ is defined as follows:

$$\begin{aligned} \widehat{gain}(\widehat{wload^{p_i}}, PART^{p_{i-1}}_{current}, PART^{p_i}_{new}) = \widehat{totalCost}(\widehat{wload^{p_i}}, PART^{p_{i-1}}_{current}) - \\ \widehat{totalCost}(\widehat{wload^{p_i}}, PART^{p_i}_{new}) \end{aligned} \tag{5.20}$$

The gain is measured in terms of reduction in distributed transactions, and is considerably impacted by the quality of the generated partition configuration, i.e., how well the configuration matches the workload, and the lifetime of the configuration (stability). By including a workload analysis based on exponential smoothing and high-pass filtering, Cumulus considerably improves both the quality and the stability of the configuration.

A transition from the current partition configuration to a new one generates costs, which incur from the necessity of relocating objects and are defined as follows:

$$tCost(wload^{p_{i-1}}, PART^{p_{i-1}}_{current}, PART^{p_i}_{new}) = \frac{\sum_{o \in OtM} relocCost(o)}{\sum_{o \in LO} relocCost(o)} \tag{5.21}$$

OtM in Equation (5.21) denotes the set of objects to be relocated given the workload $wload^{p_{i-1}}$, currently active configuration $PART^{p_{i-1}}_{current}$ and the new configuration $PART^{p_i}_{new}$; $relocCost$ denotes the relocation cost for a single object.

A transition to the new configuration $PART_{new}^{p_i}$ generates a benefit if the expected gain outweighs the transition costs to $PART_{new}^{p_i}$:

$$\begin{aligned}\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload}^{p_i}, PART_{current}^{p_{i-1}}, PART_{new}^{p_i}) = \\ \widehat{gain}(\widehat{wload}^{p_i}, PART_{current}^{p_{i-1}}, PART_{new}^{p_i}) - \\ tCost(wload^{p_{i-1}}, PART_{current}^{p_{i-1}}, PART_{new}^{p_i})\end{aligned} \tag{5.22}$$

The new configuration is applied if the expected benefit is above a certain threshold th:

$$\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload}^{p_i}, PART_{current}^{p_{i-1}}, PART_{new}^{p_i}) > th \tag{5.23}$$

In here, the choice of the threshold parameter th is crucial. A high threshold increases the life-cycle of a configuration and may lead to a high loss as there is a high tolerance to distributed transactions. A low threshold leads to frequent reconfigurations and might lead to transition costs that outweigh the gain. In future work, we plan to incorporate a machine learning-based approach into Cumulus to determine the optimal value of th.

5.2.6 Handling of Inserts and Deletes

Insert and delete actions[2] need a special treatment by a partitioning protocol. In Cumulus, delete operations are sent to the site containing the object to be deleted. A delete transaction is split into sub-transactions containing a batch of deletes for each site containing objects to be deleted. The atomicity guarantees are preserved for a transaction by using 2PC for all sub-transactions. If a delete transaction commits, the routing information needs to be updated. A delete transaction executed during a reconfiguration will additionally remove the objects from the set of objects to be relocated. Insert operations impose a challenge with regards to the most optimal location (site) of the new objects, as no access patterns are known at the insert time. The goals of the partitioning, namely the collocation of objects and load distribution, should also apply to newly inserted objects. Cumulus does not follow the collocation goal but is mainly focused on evenly distributing inserted objects across sites. It postpones object collocation to the next reconfiguration event. The even distribution of inserted objects to sites can be implemented using a round-robin approach. However, this requires a centralized instance for maintaining the next site for the insert. In Cumulus, each site can locally decide on where to next insert objects, by choosing a site from all available sites, all with equal probability.

5.2.7 Adaptive Behavior of Cumulus

In what follows we will provide a step-by-step description of the adaptive behavior of Cumulus based on the scenario depicted in Figure 5.10 (see also Figure 4.10). The mapping of the application workload to a graph and the choice of the most optimal partition configuration are described in Algorithm 4.

[2] An insert action is a write action that inserts a new object, whereas a delete action is a write action that sets an existing object to NULL.

1. In the first and second step, before p_{i-1} is finished, the workload prediction for p_i is initiated (Algorithm 1).

2. The predicted workload $\widehat{wload}^{p_i}$ from the second step is used as a basis for determining the significant access patterns (step three). The resulting significant workload is used for the generation of the workload graph as described in Section 5.2.3. The problem of finding the most optimal partition configuration is mapped to a graph partitioning problem [CZJM10].

3. In the fourth step, a new configuration is proposed. The expected gain of the new configuration will be calculated by comparing the expected costs of the current configuration to that of the proposed configuration by the graph partitioning algorithm (Equation 5.20).

4. If the proposed configuration generates a benefit (Equation 5.23), by also considering the transition costs (Equation 5.21), then a reconfiguration is initiated (step five), which will execute all necessary steps required by the new configuration.

Algorithm 4: The Algorithm for determining the most optimal configuration. The *determineConfiguration* function is invoked by Algorithm 1.

Input: $wload^{p_{i-1}}$,$\widehat{wload^{p_i}}$, $PART^{p_{i-1}}_{current}$
Output: Configuration for the next period

Function *determineConfiguration* **is**
- $Graph : g \leftarrow \{\}$;
- maxBenefitConf $\leftarrow PART^{p_{i-1}}_{current}$;
- `/* Calculate the total costs for the predicted workload based on the current configuration. */`
- totalCosts $\leftarrow$ `calculateTotalCosts(`$\widehat{wload^{p_i}}$,$PART^{p_{i-1}}_{current}$`)` ;
- **foreach** ap $\in \widehat{wload^{p_i}}$ **do**
  - **foreach** $o \in ap.LO$ **do**
    - **if** $\neg$ g`.contains(`o`)` **then**
      - g`.addNode(`o`)`
    - $ap.LO \leftarrow ap.LO \setminus o$;
    - `/* Add edges from` o `to all other objects of access pattern` ap`. */`
    - g`.addEdge(`o,$ap.LO$`)` ;
- `/* The graph partition algorithm determines the optimal partition for the workload */`
- $PART^{p_i}_{new} \leftarrow$ `graphPartition(`g`)` ;
- tmpTotalCosts $\leftarrow$ `calculateTotalCosts(`$\widehat{wload^{p_i}}$, $PART^{p_i}_{new}$`)` ;
- **if** tmpTotalCosts < totalCosts **then**
  - tmpTCosts $\leftarrow$ `calculateTransCosts(`$wload^{p_{i-1}}$, $\widehat{wload^{p_i}}$, $PART^{p_{i-1}}_{current}$, $PART^{p_i}_{new}$`)` ;
  - tmpBenefit $\leftarrow$ totalCosts- tmpTotalCosts- tmpTCosts;
  - **if** tmpBenefit > th **then**
    - maxBenefitConf $\leftarrow PART^{p_i}_{new}$;
- **return** maxBenefitConf;

5.3 QuAD: Cost and Workload-Driven Quorum Protocol

QuAD is a cost and workload-driven quorum protocol that constructs the site quorums by considering the load of the sites and their network distance. It is based on the observation that not only the size of the quorums determines the overall performance, but also the properties of the sites constituting the quorums. The 'weakest' sites of a quorum are the limiting factor to performance, as they need to be accessed by transaction for reading (*read-path*) and writing (*commit-path*) the data. Avoiding such weak sites from the read and commit paths of transactions decreases their overhead and increases the system performance [SFS15].

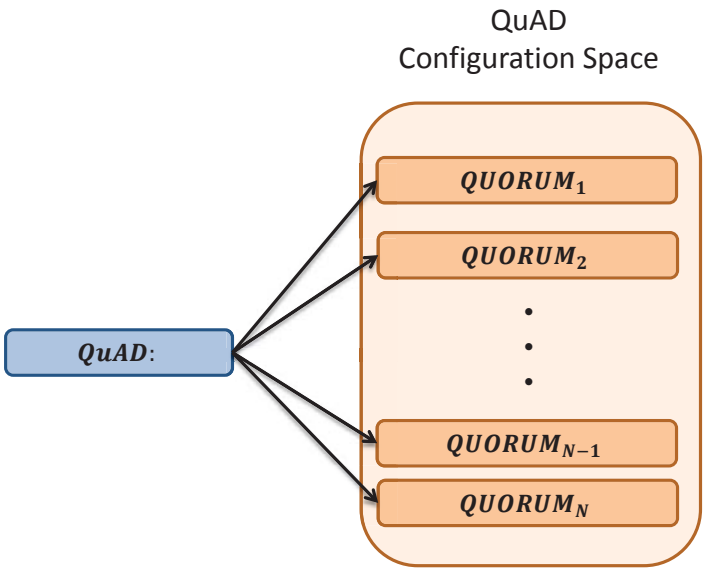


Figure 5.14: Configuration space of QuAD.

The configuration space of QuAD is determined by the quorum definition for each of the sites consisting the DDBS (Figure 5.14):

$$\mathbb{C}_{QuAD} = \{QUORUM_1, QUORUM_2, \ldots, QUORUM_N\}$$

A concrete QuAD configuration ($QUORUM$) is determined by the definition of the quorums for each of the sites (Figure 5.15):

$$QUORUM : \left\{quorums(s_1), quorums(s_2), \ldots, quorums(s_{|S|})\right\}$$

The set of quorums of a site s is determined by its read and write quorum:

$$quorums(s) : \{rq(s), wq(s)\}$$
$$rq(s),\ wq(s) \subseteq S$$

QuAD can adjust the quorum configuration at runtime if the properties of the sites change, in case of site failures or if new sites join the system. The later might be, for example, the consequence of an increased availability level. As defined in Section 4.3, QuAD is tailored to applications that demand 1SR consistency on top a of a fully replicated DDBS.

The QuAD results have been published in [FS13] and [SFS15].

5.3.1 Quorum-based Protocols

As described in Section 3.3.3, the goal of quorum protocols is to reduce the overhead for update transactions as only a subset of sites is eagerly committed. Committing a subset of sites, instead of all of them, reduces the number of messages in the system and the load generated at the sites. However, to guarantee 1SR consistency, reads must

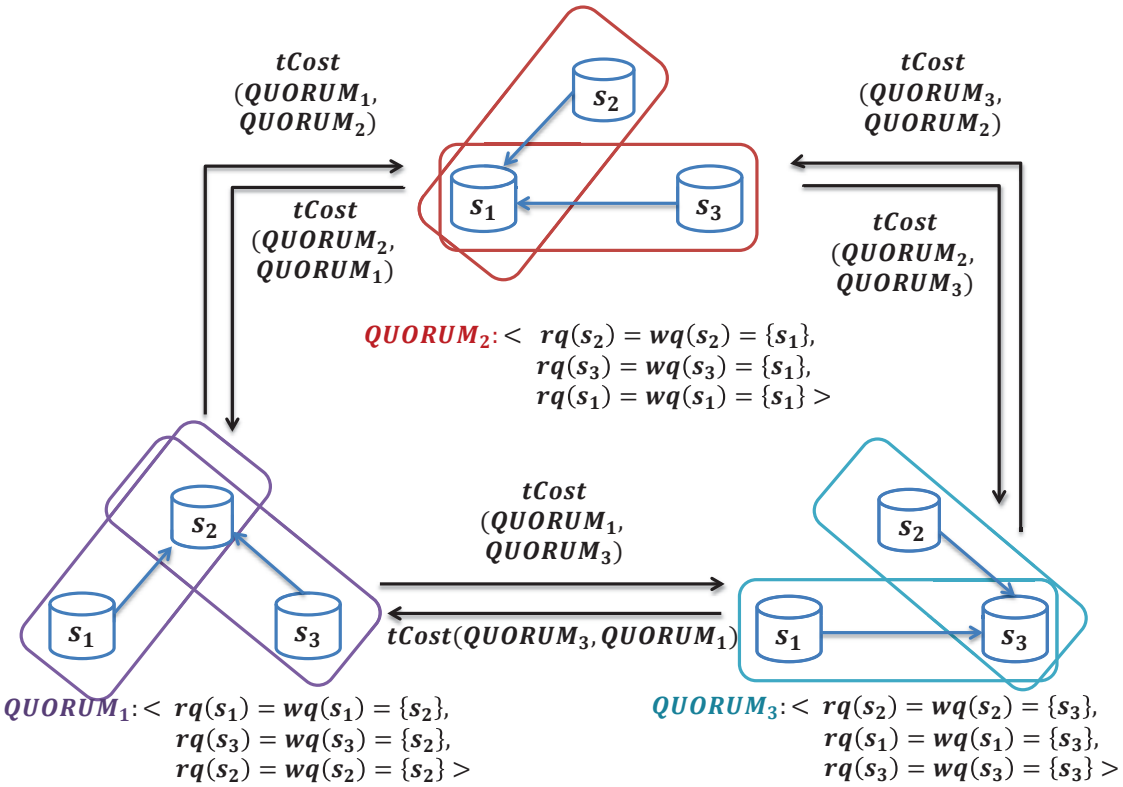
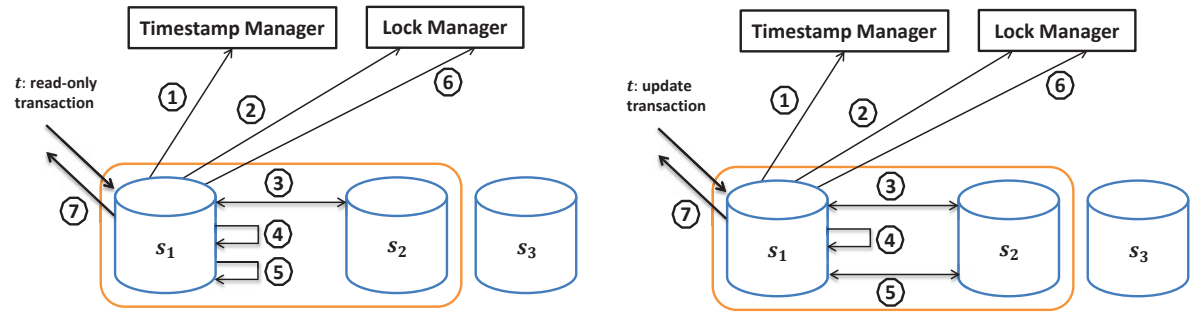


Figure 5.15: A sample configuration space consisting of three quorum configurations: $\mathbb{C}_{QuAD} = \{QUORUM_1, QUORUM_2, QUORUM_3\}$. A quorum configuration is defined by the quorums of each of the three sites. An arrow from one site to another defines an *includes-in-quorum* relationship.

also access a subset of sites and this, in contrast to the ROWAA approach, increases the overhead for the reads [KJP10, JPAK03].

The size of the quorums, i.e., the number of sites accessed by read-only and update transactions, is not the only determining factor for the overall performance. The properties of the sites consisting the quorums, such as their load, capacity, costs and the distance between them, are crucial for the satisfying the application requirements towards performance.

The life-cycle of transactions in quorum-based RPs that provide 1SR consistency is depicted in Figures 5.16a and 5.16b. In the first step, each transaction will acquire a unique timestamp from the `TimestampManager`. The timestamp is crucial in order to guarantee transactions access to the most recent data. In the second step, transactions will acquire (shared or exclusive) locks for all objects accessed according to the S2PL (see Section 3.1.5). Read-only transactions will access the read quorum by reading the values of the objects at every site that is part of the quorum (step 3), and construct the final result by taking those values that have the highest commit timestamp (step 4). At commit (step 5), a read-only transaction will release the locks (step 6) and send the response to the client (step 7).



(a) Read-only transaction

(b) Update transaction

Figure 5.16: Lifecycle of transactions in quorum-based protocols.

Update transactions must also first update the local versions at the executing site (step 3) before the processing of the transaction can be started (step 4). During the commit (step 5) the new values are first propagated to all sites consisting the write quorum of the coordinating site (s_1). After that, the locks are released(step 6), and the response is delivered to the client (step 7).

5.3.2 The QuAD Approach to Quorum-based Replication

The goal of QuAD is to construct the quorums in such a way so that 'weak' replica sites are avoided from the read and commit path of transactions (step 3 in Figure 5.16a, steps 3 and 5 in Figure 5.16b), where 'weak' can have different meanings, such as slow and distant, but also expensive. QuAD considers the load of the sites and their distance, i.e., RTT, when determining the quorums. The reduction of the read and commit overhead has a considerable impact on the overall performance [SFS15]. Additionally, QuAD seeks a possibly balanced assignment of sites to quorums, since if some sites are frequently included in the quorums, they become a bottleneck [SFS15].

QuAD is an adaptive protocol that is able to react to changes in the system, such as increased load at sites, new sites joining the system or site failures, and consequently, adapt its quorums to reflect these changes. In what follows, we will provide the details of the quorum construction in QuAD. The list of symbols is summarized in Table 5.3.

Construction of Quorums in QuAD

The QuAD quorum construction is motivated by the κ-centers problem [BKP93, KS00]: given a set of n cities and a distance between them, the goal is to build κ warehouses so that the maximum distance of a city to a warehouse is minimized.

The same analogy can be used to describe the QuAD strategy. QuAD chooses the κ-strongest sites to become *core sites* ($CS \in \mathcal{P}(S)$), and the rest forms the set of *slave* sites ($SL = S \setminus CS$). The strength of a site is determined by its *score*. The quorum construction process is then as follows (Figure 5.17):

Symbol	Description
CS	Denotes the set of core sites.
cs	Denotes a core site: $cs \in CS$.
κ	Denotes the number of core sites: $\lvert CS \rvert = \kappa$.
SL	Denotes the set of slave sites.
sl	Denotes a slave site: $sl \in SL$.
CQ_r	Denotes the core sites that are part of the read quorum of another core site.
CQ_w	Denotes the core sites that are part of the write quorum of another core site.
SQ_r	Denotes the core sites that are part of the read quorum of a slave site.
SQ_w	Denotes the core sites that are part of the write quorum of a slave site.
$crq(cs)$	Denotes the read quorum of the core site cs.
$cwq(cs)$	Denotes the write quorum of the core site cs.
$srq(sl)$	Denotes the read quorum of the slave site sl.
$swq(sl)$	Denotes the write quorum of the slave site sl.
$load(s)$	Denotes the load of the site s.
$rtt(s)$	Denotes the RTT of s to all other sites.
w_{load}	Denotes the weight of the load score.
w_{rtt}	Denotes the weight of the RTT score.
$loadsc(s)$	Denotes the load score of site s.
$rttsc(s)$	Denotes the RTT score of site s.
$totalCost(wload, QUORUM)$	Denotes the total costs generated by the quorum configuration given the specified workload $wload$.
$opCost(wload, QUORUM)$	Denotes operational costs generated by $QUORUM$ given $wload$.
$\widehat{gain}(\widehat{wload^{p_i}}, QUORUM^{p_{i-1}}_{current}, QUORUM^{p_i}_{new}$	Denotes the expected saving in costs when the predicted workload $\widehat{wload^{p_i}}$ is executed with $QUORUM^{p_i}_{new}$ compared to the costs generated by $QUORUM^{p_{i-1}}_{current}$.
$tCost(wload^{p_{i-1}}, QUORUM^{p_{i-1}}_{current}, QUORUM^{p_i}_{new})$	Denotes the costs of a transition from $QUORUM^{p_{i-1}}_{current}$ to $QUORUM^{p_i}_{new}$ given the executed $wload^{p_{i-1}}$ with $QUORUM^{p_{i-1}}_{current}$.
$\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload^{p_i}}, QUORUM^{p_{i-1}}_{current}, QUORUM^{p_i}_{new})$	Denotes the expected benefit when the predicted workload $\widehat{wload^{p_i}}$ is executed with $QUORUM^{p_i}_{new}$ compared to that of $QUORUM^{p_{i-1}}_{current}$.
$PromoSL$	Denotes the set of slave sites that are promoted to core sites during a reconfiguration.
$updateCost(s)$	Denotes the costs of updating the site s.

Table 5.3: QuAD symbols and notations.

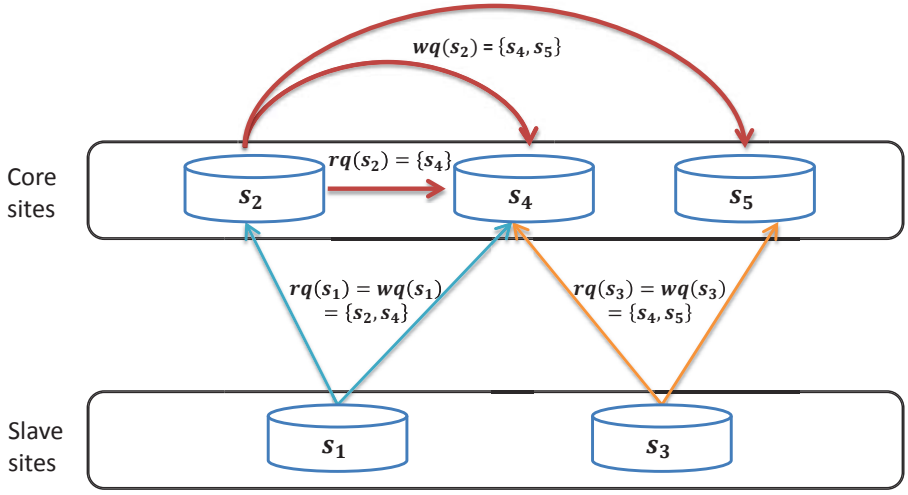


Figure 5.17: QuAD quorum configuration.

1. Each core site will create quorums – denoted as *core quorums* – that consist of core sites only. A core read quorum (*crq*) consists of the majority of core sites, whereas a core write quorum (*cwq*) consists of all core sites.

2. Each slave site will construct its quorums – denoted as *slave quorums* – by including the majority of core sites. Slave quorums always consist the majority of core sites and both the read and write quorums are the same ($srq = swq$). Hence, a quorum might not include the majority of all sites, but at least the majority of core sites.

3. The quorums of a site apply for all transactions executed by that site, i.e., the quorums are not chosen on a per transaction basis.

4. Each site includes itself in its read and write quorum.

The lifecycle of transactions in QuAD is similar to that depicted in Figures 5.16a and 5.16b (Example 5.6). The role of the sites and the quorum construction strategy is what changes in QuAD. The construction strategy is based on the idea that core sites communicate with core sites only, but never with a slave site. The reason for this is that 'strong' sites do not slow down each other. Thus, there is a bi-directional communication between core sites, and only a unidirectional communication between slave sites and core sites, i.e., a slave site can access a core site, but never vice-versa. The roles of the sites are not static and may change if site properties, such as the load, change. This may necessitate an adaptation of quorums, which may lead to old cores being demoted to slaves, and slaves being promoted to core sites.

Example 5.6 (Transaction Execution in QuAD)

In what follows we will describe the transaction execution in QuAD based on the scenario depicted in Figure 5.17. A transaction may be received by a core site or by a slave site.

1. In the first case, a read-only transaction t is submitted to the core site s_2. Since $rq(s_2) = \{s_2, s_4\}$, t needs to access the data objects at s_2 and s_4, i.e., at the *majority* of core sites. It will construct the final response by taking those values that have the highest commit timestamp.

2. In the second case, an update transaction t is submitted to s_2. Transaction t will get the most recent data by contacting the sites that consist the $rq(s_2)$. In the next step, t is executed and new values for the objects are produced, which are then eagerly committed to the sites that consist the write quorum ($wq(s_2) = \{s_2, s_4, s_5\}$), i.e., *all (available)* core sites.

3. In the third case, a read-only transaction is submitted to the slave site s_1. Its read quorum consists of the *majority* of core sites: $rq(s_1) = \{s_1, s_2, s_4\}$. Thus, the final response is constructed by reading the values locally and at s_2 and s_4 (the majority of core sites), and taking those values that have the highest timestamp.

4. In the fourth case, an update transaction is submitted to s_1. The execution process is similar to the case when an update transaction is executed by a core site, except that now only the *majority* of core sites is eagerly updated: $wq(s_1) = \{s_1, s_2, s_4\}$.

Based on Example 5.6 we can derive two main task for the quorum construction in QuAD. First, it is necessary to define a scoring model that assign *scores* to sites. The score of a site determines its strength, i.e., its role (core or slave). And second, a model is necessary that defines the assignment of slaves to cores, i.e., that determines the slave quorums. The model should not only consider the score of the sites when assigning slaves to cores but should also strive for a balanced assignment. Core sites with a high score may attract too many slaves and may thus become a bottleneck, and degrade the performance of the entire system.

Site Score

The score of a site s_i is based on its load $load(s_i)$, and its distance $rtt(s_i)$ to other sites. Let CN denote the distance matrix:

$$\begin{pmatrix} 0 & rtt(1,2) & \cdots & rtt(1,|S|) \\ \vdots & \ddots & & \vdots \\ rtt(i,1) & \cdots & 0 & rtt(i,|S|) \\ \vdots & & \ddots & \vdots \\ rtt(|S|,1) & \cdots & & 0 \end{pmatrix} \tag{5.24}$$

In Equation (5.24), $rtt(i,j)$ denotes the RTT between s_i and s_j and $rtt(i,j) = rtt(j,i)$. The i^{th} row of CN defines the RTT of s_i to all other sites in the system: $rtt(s_i) = [CN(i,1), \cdots, CN(i,|S|)]$. The score of a site is defined as follows:

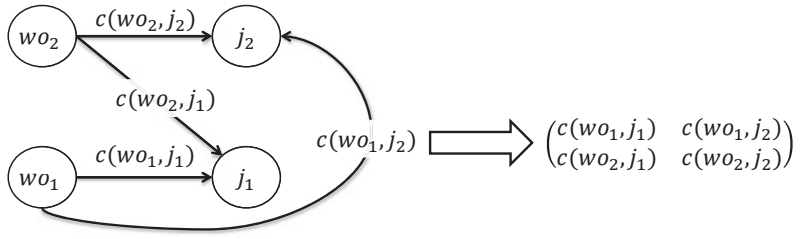


Figure 5.18: Graph and matrix representation of the assignment problem.

$$\begin{aligned} score(s_i) &= w_{rtt} \cdot rttsc(s_i) + w_{load} \cdot loadsc(s_i) \\ rttsc(s_i), loadsc(s_i) &\in [0,1] \end{aligned} \tag{5.25}$$

$$\begin{aligned} rttsc(s_i) &= 1 - \frac{\|rtt(s_i)\|}{\max\limits_{\forall s_k \in S}(\|rtt(s_k)\|)} \\ loadsc(s_i) &= 1 - \frac{load(s_i)}{\max\limits_{\forall s_k \in S}(load(s_k))} \end{aligned} \tag{5.26}$$

Based on the Equation (5.25), the κ sites that have the highest scores are chosen to become core sites. Initially, as no load data are available from the sites, each one will get the same load score. This means that the RTT will be the determining factor for the score of the sites. In case all sites get the same score, the κ-sites will be chosen randomly.

Determining the Quorums of Core Sites

The choice of the number of core sites has a considerable impact on the overall performance of QuAD. It is also crucial for the availability of QuAD, as the core sites are included in quorums of both core sites and slave sites (see Figure 5.17). The lower the number of core sites, the lower the availability of QuAD, and the lower the commit and read overhead. However, the load balancing capabilities also decrease with decreasing number of core sites. With κ, it is possible to simulate the behavior of different existing approaches, such as ROWAA. If, for example, all sites are core sites ($\kappa = S$), then the cwq will include all available sites ($cwq = CS = S$). Consequently, it would be safe from a consistency point of view to access a single site in case of read-only transactions ($crq(cs) = cs$), which corresponds to the lifecycle of transactions in ROWAA.

The read quorum of each core sites consists of the majority of core sites, and the write quorum consist of all core sites:

$$\begin{aligned} crq(cs) &= cs \cup (CQ_r \subseteq CS) \Rightarrow |CQ_r| = \lfloor \frac{|CS|}{2} \rfloor \\ cwq(cs) &= CS \end{aligned} \tag{5.27}$$

Determining the Quorums of Slave Sites

Each slave site will choose the majority of core sites as part of its quorums, and the read and write quorums are equal:

$$swq(sl) = srq(sl) = sl \cup (SQ \subseteq CS) \Rightarrow |SQ| = \lfloor \frac{|CS|}{2} \rfloor + 1 \tag{5.28}$$

Selecting the majority of the cores is crucial to ensure the intersection property between quorums of the slave sites, as there is no communication between the slave sites. The main question is however how to determine the quorums of the slaves, i.e., the subset of core sites that a slave site will be attached to so that the cost is minimized? There are two main issues here. First, we need to consider the cost of assigning a slave site to a subset of core sites so that in overall we minimize the average cost. Second, if we include the same core site in too many quorums, then that site will become a bottleneck and degrade the overall performance. This means that we need to update the costs each time we assign a slave site as the cost of the core sites will increase with every slave site attached to them.

Determining the slave quorums corresponds to the assignment problem [Kuh10, Mun57], which is defined as follows. Let WO denote the set of workers, and J the set of jobs. The goal is now to assign the jobs to the workers so that the overall cost is minimized:

$$min \sum_{j}^{|J|} cost(wo, j) \tag{5.29}$$

The assignment problem is best modeled as a directed graph $G = (WO, J; E)$ with WO denoting the worker-vertices, J the job-vertices, and with edge $e \in E$ having a non-negative weight $cost(wo, j)$, which denotes the cost for assigning j to wo (Figure 5.18). The assignment problem can be solved using the Hungarian algorithm, which has a complexity of $\mathcal{O}(n^3)$ with $n = max(|W|, |J|)$ [Mun57].

The same analogy can be used to determine the slave quorums. For that, we need to perform two steps:

1. We need to define the cost model for assigning slaves to core sites as defined in Equation (5.29). The cost model should consider the load of the cores and the distance of the slaves to the core sites.

2. The original definition of the assignment problem considers a one-to-one assignment (Figure 5.18) of jobs to workers, whereas in QuAD a slave is assigned to a subset of core sites. Hence, we need to map the one-to-one assignment to a one-to-many assignment (Example 5.7).

Example 5.7 (Assignment of Slaves to Core Sites)

In the scenario depicted in Figure 5.17, s_1 and s_3 denote the slave sites, and s_2, s_4 and s_5 the core sites. In order to guarantee the intersection property between s_1 and s_3, they must include the majority of cores in their quorums. In this case the slave quorums

consist of each slave and two of the core sites, i.e., there are three different assignments of slaves to quorums:

$$\begin{aligned} srq(s_1) = swq(s_1) &\mapsto \{s_1 \cup SQ_1, s_1 \cup SQ_2, s_1 \cup SQ_3\} \\ srq(s_3) = swq(s_3) &\mapsto \{s_3 \cup SQ_1, s_3 \cup SQ_2, s_3 \cup SQ_3\} \\ SQ_1 &= \{s_2, s_4\} \\ SQ_2 &= \{s_2, s_5\} \\ SQ_3 &= \{s_4, s_5\} \\ \mathcal{SQ} &= \{SQ_1, SQ_2, SQ_3\} \end{aligned}$$

There are $|\mathcal{SQ}| = \binom{|CS|}{\lfloor \frac{|CS|}{2} \rfloor + 1}$ majority subsets of core sites, and $|\mathcal{SQ}|^{|SL|}$ possible assignments of slaves to core sites.

In order to cope with the one-to-one mapping that is assumed by the assignment algorithm, we need to combine the cost of individual core sites to a single cost value. Let SQ denote a majority of core sites to which a slave site sl_i needs to be assigned. The costs of assigning sl_i to individual core sites $(cost(sl_i, cs_j), \cdots, cost(sl_i, cs_{|SQ|})$ are known and combined to a single value using the *max* function. In what follows we use $cost(sl_i, SQ)$ to denote the cost of constructing a quorum consisting of the slave site sl_i and the core sites in SQ. Let $w_1, w_2, w_3 \in \mathbb{R}^+$, then the cost $cost(sl_i, SQ)$ for assigning sl_i to SQ is defined as follows:

$$\begin{aligned} cost(sl_i, SQ) = w_1 \cdot \text{commCost}(sl_i, SQ) \\ + w_2 \cdot \text{loadCost}(sl_i, SQ) \\ + w_3 \cdot \text{balancingPen}(sl_i, SQ) \end{aligned} \tag{5.30}$$

Equation (5.30) jointly considers communication and load costs when assigning slaves to core sites. Moreover, in order to avoid that certain core sites become a bottleneck, their costs are continuously increased with each slave site assigned to them (*balancingPen*). The cost components are defined as follows ($\overline{load(s)}$ denotes the normalized load of site s):

$$\begin{aligned} \text{loadCost}(sl_i, SQ) &= \overline{load(sl_i)} \cdot \max_{\forall cs_j \in SQ} (\overline{load(cs_j)}) \\ \overline{load(s)} &= \frac{load(s)}{\max\limits_{\forall s_k \in S} (load(s_k))} \end{aligned} \tag{5.31}$$

$$\text{commCost}(sl_i, SQ) = \frac{\max\limits_{\forall cs_j \in SQ} rtt(sl_i, cs_j)}{\max\limits_{\forall s_k, s_m \in S \wedge k \neq m} rtt(s_k, s_m)} \tag{5.32}$$

Equations 5.31 and 5.32 define the costs for assigning a slave to core sites by considering the slave load, the load of the core sites, and the distance of the slave to the core

sites consisting SQ. It is crucial to not only consider the load of the core sites, but also that of the slave to be assigned, as the performance impact at the cores is also influenced by the slave load.

$$\text{balancingPen}(sl_i, SQ) = \exp^{(\text{countSL}(SQ)+1-|SL|)} \tag{5.33}$$

In Equation (5.33), *balancingPen* is used to increase the cost of SQ with increasing number of slaves that have been assigned to SQ. It fulfills the purpose of possibly even assignment of slaves to core sites, as in case certain core sites are included more frequently in the quorums, then they may become a bottleneck [SFS15]. $countSL(SQ)$ defines the number of slave sites that have already been assigned to SQ. The *balancingPen* function will never become zero, and will get the maximum value of 1 if all slave sites are assigned to SQ, i.e., to the same core quorum. Such a scenario would lead to SQ becoming a bottleneck and degrading the overall performance.

5.3.3 Cost Model

As described in Section 4.3, QuAD aims at reducing operational costs by reducing transaction response time. The lower the response time the higher the generated utility for applications (see Equation 4.2). Therefore, the total costs in QuAD are determined by the operational costs:

$$totalCost(wload^{p_i}, QUORUM) = opCost(wload^{p_i}, QUORUM) \tag{5.34}$$

The operational cost of QuAD are defined based on the score of a certain quorum configuration:

$$\begin{aligned} opCost(wload^{p_i}, QUORUM) &= 1 - exp^{(1-\frac{1}{score(wload^{p_i}, QUORUM)})} \\ score(wload^{p_i}, QUORUM) &\in]0, 1] \end{aligned} \tag{5.35}$$

In Equation (5.35), $score(QUORUM)$ denotes the overall average score over all quorums and is defined as follows:

$$score(wload^{p_i}, QUORUM) = \frac{\sum\limits_{q \in QUORUM} score(wload^{p_i}, q)}{|QUORUM|} \tag{5.36}$$

The score of a quorum q is calculated as follows:

$$score(wload^{p_i}, q) = \prod_{s \in q} score(wload^{p_i}, s) \tag{5.37}$$

As the average score of the quorums goes to zero, the operational costs go to maximum:

$$\lim_{score(wload^{p_i}, QUORUM) \to 0} opCost(wload^{p_i}, QUORUM) = 1. \tag{5.38}$$

Thus, it pays off to construct the quorums in such a way so that their score is maximized, as that will minimize the operational costs.

5.3.4 Configuration Model

The expected gain of adapting the quorum configuration from the currently active $QUORUM^{p_{i-1}}_{current}$ to a new configuration $QUORUM^{p_i}_{new}$, for the predicted workload $\widehat{wload}^{p_i}$ is defined as follows:

$$\widehat{gain}(\widehat{wload}^{p_i}, QUORUM^{p_{i-1}}_{current}, QUORUM^{p_i}_{new}) = \widehat{totalCost}(\widehat{wload}^{p_i}, QUORUM^{p_{i-1}}_{current}) - \widehat{totalCost}(\widehat{wload}^{p_i}, QUORUM^{p_i}_{new}) \tag{5.39}$$

QuAD is a dynamic quorum-based protocol, in which the status of sites and their properties are continuously monitored. The following cases may trigger a reconfiguration of quorums:

- Changes in the site properties. For example, the load of sites may change, which may invalidate the quorums. QuAD's load prediction is based on the model described in Section 4.2, which provides sufficient information for determining a suitable configuration. As we assume non-mobile sites, the RTT between should remain roughly constant. Thus, QuAD will not monitor the RTT, but only the load of the sites.
- Site failures might need to be addressed by QuAD by adapting its configuration. For example, if some core sites fail, then the quorums of the affected slave site need to be adapted.
- Joining of new sites, which be the case if failed sites recover, or new sites are deployed in reaction to increasing availability demands of applications.

Any transition from the currently active may generate costs especially if the new configuration promotes slaves to core sites. Each slave site to be promoted must update its objects by contacting current core sites.

$$tCost(wload^{p_{i-1}}, QUORUM^{p_{i-1}}_{current}, QUORUM^{p_i}_{new}) = \prod_{sl \in PromoSL} updateCost(wload^{p_{i-1}}, sl)$$
$$updateCost(wload^{p_{i-1}}, sl) = \frac{\#objectsToUpdate}{|LO|} \tag{5.40}$$

In Equation (5.40), $PromoSL$ denotes the set of slaves that are promoted to core sites, and $updateCost(wload^{p_{i-1}}, sl)$ denotes the cost of updating the slave site sl that is promoted to a core site.

A reconfiguration is initiated only if the expected gain outweighs the transition costs by a certain threshold th:

$$\widehat{benefit}(wload^{p_{i-1}}, \widehat{wload}^{p_i}, QUORUM^{p_{i-1}}_{current}, QUORUM^{p_i}_{new}) = \widehat{gain}(\widehat{wload}^{p_i}, QUORUM^{p_{i-1}}_{current}, QUORUM^{p_i}_{new}) - tCost(wload^{p_{i-1}}, QUORUM^{p_{i-1}}_{current}, QUORUM^{p_i}_{new}) > th \tag{5.41}$$

Algorithm 5: The Algorithm for determining the most optimal quorum configuration. The *determineConfiguration* function is invoked by Algorithm 1.

Input: S, $wload^{p_{i-1}}$,$\widehat{wload^{p_i}}$, $QUORUM^{p_{i-1}}_{current}$, κ, RTT
Output: Configuration for the next period

Function *determineConfiguration* **is**
- maxBenefitConf $\leftarrow QUORUM^{p_{i-1}}_{current}$;
- totalCosts $\leftarrow$ `calculateTotalCosts(`$\widehat{wload^{p_i}}$,$QUORUM^{p_{i-1}}_{current}$`)` ;
- $CS \leftarrow$ `determineCores(`κ, S`)` ;
- $QUORUM^{p_i}_{new} \leftarrow$ `determineQuorums(`$\widehat{wload^{p_i}}$, CS, κ,RTT`)` ;
- tmpTotalCosts $\leftarrow$ `calculateTotalCosts(`$\widehat{wload^{p_i}}$, $QUORUM^{p_i}_{new}$`)` ;
- **if** tmpTotalCosts < totalCosts **then**
  - tmpTCosts $\leftarrow$ `calculateTransCosts(`$wload^{p_{i-1}}$, $\widehat{wload^{p_i}}$, $QUORUM^{p_{i-1}}_{current}$, $QUORUM^{p_i}_{new}$`)` ;
  - tmpBenefit $\leftarrow$ totalCosts- tmpTotalCosts- tmpTCosts;
  - **if** tmpBenefit > th **then**
    - maxBenefitConf $\leftarrow QUORUM^{p_i}_{new}$;
- **return** maxBenefitConf;

Algorithm 6: The Algorithm for determining the core sites. The *determineCores* function is invoked by Algorithm 5.

Input: S, κ
Output: Set of core sites CS

Function *determineCores* **is**
- $CoreSites \leftarrow \{\}$;
- **foreach** $s \in S$ **do**
  - `determineScore(`s`)` ;
- `sortSitesByScore(`S,*DESCENDING*`)` ;
- `// determine the core sites (`CS`)`
- $CoreSites$`.addAll(`S`.getElements(`$1, \kappa$`))` ;
- **return** $CoreSites$;

5.3.5 Adaptive Behavior of QuAD

In what follows we will provide a step-by-step description of the adaptive behavior of QuAD based on the scenario depicted in Figure 4.10. The choice of the most optimal quorum configuration is described in Algorithm 5.

1. In the first step, before p_{i-1} is finished, the workload prediction for p_i is initiated (Algorithm 1).

2. In the second step, the predicted workload for each of the sites is used to calculate their score (Section 5), and determine the core sites. Next, the quorums for each of the sites are defined based on the models described in the Sections 5.3.2 and 5.3.2.

Algorithm 7: The Algorithm for determining the quorums. The *determineQuorums* function is invoked by Algorithm 5.

Input: $\widehat{wload^{p_i}}$, S, CS,RTT
Output: Quorum configuration

Function *determineQuorums* **is**

```
QUORUM ← {} ;
// Determine the quorums of core sites
foreach cs ∈ CS do
    cwq(cs) ← CS ;
    crq(cs) ← getMajorityRandomly(CS) ;
    quorum(cs) ← {cwq(cs), crq(cs)} ;
    QUORUM.add(quorum(cs)) ;
// Constructs all possible majority subsets from the set of
   core sites - see Example 5.7.  SQ = {SQ_1, ...}.
SQ ← constructMajoritySubsets(CS) ;
// See  Equation (5.30)
aggregateCosts(SQ) ;
SL ← S \ CS ;
/* Use the Hungarian Algorithm to assign the slaves (SL) to
   quorums consisting of core sites SQ [SW11].  The function
   Hungarian returs the list of read and write quorums for each
   slave site (srq, swq).                                    */
QUORUM.addAll(hungarian(wload^{p_i}, SQ, SL, w1, w2, w3, RTT)) ;
return QUORUM ;
```

3. The proposed quorum configuration is used to determine the gain and the transfer costs (steps three and four) based on the Equations 5.35, 5.39 and 5.40.

4. If the proposed quorum configuration generates a benefit (Equation 5.41), by also considering the transition costs, then a reconfiguration is initiated (step five), which will execute all necessary steps required by the new configuration.

5.3.6 Intersection Property

Theorem 5.1. *(Intersection property of quorums in QuAD) Given a set of sites S. QuAD will construct quorums in such a way so that each read quorum will intersect with each write quorum, and any two write quorums will intersect with each other (Algorithm 5).*

Proof. Let $cwq(cs_i)$ denote the write quorum of a core site cs_i, and $crq(cs_i)$ denote the read quorum of cs_i. The write quorum of cs_i consists of all core sites, whereas its read quorum from the majority of sites. It is trivial to prove that the $cwq(cs_i)$ intersects with the $cwq(cs_j)$ of any other core site cs_j, and $crq(cs_i)$ will intersect with the $cwq(cs_j)$ of any other cs_j: $\forall cs_j \in CS, i \neq j : |cwq(cs_i)| + |cwq(cs_j)| > \kappa$ and $|crq(cs_i)| + |cwq(cs_j)| > \kappa$.

Next we need to prove that any slave quorum (read and write quorums are the same) will intersect with any slave quorum, and with any core read and write quorum.

Let $sq(sl_i) = srq(sl_i) = swq(sl_i)$ denote the quorums of the slave sl_i. As each slave site will create quorums consisting of the majority of core sites it follows that, for any two slave sites sl_i and sl_j with $i \neq j$: $sq(sl_i) \cap sq(sl_j) \neq \emptyset$.

The slave quorum $sq(sl_i)$ will intersect with any $cwq(cs_j)$ and $crq(cs_j)$ as $|sq(sl_i)| + |cwq(cs_j)| > \kappa$ and $|sq(sl_i)| + |crq(cs_j)| > \kappa$. □

5.4 Summary

In this chapter, we have described the concepts of our CCQ protocols, which can dynamically adjust their configuration based on realistic cost models and application workload. Moreover, we have discussed in detail their cost and configuration models that steer the dynamic adaptation, and ensure that the adaptation costs do not outweigh their gain. This control mechanism avoids frequent and unnecessary reconfigurations, which in long term would not generate any benefit to applications.

The CCQ protocols allow for a flexible and seamless enlargement of their configuration spaces. Enlarging the configuration space of a CCQ protocol also increase its optimization capabilities. However, at the price of increased overhead, which is in sharp contrast to the requirements of dynamically adaptable protocols.

6

CCQ Implementation

IN THIS CHAPTER, we describe the implementation details of the workload and cost-driven CCQ protocols introduced in the previous chapters. We will summarize the functionality common to all CCQ protocols and the modules implementing this common functionality. For each CCQ protocol, we will describe its specific implementation, and the way it reuses (inherits) common functionality. Moreover, we provide a detailed discussion of the online reconfiguration approaches that allow the CCQ protocols to adapt dynamically the behavior without violating their correctness. The chapter concludes with the overview of the software stack of the modules and the CCQ deployment architecture.

6.1 System Overview

Each CCQ protocol is implemented as a *module*. A module[1] is the aggregation of operations and data for providing a self-contained functionality [PB93]. Modules have well-defined interfaces and a set of parameters that influence their behavior. As depicted in Figure 6.1, a module can consist of (sub-) modules and can use the functionality provided by other modules.

We distinguish between *common* and *dedicated* modules. The former provide common functionality, whereas the later provide functionality that is tailored to the needs of a specific CCQ protocol. A dedicated module can reuse and extend the functionality provided by common modules. A CCQ system consists of a set of sites that host the modules, and optionally the data (Figure 6.2).

As described in Chapter 5, the CCQ protocols have the following properties in common. First, they use the same 1SR implementation that is based on S2PL and 2PC. Sec-

[1]There is a slight difference between a module and a service. A module usually denotes a piece of software that is part of another software. A service in contrast, can provide its functionality independently. The term Service is used when the functionality is provided and accessed through a network [MZ]. Despite this difference, in the context of our discussion, we will stick to the term module, that will denote a piece of software that is part of another software, or that provides an independent functionality accessible through a network.

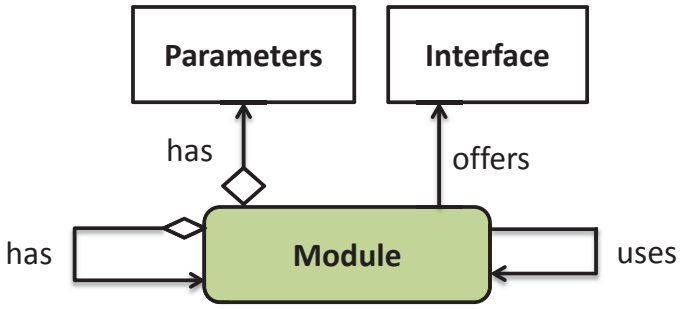


Figure 6.1: Modules and their properties.

ond, they need to collect, manage and analyze meta-data at runtime. Third, they need to predict the workload based on the collected meta-data from each of the sites consisting the DDBS. And fourth, based on the predicted workload and other parameters, each CCQ protocol needs to decide on the best configuration at runtime and reconfigure the system if the configuration is to be adapted.

Based on these properties, we have identified a set of modules that provide common functionality, which can be further specialized or reused by concrete CCQ modules. In what follows, we will describe the functionality provided by the common modules, which are depicted in Figure 6.3.

TimestampManager

The *TimestampManager* provides the functionality for assigning system-wide unique timestamps to transactions. A timestamp is defined as follows: $\tau = currentTimeInMs.ipaddress$, with $currentTimeInMs$ denoting the current time in milliseconds retrieved from the underlying operating system at the site, and $ipaddress$ denoting the Internet Protocol (IP) address of a site. The comparison rule for two timestamps is defined as follows:

$$\begin{aligned}\tau_1 < \tau_2 \Leftrightarrow\ & \tau_1.currentTimeInMs < \tau_2.currentTimeInMs \\ & \vee\ \tau_1.currentTimeInMs = \tau_2.currentTimeInMs \wedge \tau_1.ipaddress < \tau_2.ipaddress\end{aligned}$$

2PCManager

The *2PCManager* is responsible for the execution of 2PC messages. It consists of a First In, First Out (FIFO) queue for managing the messages, and a thread pool for their execution (Figure 6.5). One of the *2PCManagers* involved is declared being the *coordinator* for the processing of a particular distributed transaction. For example, the *2PCManager* which has initiated a global transaction can be chosen as coordinator. All other *2PCManagers* are called *agents*. The role also defines the information that a *2PCManager* needs to store for each transaction on a persistent storage as defined by the 2PC protocol [BHG87].

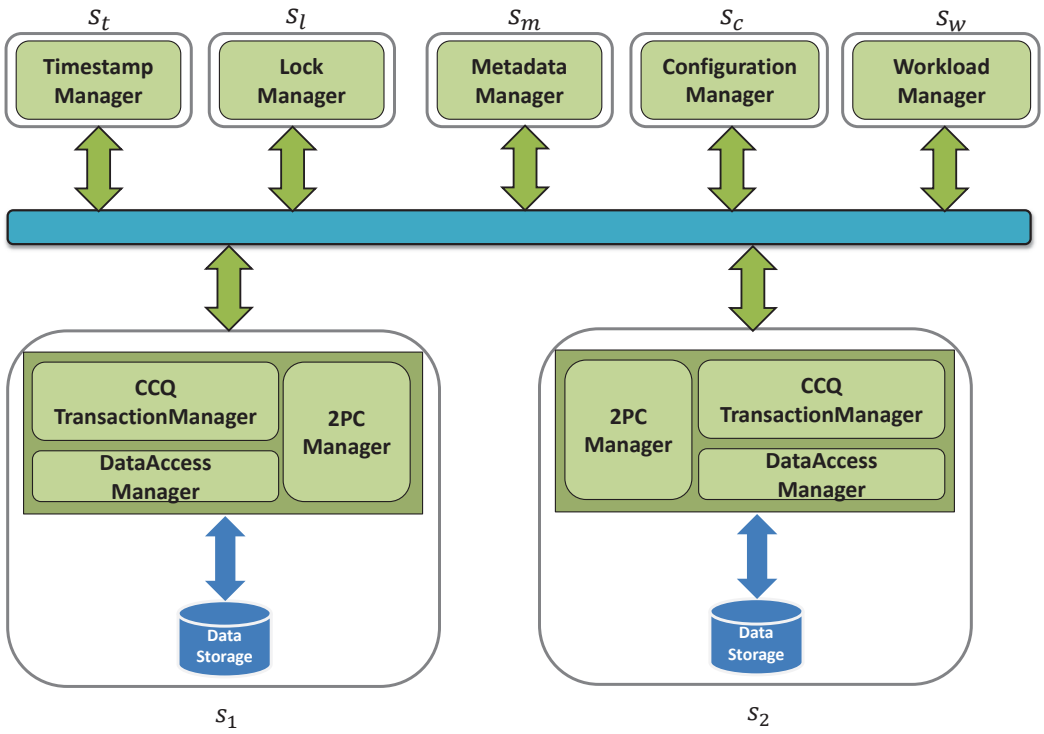


Figure 6.2: CCQ system overview.

We have implemented a multicast-like approach sending 2PC messages to the agents. This means that a number of threads are started, one for each of the agents, which send the messages to the agents concurrently and collect their responses. The evaluation of the responses and the decision on the transaction faith is taken once all responses are available. Thus, the slowest agent determines the overall duration of the 2PC coordination.

LockManager

The *LockManager* module implements the S2PL functionality, i.e., it provides operations for locking and unlocking a set of objects based on the S2PL rules. The locked objects and the ids of transactions holding the locks are stored in a *Hashtable* that guarantees $\mathcal{O}(1)$ lookup. As transactions need to predeclare their read and write sets, the *LockManager* has all necessary information to acquire the locks based on the *all or nothing* principle. This means that a transaction either has successfully acquired all locks or none. The lock requests can be operated in the blocking mode, i.e., a request is set in the wait-state if the locks could not be acquired, or in the timeout mode, in which the request will fail if all locks could not be acquired withing the specified timeout.

DataAccessManager

The goal of the *DataAccessManager* is to encapsulate the access to the underlying database, so that support for different databases languages (e.g., Structured Query Language (SQL), N1QL [Cou], UnQL [BFS00] and others) can be added seamlessly in form

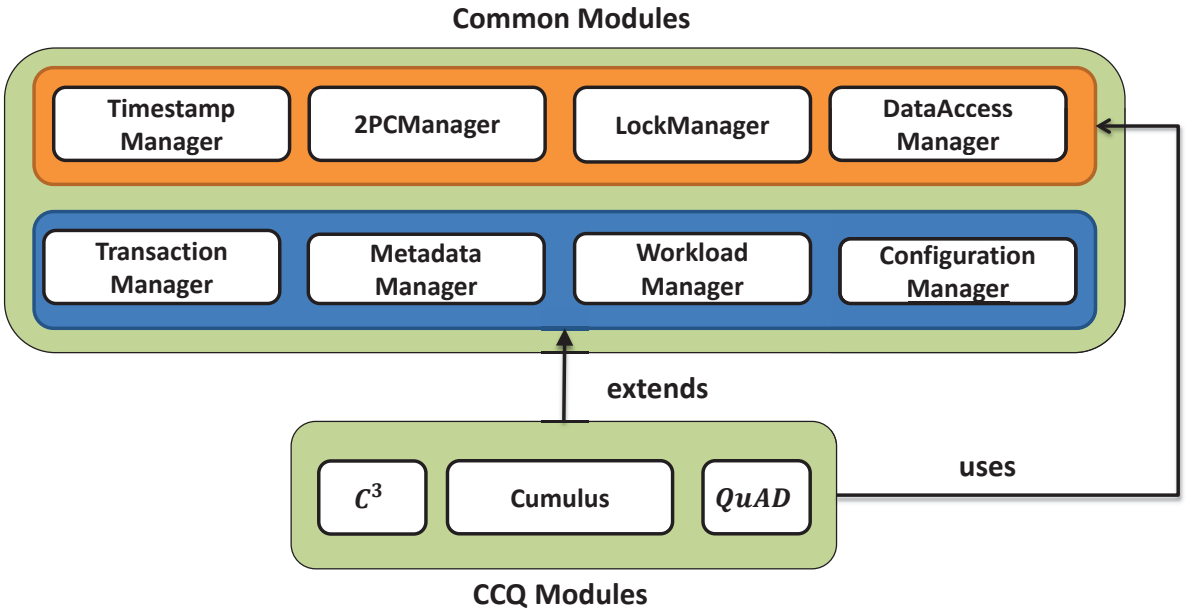


Figure 6.3: Common and dedicated modules.

of modules that implement the mapping of the generic API to the specific database language.

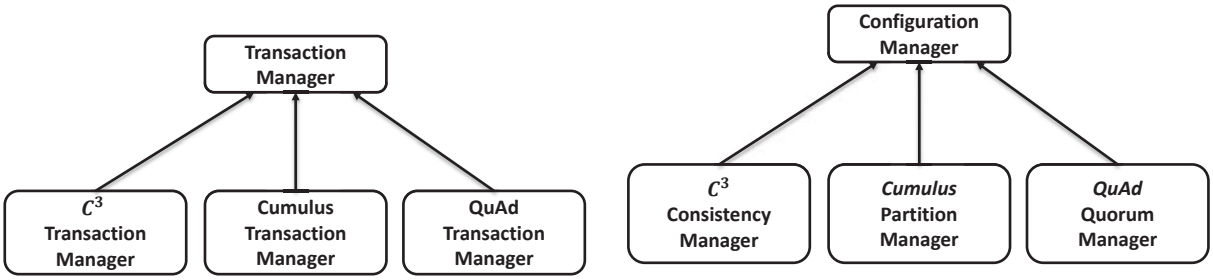
TransactionManager

The *TransactionManager* is responsible for the management of the entire transaction lifecycle. It contains a FIFO queue to which incoming transactions are added and a thread pool that continuously pulls transactions from the queue and executes them (Figure 6.5). The execution steps are as follows:

1. For each transaction it receives the *TransactionManager* will contact the *TimestampManager* and request a unique timestamp for the transaction.

2. The read and write sets of a transaction are extracted, and the lock request containing all actions is forwarded to the *LockManager*. If the same object is accessed by a read and write operation inside the same transaction, then only the write action is forwarded, i.e., only an exclusive lock needs to be acquired for that object.

3. The transaction processing, i.e., the processing of each action, is initiated once all locks are successfully acquired, which includes reading object values and generate new ones. For that, the *TransactionManager* can communicate with the *DataAccessManager* and other *TransactionManagers*. The latter is, for example, necessary in QuAD as a quorum of sites need to be accessed to guarantee access to most recent data. Thus, the concrete processing behavior may be overridden by the CCQ protocols.



(a) Transaction Managers (b) Configuration Managers

Figure 6.4: Extension of the common functionality by the specific CCQ (sub-)modules.

4. Once the processing is done, the transaction needs to be committed. A read-only transaction can immediately commit without any further coordination with the rest of the system, and can release the locks. For update transactions, the default behavior is to commit eagerly all available sites (ROWAA). However, this behavior may be overridden by the CCQ protocols (Figure 6.4).

WorkloadManager

A *daemon* will periodically collect the workload from the *TransactionManagers* at a frequency defined by a parameter; that can also be changed at runtime (Algorithm 1). The *WorkloadManager* will store the collected workload, and the predict expected workload based on EMA as described in Section 4.2. Moreover, it will continuously compare the predicted values with the occurred ones in order to optimize the prediction parameters.

The workload is managed in a *Hashtable* with the access pattern denoting the keys, and their occurrences the values. Each access pattern has a hash code [2], so looking up for the existence of the same access pattern in the table is a $\mathcal{O}(1)$ operation. The particular CCQ protocols may derive further information from the predicted workload that is necessary to determine the optimal configuration.

MetadataManager

The *MetadataManager* is responsible for the collection and management of system-wide metadata, such as available sites, the RTT between them, site failures and others. The metadata is continuously retrieved by a *daemon* from the underlying system and managed in a *Hashtable*. The size of metadata is small and does not grow with time, as new data will replace old one. The *TransactionManagers* at the sites are the primary contact of the *MetadataManager* for retrieving the metadata.

ConfigurationManager

The *ConfigurationManager* retrieves the predicted workload from the *WorkloadManager* and the metadata from the *MetadataManager* to determine the optimal configuration based on the cost and configuration model of the particular CCQ protocols. Thus, the

[2]Equal access patterns must have the same hash code.

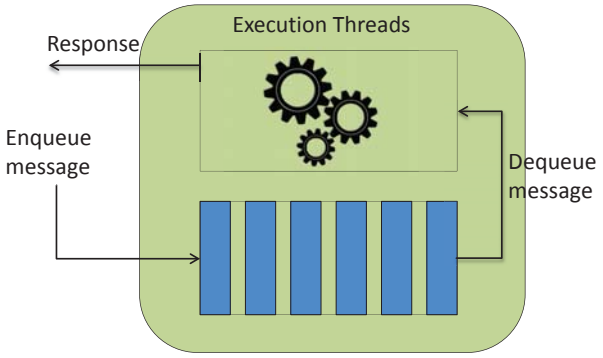


Figure 6.5: Thread pool for the execution of messages at the *TransactionManager* and *2PCManager*

behavior of the *ConfigurationManager* is determined by the particular CCQ protocol (Figure 6.4).

6.2 CCQ Modules

In what follows, we will describe the implementation of CCQ specific modules, that either implement entirely new functionality, or reuse common functionality and extend it where necessary.

6.2.1 C^3 TransactionManager

The *TransactionManager* is responsible for the execution of transactions based on the currently active consistency model. It uses the functionality provided by the generic *TransactionManager* and extends it with two different execution modes, namely the 1SR and EC mode. In contrast to the 1SR mode, in which all available sites are eagerly committed, during the EC mode update transactions will only commit at the local site (Figure 6.6).

As described in Section 6.3.1, if there is a switch from the EC to the 1SR consistency level, sites must first reconcile. The reconciliation process requires the propagation of the most recent values to all sites in the system, which then decide on the winning value based on TWR. To speed up the reconciliation process, each C^3 *TransactionManager* caches all modified objects executed by EC transactions in the `DO`, which contains the id of the modified object, its value and the timestamp of the last modification. The `DO` is used either for the periodical propagation of updates to other sites by the `propagation daemon`, or for the forced reconciliation in case of a switch from the EC to the 1SR mode (Section 5.1).

In any case, each *TransactionManager* will multicast its *DO* to all other sites, receive *DO*s from other sites and merge the received ones with the local *DO*. If the resulting *DO* is not empty, refresh transactions will be initiated that will proactively fetch the data from the other sites using a batch-based approach.

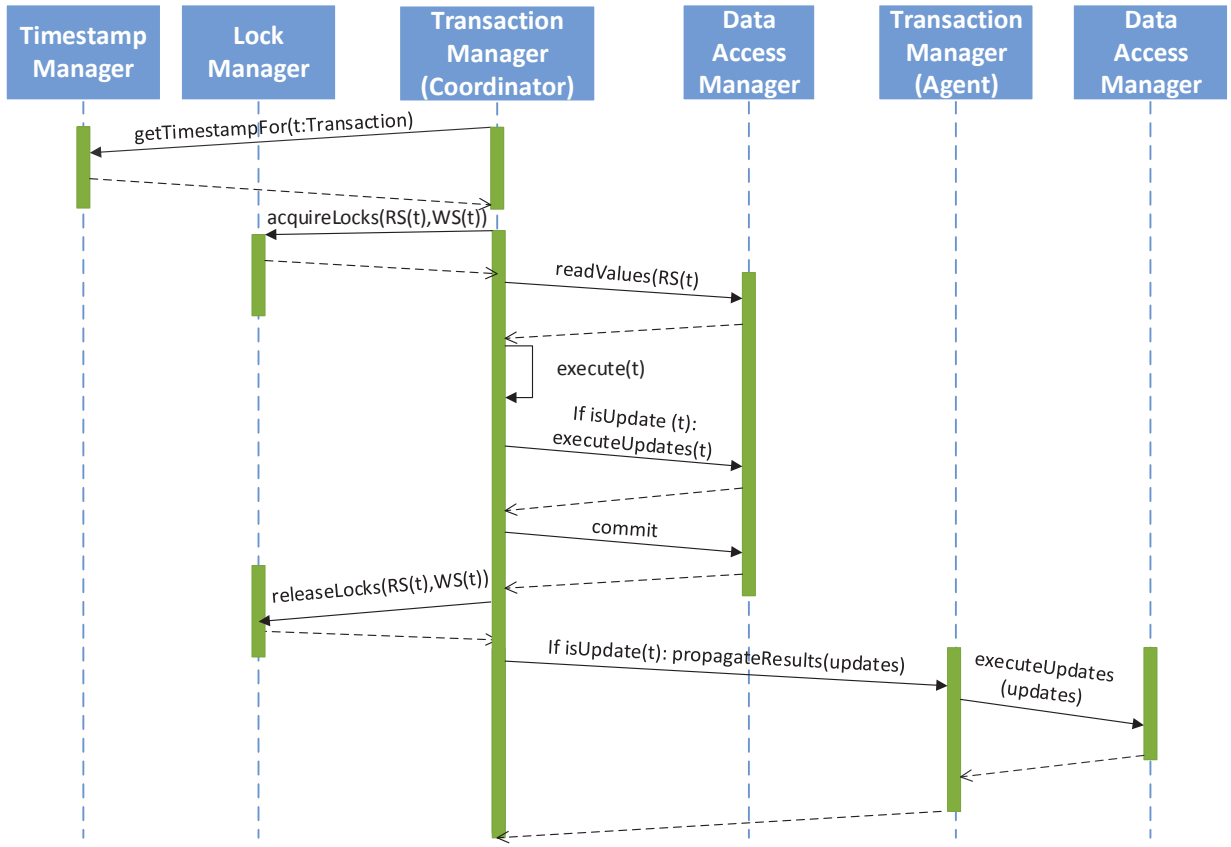


Figure 6.6: Execution of a transaction in C^3 with the EC consistency model.

6.2.2 C^3 ConsistencyManager

The C^3 *ConsistencyManager* is responsible for analyzing the workload and predicting the inconsistency, as well as the consistency costs based on the model described in Section 5.1. It will choose that consistency level (configuration) that incurs the minimal overall costs by also considering the transition costs from the existing consistency level to the new one (Algorithm 3). The *ConsistencyManager* is responsible for the initiation and the coordination of the reconfiguration process described in Section 5.1. For that, it uses the interface provided by the *TransactionManager*.

As described in Section 5.1, C^3 needs to extract the *w/w* conflicts between sites to predict the expected inconsistencies, and with that the expected inconsistency costs. Currently, the conflict detection between access patterns is based on a brute-force approach by comparing access patterns action by action. Using more sophisticated approaches, such a hash-based set intersection algorithms would lead to considerable improvement in the runtime of the conflict detection [DK11]. This aspect is considered as part of future work.

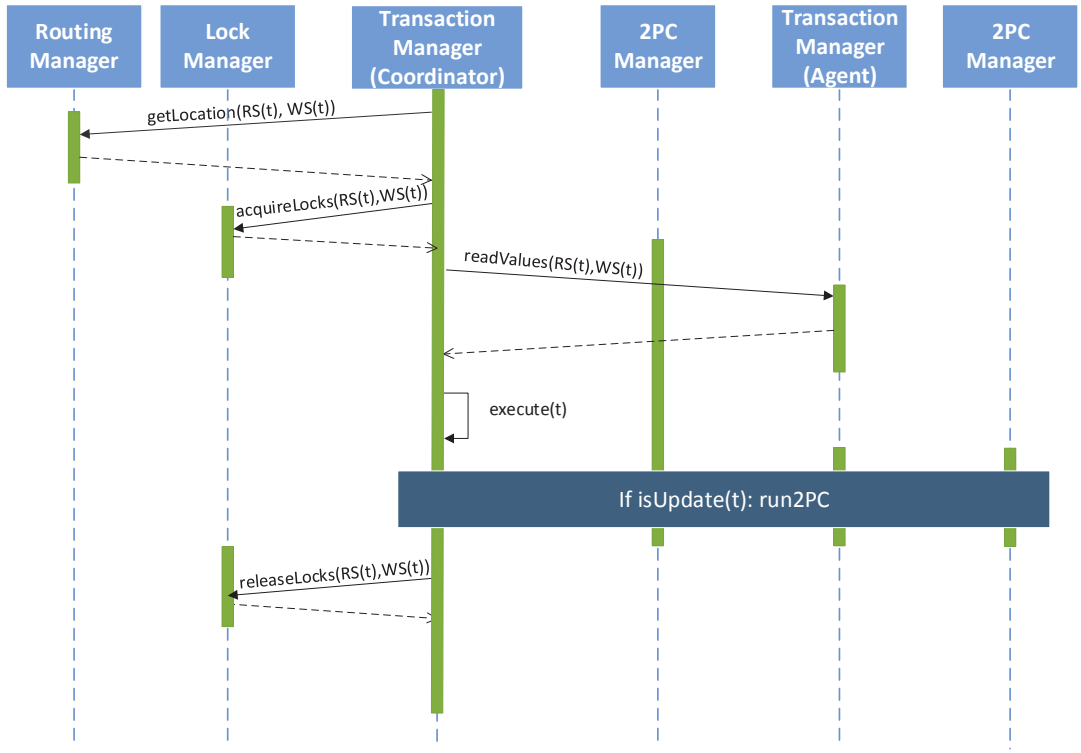


Figure 6.7: Execution of transactions in Cumulus. We have omitted the *DataAccessManager* due to space reasons.

6.2.3 Cumulus TransactionManager

In Cumulus, both read-only and update transactions may be distributed, which is inherent to all partitioned DBSs. In a case of distributed read-transaction, the *TransactionManager* will collocate all operations that can be served locally into a local sub-transaction at the coordinating site; the rest will be greedily distributed as sub-transactions to the sites at which the accessed objects are located. Greedy means in this case that sub-transactions are generated with the maximum number of objects. Distributed update transactions occur as soon as the transaction accesses objects from different partitions, or during the migration of objects (Section 5.2). To preserve ordering of operations inside the same transaction, all objects accessed by the transaction will be collected at the coordinating site, the transaction will get executed and the generated results will be written back to the corresponding sites (Figure 6.7).

6.2.4 Cumulus PartitionManager

The *PartitionManager* module is responsible for analyzing the workload, deriving the significant workload and for generating partitions that best matches the workload (Algorithm 4). With regards to the workload prediction, as described in Section 5.2.3, Cumulus neglects the actions and considers only the objects, i.e., it does not differentiate between read and write operations. The *PartitionManager* handles this aspect. It feeds

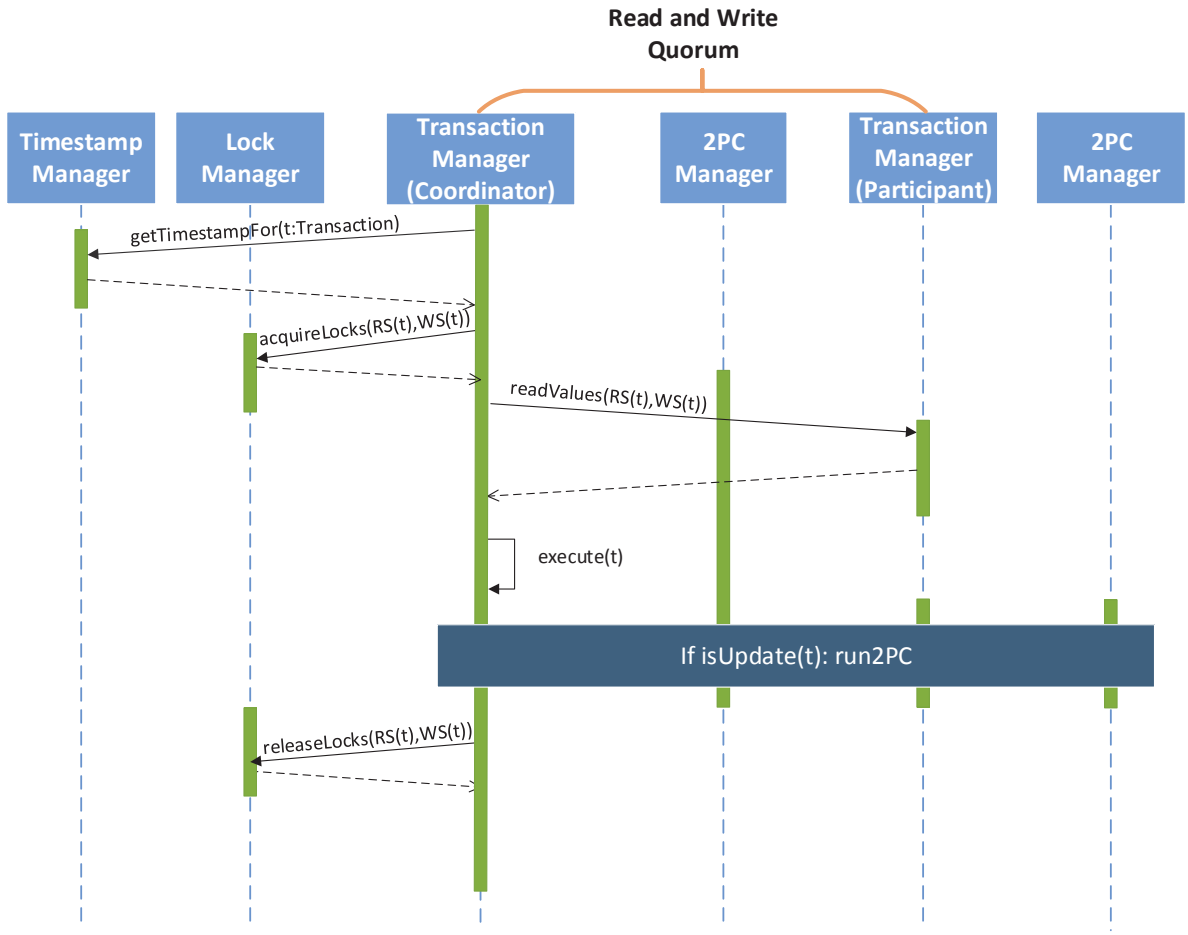


Figure 6.8: Execution of a transaction in QuAD. We have omitted the *DataAccessManager* due to space reasons. The two *TransactionManagers* define the read and write quorums.

the *PartitionEngine* submodule with the workload data, and retrieves the most suitable partitions for the given workload.

6.2.5 Cumulus RoutingManager

In contrast to a fully replicated DBS, in a partitioned database, the management of the routing information is crucial as the data is available only at a subset of sites. Cumulus provides full routing transparency to clients by allowing them to connect to any site in the system and submit transactions to it. All routing information is managed internally in the system and completely hidden from the client.

The behavior of the *RoutingManager* is as follows. The site, i.e., the *TransactionManager*, that receives a transaction will forward it to any of the sites that contain objects accessed by the transaction and that site will become the coordinator of the transaction. It is well possible to declare the receiving site to become a coordinator even if it does not contain any of the objects accessed by the transaction. However, such an approach is not optimal from the data transfer point of view, as the coordinating site must receive

all objects, execute transactions locally and throw away the data. The routing strategy of Cumulus is based on the idea that it is cheaper to transfer transactions than objects.

If clients send randomly transactions to sites, in the worst-case there is an additional forward step required. In that case, the transaction is a remote-transaction as it needs to be forwarded to another site. The probability of a transaction becoming a remote one is $1 - \frac{|S_{rel}|}{|S|}$, with S_{rel} denoting the set of sites containing any object accessed by the transaction.

The distribution degree of transactions is however not mainly determined by the routing strategy, but by the quality of the partitions. In a case of an additional forward step, one might choose the optimal approach with regards to the number of objects transferred in the system by sending the transaction to the site containing the greatest subset of objects accessed by the transactions. Such an approach would have a worst-case complexity of $\mathcal{O}(|S| * m)$ with m being the transaction size. The optimal strategy would reduce the bandwidth consumption as the number of objects transferred would be smaller compared to the naïve approach (assuming same object size). Consider that, if $|S| >> m$ then the complexity is reduced to that of naïve approach routing. The problem with sending transactions to the optimal replica is that a tight coupling between server and client is necessary, as the partition configuration needs to be known by the client.

6.2.6 QuAD TransactionManager

The behavior of the common *TransactionManager* needs a slight modification, as now each read must be executed by the read quorum of the coordinating site, and update transactions must eagerly update the sites of the write quorum (Figure 6.8). If a read retrieves multiple values for the same object, only that value is taken that has the highest timestamp. Thus, a read must retrieve not only the value, but also the commit timestamp of the object.

6.2.7 QuAD QuorumManager

The *QuorumManager* is for determining the quorums for each of the sites based on the cost model defined in Section 5.3.3 (Algorithm 5). For that, it needs access to the predicted workload and the metadata, such as the available sites and the RTT between them. With regards to the workload, it only requires the expected number of transactions for each of the sites.

If the quorums need to be adapted, and if adaptation includes promotion of slave sites to cores, the *QuorumManager* will initiate and coordinate the entire reconfiguration process, as well as the site reconciliation as described in Section 6.3.3.

6.3 CCQ Online Reconfiguration

As already described, a change in the application or infrastructure properties may lead to the necessity of adapting the configuration. The transition to a new configuration

is known as *online reconfiguration*, which, as opposed to the offline configuration, can be done without "shutting down" the system. Common to all CCQ protocols is the requirement that the sites consisting the system must be consistent with regards to the active configuration. This property, known as *one-copy view*, means that no site may take a decision on its own, i.e., sites do not have any autonomy with regards to the active configuration; and that at any point in time, each site behaves in accordance to the same single configuration to which all sites have agreed upon.

A reconfiguration is a distributed activity as it involves coordination between multiple sites in a DDBS. Hence, a distributed coordination protocol, such as 2PC, is necessary to guarantee one-copy view on the active configuration.

During a reconfiguration process, sites may fail. Moreover, site failures may be the reason for initiating a reconfiguration, because they impact the characteristics of the underlying infrastructure. Therefore, a means of detecting failures is necessary.

CCQ protocols assume a *crash-recovery* failure model, in which sites behave correctly (according to their specification), at some point may crash and then recover [CGR11]. This is in sharp contrast to the *byzantine* model, in which sites may misbehave (deviate from their specification).

In CCQ, site failures can be detected via *timeouts*. A dedicated site in the system may be responsible for periodically sending ping requests, or any site may detect a failure based on user transactions. For example, one site s_1 may forward transactions to another site s_2 (e.g., in a partitioned database), and if s_2 does not respond, s_1 inform all other sites about the failure. The basic assumption is that if a site does not respond during a specified timeout, it is assumed that it has crashed. A failed site may later recover, and may require participating again in the system. However, as the site might not be up-to-date with regards to both configuration and application data, it must first reconcile. To properly handle such case, we distinguish between a *recovered* and *operational* site. A site is in a recovered state if it is able to respond to ping requests, but is not yet able to participate in the execution of transactions. The reconciliation consists of activities for bringing application data to a consistent state, but also moving to the currently active configuration. The later includes all activities for acquiring the necessary meta-data related to the active configuration (e.g., the current consistency configuration in C^3).

6.3.1 C^3 Reconfiguration

The reconfiguration process must ensure that all sites have the same view on the consistency level for the workload, and also for each transaction class in case of multi-class transactions.

The one-copy view on the active configuration is ensured using an approach based on 2PL and 2PC as the reconfiguration is a distributed activity. The *ConsistencyManager* is responsible for coordinating the reconfiguration process, which works as follows:

- In the case of a switch from 1SR to EC, the reconfiguration process consists of only notifying the sites about the new consistency level by the *ConsistencyManager* (Figure 6.9). This is done by sending a `prepare` message to all *TransactionManagers* that includes the new consistency level. In response to the `prepare` message, the *TransactionManagers* will set a `lock` locally. All incoming transactions after

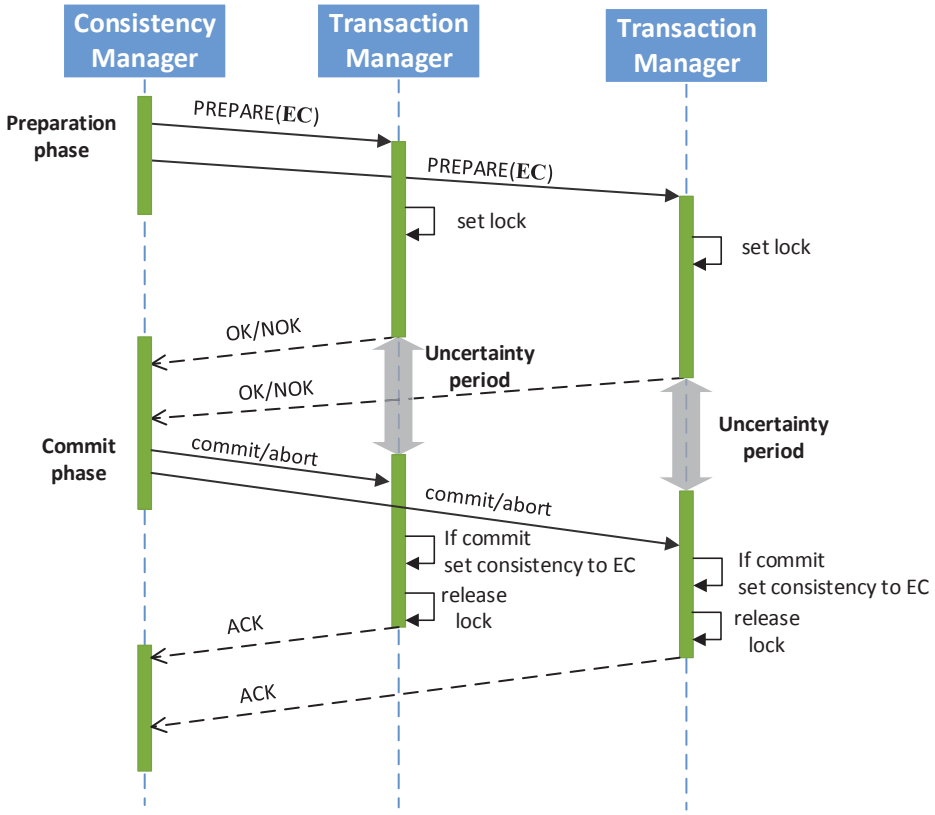


Figure 6.9: Reconfiguration to the EC consistency level.

the lock has been set, will be added to a wait queue. Once the currently running transactions have finished their execution, the consistency level is adjusted and the `prepare-ack` message is set to the *ConsistencyManager*. If all sites send positive votes, *ConsistencyManager* will send a commit, otherwise an abort. In the reaction to the decision, sites will release the locks and resume transaction execution with the new consistency (EC) levels in case of a commit, or with the old ones in the event of an abort (1SR).

- If there is an adaptation of the consistency level from EC to 1SR, then a reconciliation of sites is necessary to allow 1SR transactions to run on a consolidated data state. As already described, the reconciliation does not mean that already existing inconsistencies that were added by EC transactions are removed, but that 1SR transactions are guaranteed to get the most recent state independently of the site they are executed. In its current version, C^3 implements a stop-and-copy approach for the reconciliation of the sites. We have developed an on-the-fly and on-demand approach for the site reconciliation that can be found in the Appendix A. However, it is currently not implemented as part of C^3. The reconfiguration process differs from the process when consistency is adjusted to EC only in the activities that are executed before the `prepare-ack` message is send to *ConsistencyManager* (Figure 6.10). Now, each site will start one single distributed transaction that is coordinated with 2PC for pushing all modified objects by EC transactions to all other

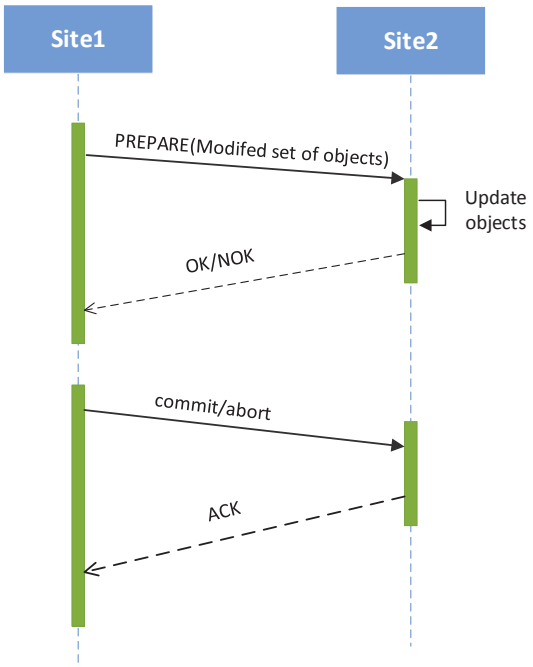


Figure 6.10: Site reconciliation: s_1 (coordinator) pushes its modified Objects to s_2. The pushed data contain the timestamp for each modified object that allows the sites to decide on the winning values based on TWR.

sites. If the distributed push transaction fails at one site, then that site will vote negatively. As a consequence the reconfiguration process is aborted; otherwise, the reconfiguration is committed.

6.3.2 Cumulus Reconfiguration

In step 4 of the workflow depicted in Figure 5.10, new partitions are proposed, which will be applied if they generate a benefit (Equation (5.22)). As a consequence, the system must accordingly be reconfigured.

The reconfiguration process in Cumulus consists of the migration of data objects to the new locations (sites) and the update of the routing information. Two aspects need to be treated with care to guarantee proper behavior. First, the reconfiguration should be synchronized with the execution of user transactions. Second, the distributed reconfiguration should be atomic even in the presence of failures as in contrary, different sites may have different views on the partition configuration, and this may lead to unnecessary and expensive forwarding activities of transactions to supposed locations. In the worst case, more than one site might become responsible for an object which might lead to inconsistent data (Figure 6.11). This last aspect is critical and should be avoided as Cumulus must ensure 1SR correctness to applications.

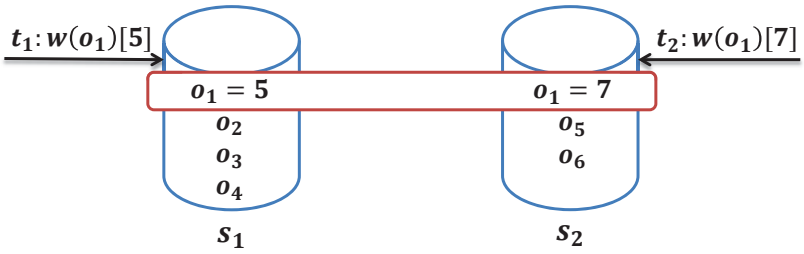


Figure 6.11: Data inconsistencies as the consequence of both sites considering themselves responsible for o_1.

Cumulus implements an *on-the-fly and on-demand* reconfiguration approach (Algorithm 8), which works as follows. Once, in step 5 of the workflow (Figure 5.10), a new partition configuration has been generated, it is sent to all *TransactionManagers*. Based on the received configuration, the *TransactionManagers* locally calculate a *change-set* consisting of triples $\langle o_{id}, currLoc, newLoc \rangle$ which encompasses the id o_{id} of the object o_i to be migrated, its current location ($currLoc$), and the new location ($newLoc$), i.e., the *TransactionManager* that will be responsible for the object after the reconfiguration. The *PartitionManager* coordinates the entire reconfiguration process as follows (Figure 6.12):

1. In the first step, the *PartitionManager* will send a `prepare` message containing the new configuration to all *available TransactionManagers*. The message denotes at the same time the reconfiguration incentive.

2. The *TransactionManagers* that receive the `prepare` message will locally set a `lock` and, after the successful lock, add all incoming transactions to a wait queue. The new change-set will be calculated and merged with the old change-set once the execution of currently running transactions has finished. This is necessary due to the on-the-fly and on-demand reconfiguration which may lead to situations in which some of the objects from the previous reconfiguration events have not been migrated to the designated locations. A merge does not immediately delete the old change-set. It will be deleted, i.e., replaced by the merged set only at commit.

   A successful merge denotes the end of the preparation phase which is indicated by a `prepare-ack` message sent to the *PartitionManager*. To avoid different views on the configuration, the *PartitionManager* will collect the `prepare-ack` messages from the *TransactionManagers* and send a commit once all sites have positively voted. Clearly, during this phase, the sites are uncertain and may block in case of failures, which is an inherent property of 2PC.

3. If a *TransactionManager* receives a commit, it will set the new merged change-set as its active set, will release the lock and resume transaction execution. Otherwise, it will discard the change-set and continue transaction execution using the old change-set.

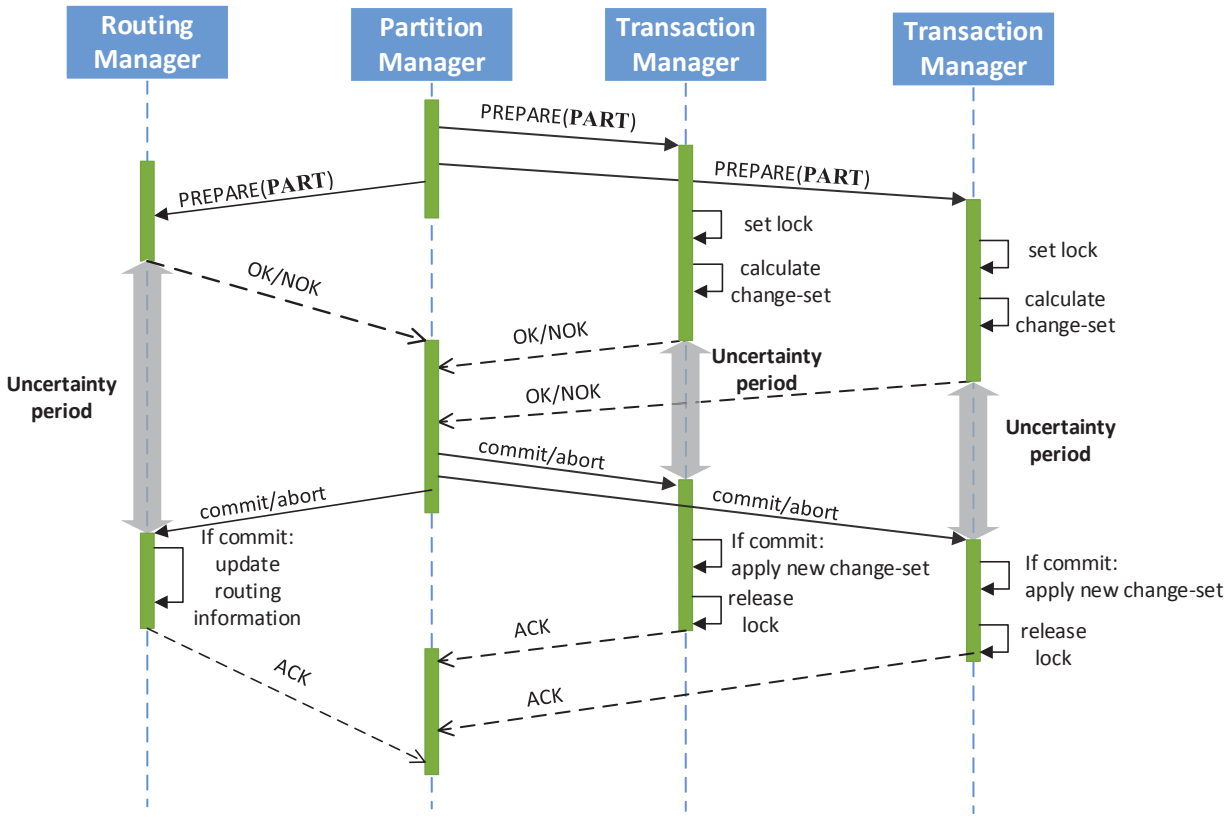


Figure 6.12: Cumulus reconfiguration.

Change-Set Handling

As objects are migrated only when accessed by transactions (on-demand), a mechanism has to be in place to handle the case of non-migrated objects during subsequent reconfiguration events. This can be done easily: if an object is still present in the old change-set, the current location of the object will be the location specified there and not the one in the new configuration. Thus, the change-set takes over the current location but updates the target location with the new location from the new partition configuration. This allows postponing the migration of an object even across different reconfiguration events.

The overhead introduced by the migration step in the on-the-fly approach results from distributed write transactions across all sites which either have the up-to-date data or need to update their data. Thus, all transactions which migrate data will be distributed update transactions, regardless of their true nature. As the migration must only take place once, this overhead will decrease over time.

Figure 6.13 depicts an example of merging the old and new change-set at s_1. As it can be seen, the *newLoc* of the new change-set will replace the *newLoc* of the old change-set. All entries with $currLoc = newLoc$ in the new change-set are not added to the merged set. Clearly, it is possible to execute the reconfiguration by first moving objects to the designated locations based on the old change-set and then calculating the

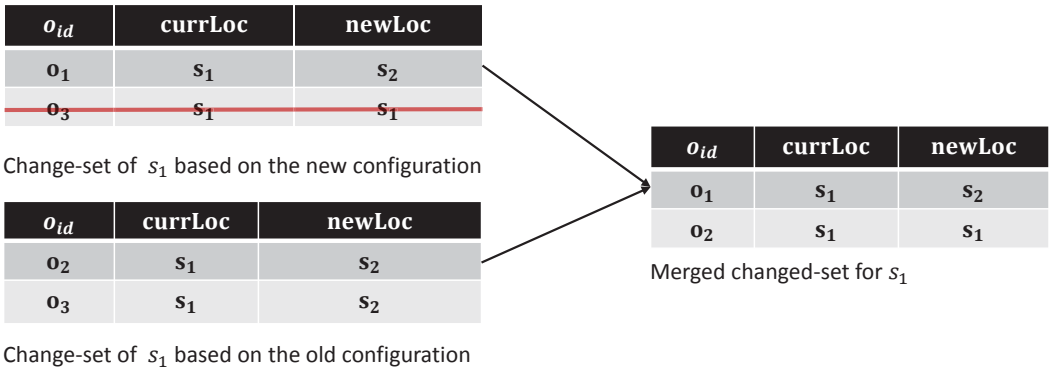


Figure 6.13: Merging of old and new change-sets.

new one. However, such an approach would lead to many objects being moved around and would increase the overhead of the reconfiguration.

Transaction Execution during the Reconfiguration

Once a site has calculated and applied its change-set, each transaction to be executed must consult the change-set unless it is empty before it is executed. The goal is to migrate objects to the correct locations by appending a distributed write action for each read action, and by transforming a write action to a distributed write action (Algorithm 8). If the same object is accessed inside the same transactions by both, a read and write action, then only the write action will be transformed to a distributed write. After a successful migration, the objects will be removed from the change-set and the routing information will get updated.

Algorithm 8: On-the-fly and on-demand reconfiguration driven by user transactions.

```
Function t: Transaction is
    if ¬ isChangeSetEmpty (cs) then
        foreach o ∈ getSet (t) ∧ cs.contains(o) ∧ cs.currLoc(o) <> cs.newLoc(o) do
            if o ∈ getWriteSet (t) then
                setIsWriteDistributed (o, TRUE) ;
            else
                appendDistributedWriteForObject (t, o) ;
```

Failure Handling

In case a site fails then the data hosted at that site is not available and all transactions accessing that data will be aborted. Sites can also fail during the reconfiguration and the only critical point, in which the system may become unavailable if a site fails, is during the uncertainty phase.

Once the new change-sets have been calculated, then during a migration of objects from one site to another, the target site may fail. In that case, the distributed write will fail, and so the change-set update will fail. The transaction will not be aborted but restarted and served from the local data. Once the failed site recovers, the migration will be restarted.

If a failed site did not receive the new configuration, at recovery it must first contact the *PartitionManager* and request the current configuration. If the current configuration does not match the local configuration, the site must calculate the new change-set, merge it with old one, and only after having executed these steps successfully can become operational.

On-Demand vs. Stop-and-Copy Reconfiguration

As already described, the Cumulus' on-demand approach incurs an overhead for single user transaction. However, compared to the stop and copy approach [EAT+15], it has considerable advantages for the overall availability, as it avoids situations in which high arrival rate of transactions fill-in the wait queues and lead to an explosion in response time and possible system instability [LZGS84]. In the stop and copy approach, the system remains unavailable during the entire reconfiguration, as all objects will be migrated to the new locations. This increases the lock duration at the sites, during which the transactions will be added to a wait queue. It needs some time after the reconfiguration for the response time of the transactions to stabilize, especially if the arrival rate of transactions during this phase is very high.

The reconfiguration process can be extended to consider additional aspects such as the overall load of the system. In high load situations it might be more beneficial to postpone the reconfiguration, as, although it might be advantageous with regards to the reduction of distributed transactions, the additional steps may destabilize the system. We consider this aspect as part of future work.

6.3.3 QuAD Reconfiguration

The reconfiguration process in QuAD consists of the quorum configuration, and the optional update of slave sites. It is crucial that the reconfiguration process is done in consistent manner, as otherwise the consistency of data may be violated. Two aspects need to be treated with care. First, all sites should have a consistent view on the quorums to avoid that the intersection property is violated and with that also the consistency of data provided to transactions. Second, as described above, the role of the sites may change, i.e., slave sites may be promoted to cores and cores demoted to slaves. It must be ensured that promoted slave sites reconcile to provide transactions access to consistent data.

Safe Reconfiguration

Let us assume the scenario depicted in Figure 6.14, which depicts the current and targeted quorum configuration. If the reconfiguration is done in an unsafe manner, then

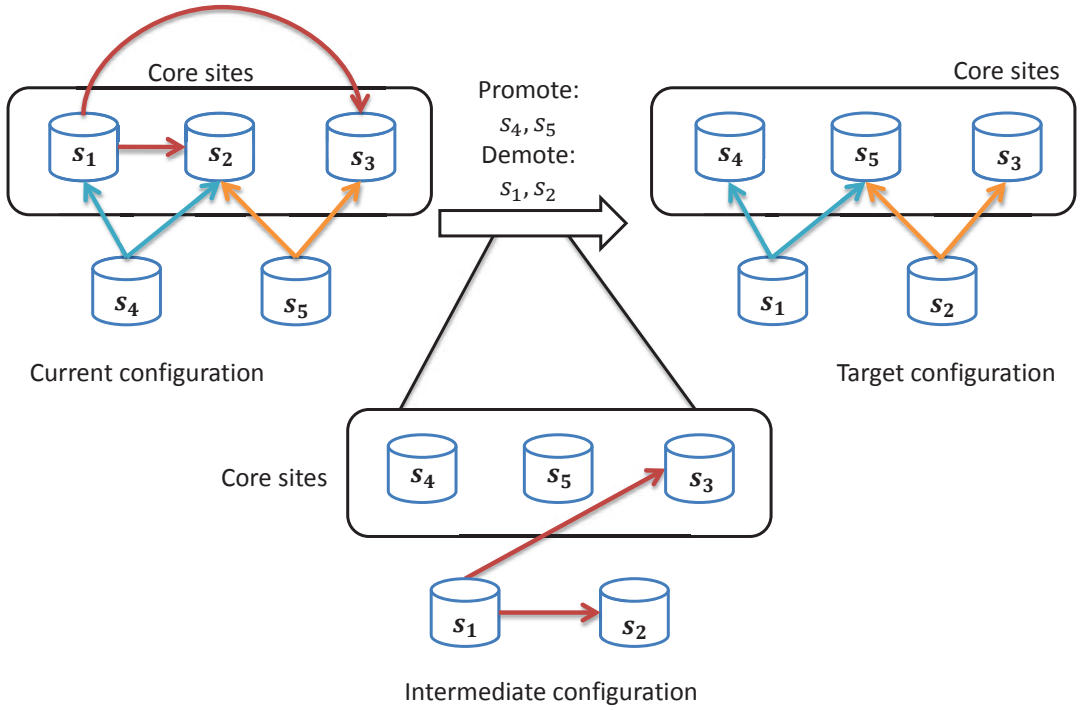


Figure 6.14: Inconsistent online reconfiguration.

certain sites may observe intermediate configurations, and this may violate the intersection property and thus consistency guarantees. In the concrete scenario, the intermediate quorum configuration leads to a situation in which s_1 considers its quorums to consist still of s_2 and s_3. This might lead to a situation in which transactions executed at s_1 and s_2 do not receives the newest updates from s_1, which is a clear violation of the 1SR consistency model.

QuAD implements a stop-and-copy approach based on 2PC that is coordinated by the *QuorumManager*. The reconfiguration works as follows (Figure 6.15):

1. In the first step, the *QuorumManager* will send a `prepare` message to all *TransactionManagers* containing the new quorum configurations. This step denotes the incentive for reconfiguring the quorums.

2. Once a site receives the new configurations, it will set a `lock`, and add after that all incoming transaction to a wait queue. Once the execution of currently running transactions has been finished, each site will apply its new quorum. If a slave site becomes a core site, it must reconcile, and respond with `prepare-ack` only after a successful reconciliation. If for any reason a slave site fails to reconcile it must negatively (`prepare-no`) respond to the *QuorumManager*.

3. *QuorumManager* will collect all answers, and send the decision to the sites, which is in response to that will release the locks and resume transaction execute. The new configuration is set to active, only if all sites vote with yes, otherwise the old configuration remains active. This means that although the slaves execute

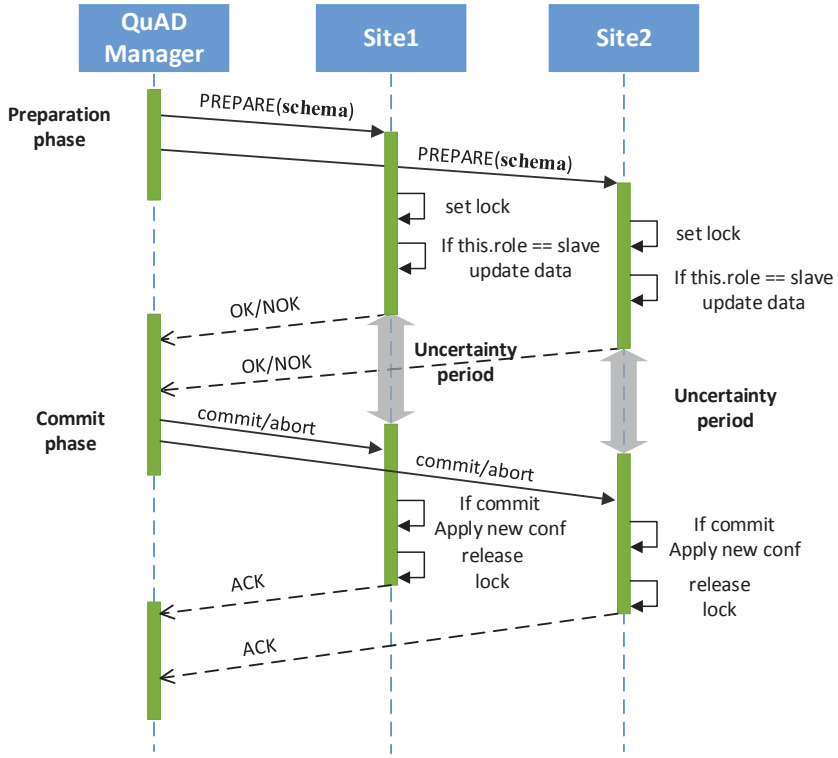


Figure 6.15: QuAD reconfiguration.

expensive update actions, it might be that they remain in the old role. From the correctness point of view this does not harm. However, it generates unnecessary costs. QuAD will tackle with this issue as part of future work.

Site Reconciliation

We distinguish between site reconciliation during the promotion/demotion without failures in reaction to changes in the site properties, and during the promotion of one or more slave sites to core sites in reaction to core site failures. The reconciliation is an activity that is executed as part of the reconfiguration workflow after setting the lock and before sending the vote message to the *QuorumManager* (Figure 6.15).

Each site in QuAD manages a *difference object (DO)* that contains the latest timestamp for each object that has been modified since the last quorum reconfiguration. The *DO* of a site s_j is a set of tuples: $DO(s_j) = \{\langle o_{id_a}, \tau(o_{id_a})\rangle, \cdots, \langle o_{id_z}, \tau(o_{id_z})\rangle\}$, with o_{id_a} denoting the id of the object o_a, and $\tau(o_{id_a})$ its latest commit timestamp. Notice that the *DO* is maintained in memory and does not need to fulfill any durability constraints as its content can be reconstructed from the application data.

First, consider the promotion of slaves to cores as a reaction to load changes depicted in Figure 6.16. The demoted core sites s_2 and s_3 will multicast their *DOs* to slave site s_1 that is to be promoted to a core site. s_1 will merge the received *DO*s with its local *DO* as follows:

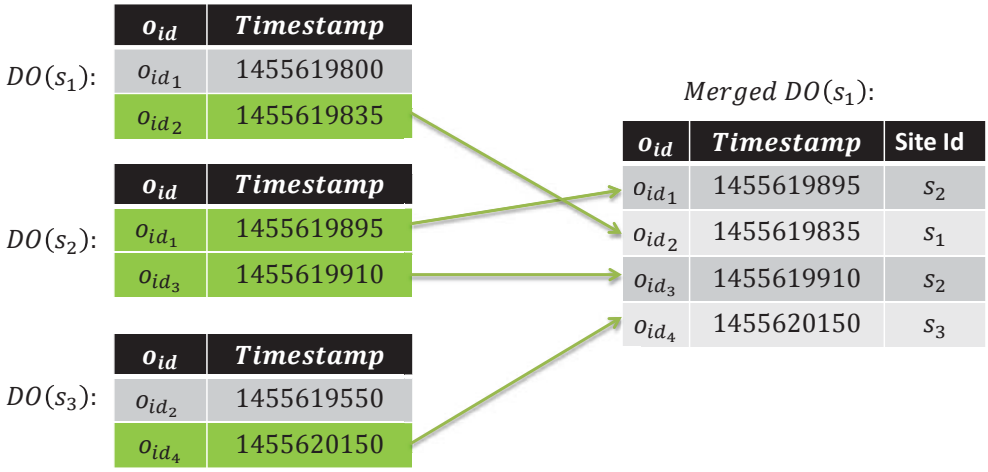


Figure 6.16: QuAD: Merging of *DO*s at s_1.

- Add each object available in one of the *DO*s to the merged DO together with its timestamp and the id of the site to which the *DO* belongs.
- If an object with the same object id is contained multiple times in the merged DO, then keep only the entry with the highest timestamp, and drop out the others.
- Remove all entries with the site id equal to the local site id.

Once the *DOs* have been merged and cleaned-up, s_1 will initiate a reconciliation with the goal of updating the objects contained in the merged *DO*. QuAD supports the stop-and-copy approach, in which the promoted slave will send a batch request to the demoted core site for pulling all objects in the merged *DO* that have the id of that demoted core site. Once a slave has successfully updated its data, it will send a positive vote to the *QuorumManager* (Figure 6.15) and reset its *DO*. The *QuorumManager* will take a commit decision only when all slaves to be promoted have sent positive votes. This approach of updating all data has the drawback that it decreases system availability due to a possibility high update overhead, especially if the set of data that has been modified between two reconfiguration events is big. However, once the reconfiguration has finished, the system is ready to serve transactions based on the new quorums. We plan to implement an on-the-fly approach that will only pull those objects accessed by transactions on demand [EDAE11, EAT+15].

In the case of core site failures, the corresponding number of slave sites will be promoted to cores sites. Since the failed core sites contain the data updated by the slaves, and these core sites have failed, the slave site to be promoted need to synchronize with all slave sites. This ensures that site to be promoted contains the data of all slaves. However, we need to ensure that it also contains the core site data. It is sufficient that they synchronize with a single core site, as the core writes behave according to the write-all approach. If also slaves have simultaneously failed, then the promoted slave site must synchronize with the majority of cores.

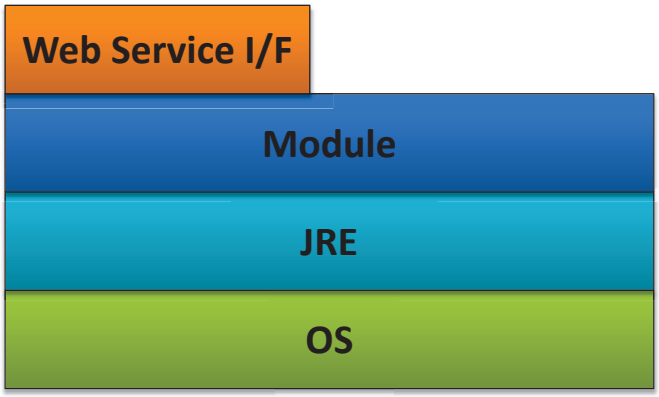


Figure 6.17: The software stack of modules.

QuAD tolerates up to $|CS| - 1$ simultaneous core sites failures, with CS denoting the set of core sites, under the assumption that no slave site fails at the same time. It remains available if the majority of cores is available independently on the number of failed slave sites.

In the case of slave site failure, QuAD will not take any immediate actions as anyways the quorums will be adapted during the next reconfiguration period. From the viewpoint of QuAD, a failed slave site is simply reduced processing capacity and has no further impact to the overall guarantees. However, core site failures reduce the availability of the system. Currently, QuAD will try to keep the number of core sites by promoting slave sites to cores. It is clear that if no sites are available for a promotion that QuAD cannot provide the desired availability. We assume a system model in which the creation or deletion of sites is outside the QuAD control, and is steered by dynamic and cost based replication protocols such as the one defined in [BPA10].

Sites Joining

If a new site joins the system, it will become a slave and will be assigned to a quorum consisting of core sites according to the cost matrix that was created during the last quorum construction. The recalculation of quorums is postponed to the next reconfiguration period, as then enough information might be available for determining the score of the sites. However, if the site joins immediately before the new period starts, then it is still not possible to determine its score so that it will remain a slave. QuAD requires each site to run at least one period before it is considered for the scoring, otherwise it will be labeled as a slave.

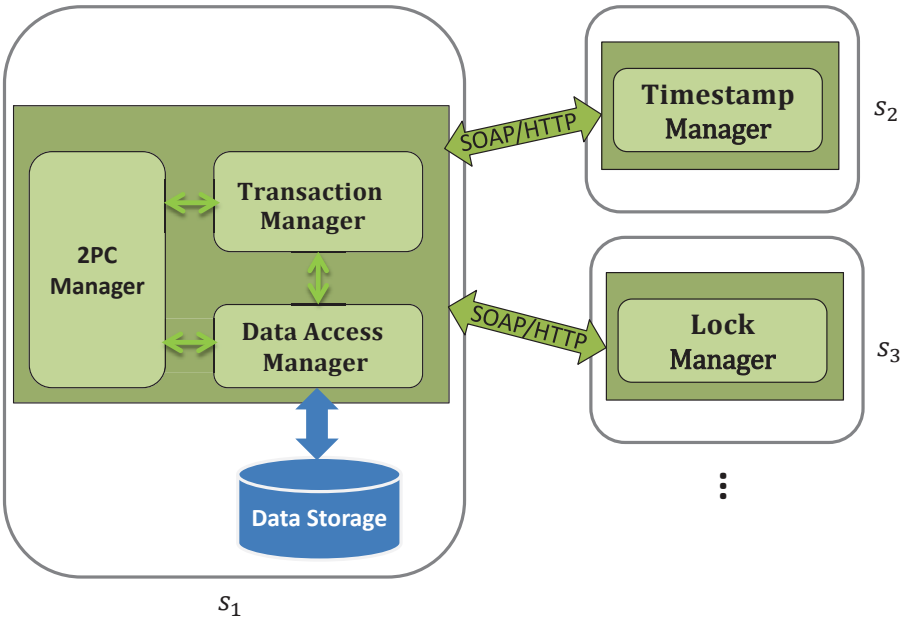


Figure 6.18: Deployment architecture.

6.4 Software Stack of Modules and the Deployment Architecture

The modules are implemented using the Java programming language, and each of the modules provides a Web Service Interface (Figure 6.17).

A CCQ deployment consists of a set of sites[3] that host the *TransactionManager* modules and the storage for managing the application data, and a set of helper sites that provide functionality usually accessed only withing the DBS, such as the *LockManager* or *TimestampManager*. Helper sites may also contain a persistent storage for managing their metadata. For example, the *LockManager* may store the mapping of locks to transactions persistently to support recovery in case of failures. Modules that are deployed at different sites communicate with each other via Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP). The modular architecture and the Web Service interface provided by the modules allow for different physical deployments. Thus, it is possible to deploy each single module on a different site and access it via SOAP/HTTP, but also to collocate different modules to a single site. In the later case, modules would communicate via local operation invocations (Figure 6.18). The centralized deployment of modules avoids coordination between them, which is in sharp contrast to a distributed deployment. In the later case, the coordination overhead, which is a consequence of the transparency demands (e.g., one-copy view) that applications have towards a DBS, may have a significant impact to the overall performance. This impact is strengthened even further if one considers the overhead introduced by the SOAP/HTTP

[3]In general term a site does not map to a machine, but simply defines the boundaries to the functionality that belongs together. However, in context of our discussion a site is a synonym for a machine.

protocol regarding bandwidth consumption. Despite its high network overhead compared to low-level communication protocols, such as sockets, SOAP/HTTP hides the low-level communication details from the programmer and decouples the communication between the modules from the underlying implementation environment.

7

CCQ Evaluation

In this Chapter, we describe the experimental results that validate the concepts of the CCQ protocols. First, we summarize the main concepts of TPCC and EC2, which are the main building blocks of our experiments. The former defines basis of the data model that is used by the experiments and the later provides the computing resources for running the experiments. Moreover, we specify the basic experimental setting that applies to the evaluations of all CCQ protocols. Second, we provide definitions that apply to the evaluation of a specific protocol together with a concrete specification for each experiment. And third, we summarize main results and provide a critical discussion on possible extensions for the experiments.

The goal of the C^3 evaluations is to examine the saving in monetary costs when a workload is run with C^3 compared to the costs generated when the same workload is run with the predefined and fixed 1SR and EC consistency levels. We will define a set of workloads that differ in terms of inconsistency and consistency costs, and compare C^3 to 1SR and EC. Furthermore, we will also define workloads consisting of different transaction classes, and evaluate the ability of C^3 to determine the most cost-efficient consistency level for each of the transactions classes inside the same workload. In overall, C^3 should lead to a considerable cost improvement compared to 1SR and EC. The adaptive adjustment of consistency at runtime should also lead to a performance speedup compared to 1SR. The response time of transactions when executed with EC define the lower bound that can not be further improved, and usually, if C^3 is deployed with the goal of saving costs, it will in overall incur higher overhead for transactions compared to EC. However, if the performance is the primary goal, then, by setting inconsistency cost to zero, C^3 will execute all transactions with the EC consistency model. In that case, as C^3 will not perform any workload analysis, it should only generate minimal additional overhead compared to EC, which mainly incurs from the collection and management of meta data.

The purpose of Cumulus is to avoid or reduce distributed transactions by generating partitions tailored to the application workload. Workload analysis, i.e., the separation of significant from the noisy patterns, plays a crucial role for both, the quality of the partitions and the processing overhead generated for the definition of the partitions. In order to evaluate the importance of workload analysis, and the ability of Cumulus

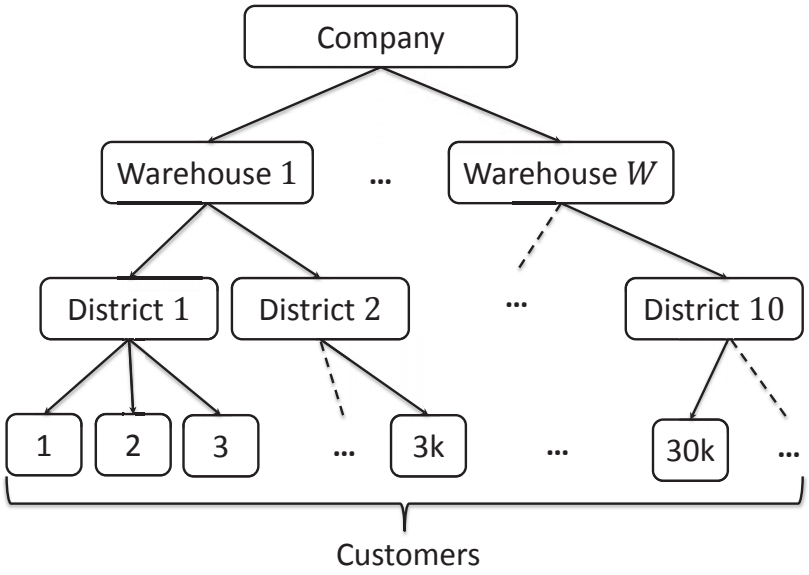


Figure 7.1: Relationships between warehouses, districts and customers in TPCC.

to detect significant patterns in a workload, we will define a set of workloads with a varying noise in them. We will then run these workloads with Cumulus, and a similar graph-based partition protocol that considers the entire workload, i.e., does not pre-process the workload. Cumulus should considerably outperform the approach without workload analysis in terms of distributed transaction. This can be explained by the fact that noise represents access patterns that will occur infrequently in the future. Thus, partitions tailored to noisy patterns will generate only low benefit for future workloads. The adaptive behavior of Cumulus will be evaluated using the a set of workloads that differ in terms of significant patterns. We will exchange the workloads at runtime, and expect Cumulus to detect changes in the significant patterns and correspondingly adapt its partition configuration.

For the evaluation of QuAD, we will simulate single-data and multi-date center deployments with varying site properties, and compare the performance gain of QuAD's approach for constructing quorums to approaches that neglect the site characteristics. We expect QuAD to considerably outperform the other approaches, as it strives towards avoiding weak sites from the read and commit paths of transactions. It is not the number of sites consisting a quorum that limits the performance, but the load and network distance between the sites. Without a proper control mechanism, such as the one included in QuAD, when quorums are defined, a set of very powerful core sites may be included in far too many quorums and become the performance bottleneck. We will compare the balanced quorum construction approach of QuAD to a version that does not consider the balancing, and expect the balanced approach to become more advantageous as the system load increases. Similar to C^3 and Cumulus, we will evaluate the adaptive behavior of QuAD by modifying the site properties at runtime, and compare the results to those of a static QuAD that does not adapt the quorums. We expect the benefit of adaptiveness to be higher than its cost leading in an overall performance gain for the applications.

7.1 TPCC

The TPCC benchmark models a wholesale supplier, that manages and sells items. A wholesale supplier has a number of geographically distributed sales districts and warehouses (Figure 7.1). Each warehouse maintains specified stock of items sold by the company, and each district serves a number of customers [TPP]. The TPCC data model consists of nine database tables, namely `Warehouse`, `District`, `Stock`, `Item`, `Customer`, `Order`, `History`, `New-Order` and `Order-Line`.

The TPCC benchmark models a wholesale supplier, that manages and sells items. A wholesale supplier has a number of geographically distributed sales districts and warehouses (Figure 7.1). Each warehouse maintains specified stock of items sold by the company, and each district serves a number of customers [TPP]. The TPCC data model consists of nine database tables, namely `Warehouse`, `District`, `Stock`, `Item`, `Customer`, `Order`, `History`, `New-Order` and `Order-Line`.

TPCC specifies following transaction types with their frequencies specified in parenthesis [TPP, TPK+13]:

- `NewOrder` (45%): it inserts a new order into the database.
- `Payment` (43%): it makes a payment on an existing order.
- `OrderStatus` (4%): it checks the shipping status of an order.
- `Delivery` (4%): it selects the oldest undelivered order for each warehouse and sets them to shipped.
- `StockItem` (4%): it joins order items with their stock entries for reporting purposes.

For the purpose of our experiments we have implemented additional transaction mixes that allow us a more fine-grained analysis of the CCQ protocols. The predefined frequency of the TPCC transactions is used as basis to determine their frequencies in the new mixes, which are defined as follows:

- **Read-only** mix consists of `OrderStatus` and `StockItem` transactions.
- **Write-only** mix consists of `NewOrder`, `Payment` and `Delivery` transactions.
- **RW8020 mix** consists of 80% read-only and 20% update transactions.
- **RW5050 mix** consists of 50% read-only and 50% update transactions.

7.2 AWS EC2: Amazon Elastic Compute Cloud

EC2 is Amazon's IaaS that provides virtual computing resources known as *instances* [Ama]. An Amazon Machine Image (AMI) is a template that contains all pre-configured software, such as the operating system, database, application server and others, that can

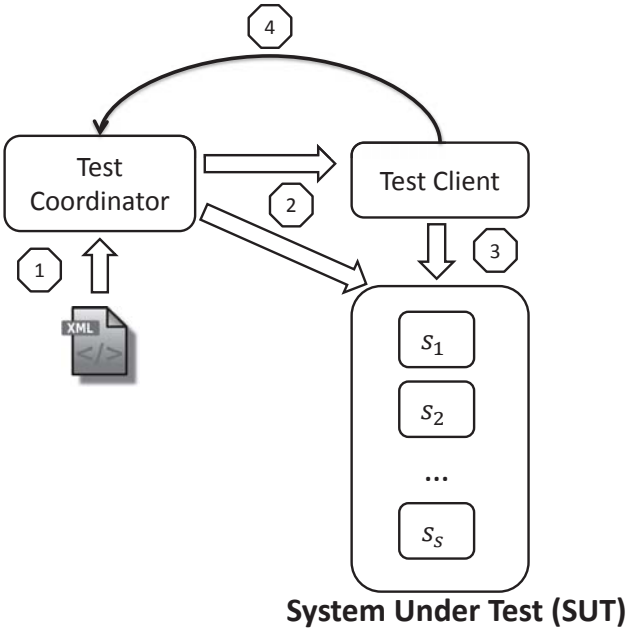


Figure 7.2: Overview of the test infrastructure setup.

be used to lunch EC2 instances. An instance *type* defines the hardware configuration, and hence the cost that customers pay on the hourly basis. Multiple instances of the same type lunched from the same AMI may exist. They would all share the same hardware and software configuration.

Amazon hosts its resources in multiple geographic locations known as *regions*. Regions are isolated from each other, and per default AWS does not replicate resources across regions. A region is composed of multiple *Availability Zones (AZs)* that are isolated from each other and connected through low latency links. This allows applications to run their instances in multiple AZs, and benefit in terms of availability from their isolation with almost no performance penalty due to the high-speed links between them. It is clear that replicating resources in different AZs does not provide the same availability guarantee as the replication in different regions, as if a region goes down, then all AZs within that region also become unavailable. Moreover, if the application needs to serve requests from the different regions, client proximity can only be satisfied if resources are also replicated at the region closest to the client. An AMI always belongs to a region, whereas instances belong to an AZs inside the same region as the AMI that was used to lunch the instances.

7.3 Basic Experimental Setting

All evaluations use the same workload prediction model based on EMA as described in Section 4.2.3. The choice of the smoothing factor α is is essential for accurately predicting the workload. In order to determine the most suitable value for α, we apply Equation (4.16) using $\alpha \in \{0.1, 0.2, ..., 0.9\}$, and then use that value of α that has the lowest overall Mean Absolute Deviation (MAD). Other approaches, such as expectation maxi-

Parameter	Description
#site	Defines the number of sites in the system.
r/w ratio	Defines the ratio of read and update transactions in the workload.
#accesspatterns	Defines the number of different access patterns.
noiseLevel	Defines the percentage of noisy transactions in the workload.
#workers	Defines the number of worker threads that generate transactions.

Table 7.1: Basic setup parameters.

mization and maximum likelihood [HS98], for determining the best value of α are considered part of a future work.

The infrastructure for running the experiments is created by the `TestCoordinator`, which receives the definition of the experiment to be executed in the form of Extensible Markup Language (XML) files (Figure 7.2). The configuration file contains basically the values for the parameters defined in Table 7.1. An example configuration file is provided in the Appendix B.

The `TestCoordinator` will deploy a number of sites as defined by the *#sites* (Step 2 in Figure 7.2). Each site consists of an Apache Derby database version **1.9.1.0** for managing the application data, and an Apache Tomcat application server version **7.0.32** for managing the CCQ modules. The deployment of a site includes the population of the Derby database with the TPCC data, and the deployment of CCQ modules. The deployment architecture is depicted in Figure 7.3. As shown there, the deployment consists of a set of sites that are composed of the *TransactionManager*, *2PCManager*, *DataAccessManager* and the Derby storage for managing the data. The helper modules are deployed to dedicated helper sites, and are accessed only by the *TransactionManagers* over SOAP/HTTP.

The TPCC data model is defined by the following parameters. There is one wholesale supplier that has 10 districts. Each district serves 300 customers, i.e., in total there are 3′000 customers. The number of stock entries is set to 10′000.

The `TestCoordinator` will also create the `TestClient` and provide it with the necessary setup parameters that determine the workload (Step 2 in Figure 7.2). The `TestClient` runs on a separate machine then the SUT. It initiates a number of workers threads (*#workers*) that generate the desired workload by considering the specified *noiseLevel* and *r/w* mix (Step 3 in Figure 7.2). The parameter values are varied based on the purpose of concrete experiments. Each worker either creates a transaction from scratch, or retrieves one from a predefined pool of transactions, submits it for execution at a specific site, and waits for its response before submitting the next transaction.

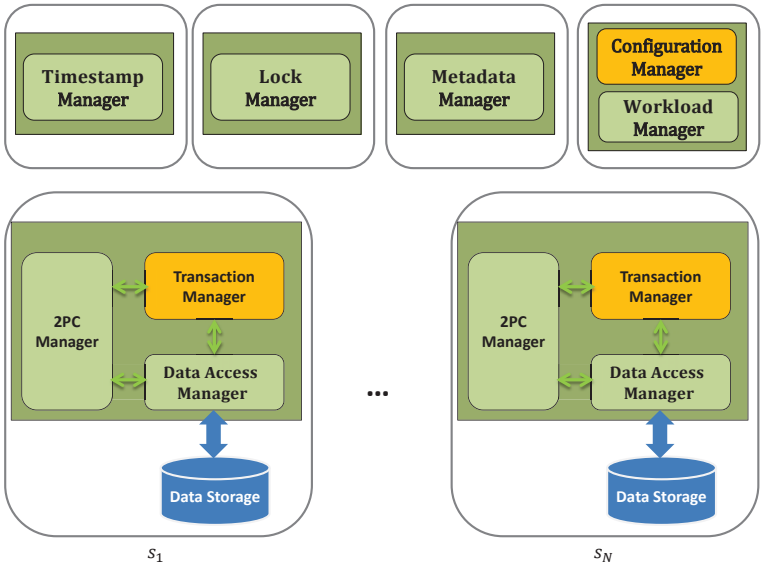


Figure 7.3: Deployment architecture used in the evaluations. The orange modules denote placeholders that are replaced during the deployment with concrete CCQ modules as described in Section 6.

All experiments are conducted in the AWS EC2 environment[1]. We have used `c1.medium`[2] as a machine type deployed in the *eu-west*[3] region, which is located in Ireland. A `c1.medium` EC2 machine consists of two virtual CPU and has a capacity of 1.7GiB RAM and 350GB of disk storage. `c1.medium` instances yield only a medium network performance, which is not exactly quantified, but some unofficial measures indicate a network bandwidth of 1Gb/s within the same availability zone [Ama].

Once a running experiment is finished, the `TestCoordinator` will download the results using the File Transfer Protocol (FTP) (Step 4 in Figure 7.2), destroy the infrastructure, and recreate it from scratch before starting the next experiment. This ensures that results are not affected by any side effects, such as available calculations from the last experiment or any locked resources, that would distort the results of the subsequent experiment.

7.4 C^3 Evaluation Results

The main goal of the C^3 evaluations is the assessment of costs generated when a certain workload is executed with C^3. We compare these costs to the costs generated when the same workload is executed using a fixed consistency level, namely 1SR and EC, which form the evaluation baseline. More concretely, we have evaluated following aspects. First, we show the ability of C^3 to choose the most cost-efficient consistency model based on application workload and application specific inconsistency costs. Second, we show

[1] http://aws.amazon.com/de/ec2/
[2] http://aws.amazon.com/de/ec2/previous-generation/
[3] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ using-regions-availability-zones.html

the sensitivity of C^3 to consistency and inconsistency costs. For that, we run the same workload with varying consistency and inconsistency costs. And third, we evaluate the ability of C^3 to handle more than one class of transactions, each class being defined by its specific inconsistency cost, and compare the generated costs to the baseline costs.

7.4.1 Definition of Transactions

For the evaluation of C^3 we have defined the following update transactions, which allow us to steer the generation of inconsistencies.

1. `Buy` transactions for buying items. It will assign the id of the customer to the item to be bought. It allows as to simulate the lost-update inconsistencies, which occur if more than one customer buys the same item. We assume that there is only one item in the stock.

2. `Update details` transaction that update the details of an item. If more than one transaction updates the details of the same item, then a lost-update occurs.

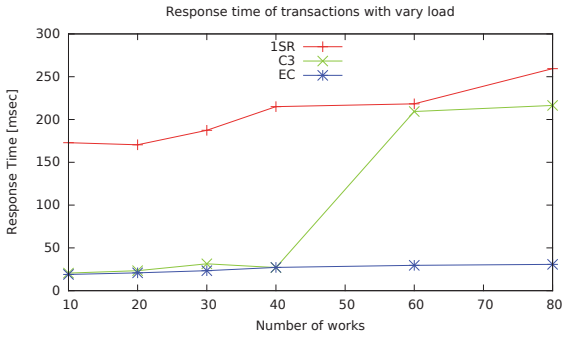
We have adapted the original TPCC model to implement the C^3 aforementioned transactions.

7.4.2 Cost-driven Consistency

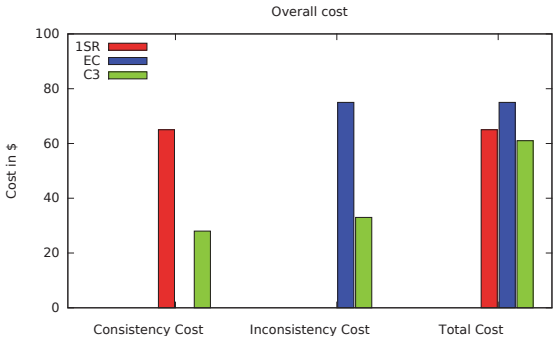
The first series of experiments compares the cost and performance of C^3 for varying workloads given specific consistency and inconsistency costs to that of 1SR and EC. The SUT consists of eight sites ($\#sites = 8$). For each of the sites, 10 workers are created that will continuously generate transactions from a site-specific pool of transactions, and submit them for execution to that designated site. The selection of transactions from the pool is done randomly. The site-specific pools are disjoint, i.e., do not contain any common transaction. Furthermore, we have defined a common pool of read-only and `buy` transactions with a $r/w = 0.5/0.5$, and a varying number of *common workers*, again assigned to a specific site, which will pull transactions from the common pool, and submit them to their own site. The goal of the common pool is to generate inconsistencies between the sites. Each run lasts for 300 seconds, and continuously logs statistics in a Comma Separated Values (CSV) file. After the current run is finished, the number of common workers is increased, and the next run is initiated. For this set of experiments, the consistency cost (i.e., the cost for executing a transaction with 1SR – $cost_{2pcmess}$ in Equation (5.5)) is set to 0.01, and the cost for a single inconsistency ($incCost$ in Equation (5.2)) to 0.03.

Expected Results

As the number of common workers increases, the number of inconsistencies should increase, and with that, also the inconsistency costs. We expect the EC consistency model to be the model of choice for C^3, as long as the number of inconsistencies remains low. However, the increase in number of common workers, should also lead to an increase in



(a) Average response time of transactions



(b) Overall costs

Figure 7.4: Cost and response time of transactions for different consistency models with a varying workload.

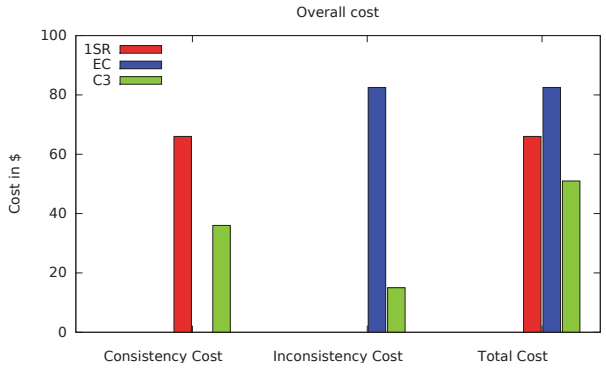
inconsistency cost, and at some point it should be cheaper to switch to 1SR. C^3 should outperform the baselines in terms of cost. Moreover, the adjustment of consistency level should also in overall lead to a considerably better transactions response time compared to that of 1SR.

Experimental Results

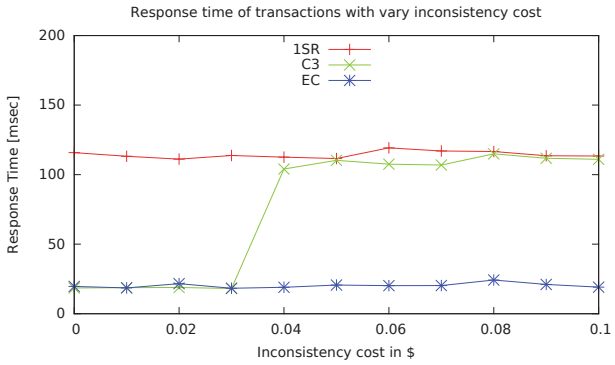
The evaluation results are depicted in Figure 7.4. As shown in Figure 7.4a, initially, C^3 will execute transactions with the EC consistency level as the expected inconsistency costs of EC are lower than the consistency cost of 1SR. However, as the load increases, the inconsistencies also increase. Starting with 60 common workers, C^3 will switch to 1SR, which is clearly visible in the increase of response time. With regards to the cost (Figure 7.4b), EC exhibits zero inconsistency cost and 1SR zero consistency cost. As C^3 adjust the consistency level at runtime, both consistency and inconsistency costs incur. However, compared to the baselines, C^3 incurs $20\% - 25\%$ less cost in total.

7.4.3 Sensitivity of C^3 to consistency and inconsistency cost

This series of experiments compares the generated cost for a workload when the consistency ($cost_{2pcmess}$ in Equation (5.5)) and inconsistency cost ($incCost$ in Equation (5.2)) are varied. This corresponds to the case, in which, either the Cloud provider adapts the cost of its resources, or the application provider adapts the cost for compensating an inconsistency. The goal is to show the ability of C^3 to account for these adaptations, and to correspondingly adjust the consistency level. For this experiment, we have used the same setup as defined in Section 7.4.2, except that the load remains constant, i.e., the number of common workers does not increase, as the goal is to analyze the sensibility of C^3 to consistency and inconsistency cost.

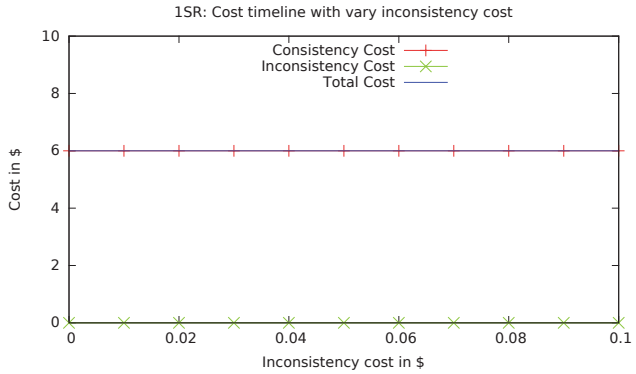


(a) Overall costs

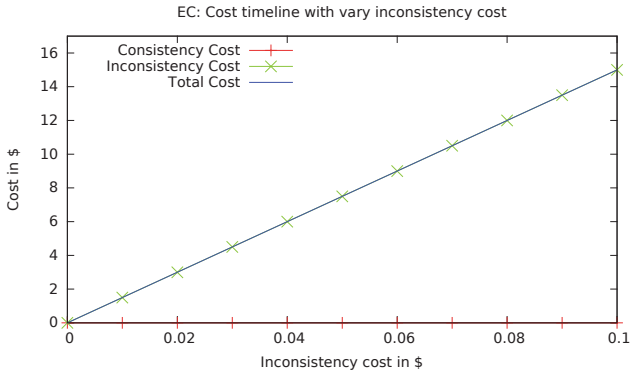


(b) Average response time of transactions

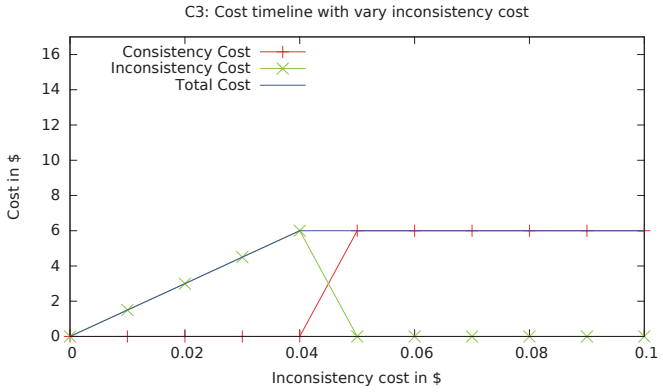
Figure 7.5: Cost and response time of transactions for different consistency levels with varying inconsistency cost.



(a) 1SR consistency level



(b) EC consistency level

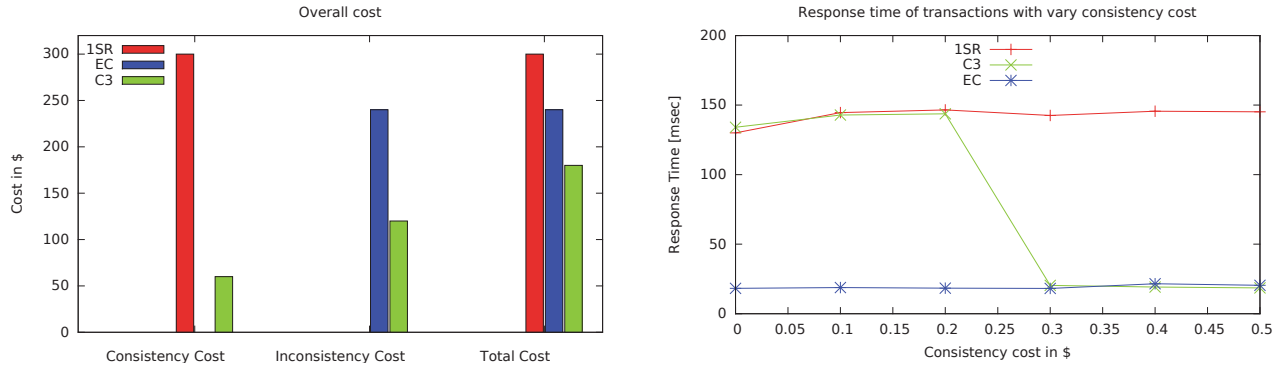


(c) C^3 consistency cevel

Figure 7.6: Cost behavior of the different consistency levels with varying inconsistency costs.

Expected Results

We expect C^3 to adjust the consistency at runtime based on the incurring inconsistency and consistency costs. In terms of generated monetary costs, C^3 should outperform,



(a) Overall costs

(b) Average response time of transactions

Figure 7.7: Costs and response time of transactions for different consistency models with varying consistency cost.

both, the 1SR and EC consistency level. The response time of transactions executed with C^3 should lie between that of 1SR and EC.

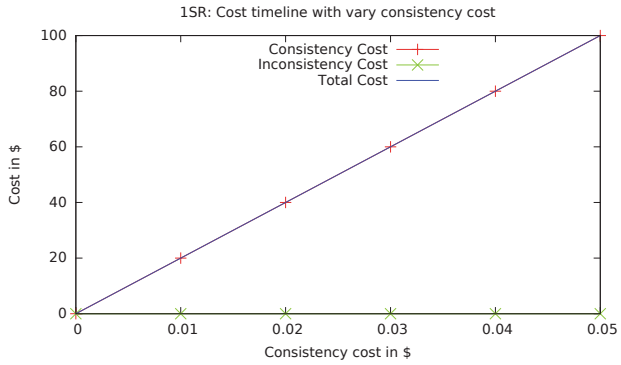
Experimental Results

The experimental results when *incCost* is varied are depicted in Figures 7.5 and 7.6, and the results for a varying $cost_{2pcmess}$ are depicted in the Figures 7.7 and 7.8. As shown there, C^3 generates in overall $15\% - 35\%$ (see Figure 7.5) less cost for varying inconsistency cost compared to 1SR and EC, and $28\% - 40\%$ (see Figure 7.7) less cost for varying consistency cost. With regards to the response time, C^3 decreases the average response time of transactions by a factor of 1.5 compared to 1SR in the varying inconsistency cost experiment, and by a factor of 1.7 in the varying consistency cost experiment.

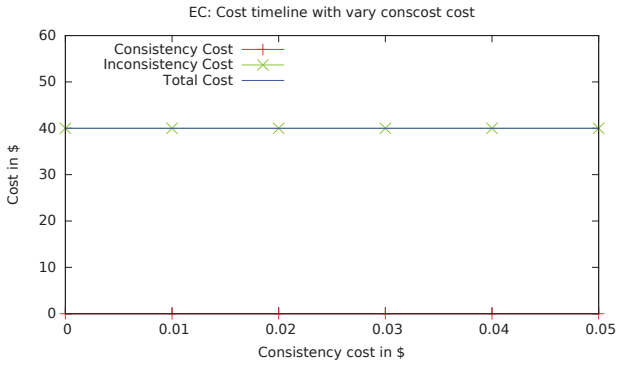
As shown in the Figures 7.6 and 7.8, C^3 will execute transactions with the cheapest consistency level, and adjust consistency once this is not the case anymore. Notice that the overall cost of an application, when executed with 1SR, is determined by the consistency cost, whereas when executed with EC by the inconsistency cost. In the case of C^3 both, consistency and inconsistency cost may occur.

7.4.4 Workload with Multi-Class Transactions

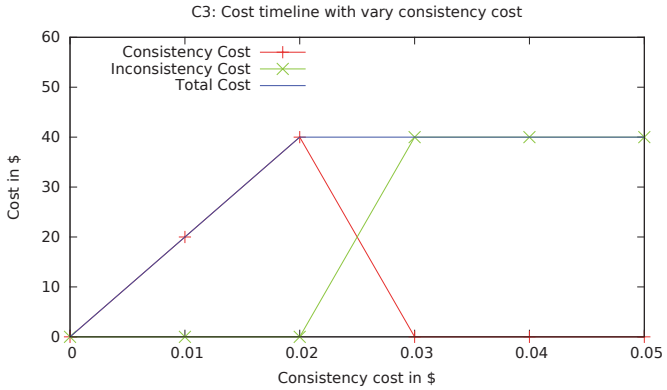
The goal of this experiment is to depict the ability of C^3 to detect and handle multi-class transaction workloads. As described in Section 5.1, common to all transactions of the same class is their inconsistency cost, i.e., they all generate the same overhead for the compensation of their inconsistencies. For this experiment, we have generated two common pools of transactions ($r/w = 0.5/0.5$). The update portion of the first pool consists of `buy` transactions and that of the second pool of `update details` transactions. Inside the same pool, update transactions are annotated with the same class and assigned the same inconsistency cost. A `buy` transaction has $incCost = \$0.03$, whereas an `update details` transaction $incCost = \$0.001$. The common pools are disjoint as the purpose of this experiment is mainly to asses the overall monetary cost, and not



(a) 1SR consistency level

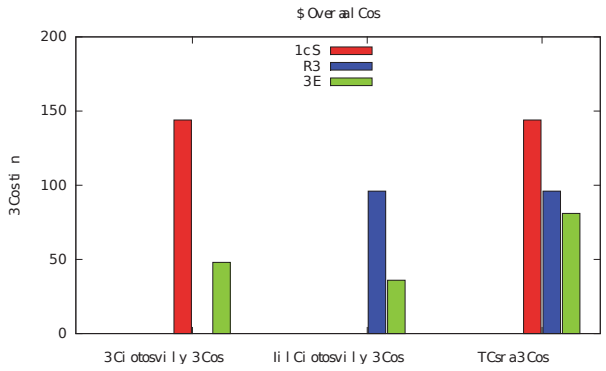


(b) EC consistency level

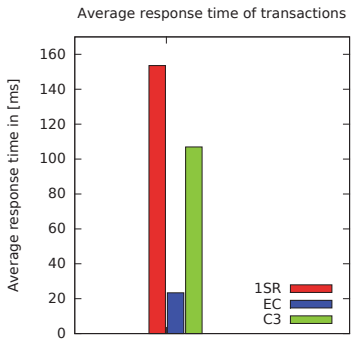


(c) C^3 consistency level

Figure 7.8: Cost behavior of different consistency levels with varying consistency cost.



(a) Overall costs



(b) Average response time of transactions

Figure 7.9: Costs and response time of transactions for a workload consisting of multi-class transactions.

the handling of transaction mixes with regards to the conflicts. Two sets of common workers are created, each of the size of forty, that will pull transactions from one of the common pools, and submit them to the sites in a round-robin manner.

Expected Results

We expect C^3 to detect the two classes of transactions based on their annotations, and extract their *incCost*. Moreover, C^3 should predict the expected cost for each of the classes and choose the most suitable consistency level for each class. C^3 should outperform in terms of monetary cost both fixed consistency levels, as minimizing the cost for each class also minimizes the overall cost. Furthermore, it should improve the average response time of transactions compared to 1SR.

Experimental Results

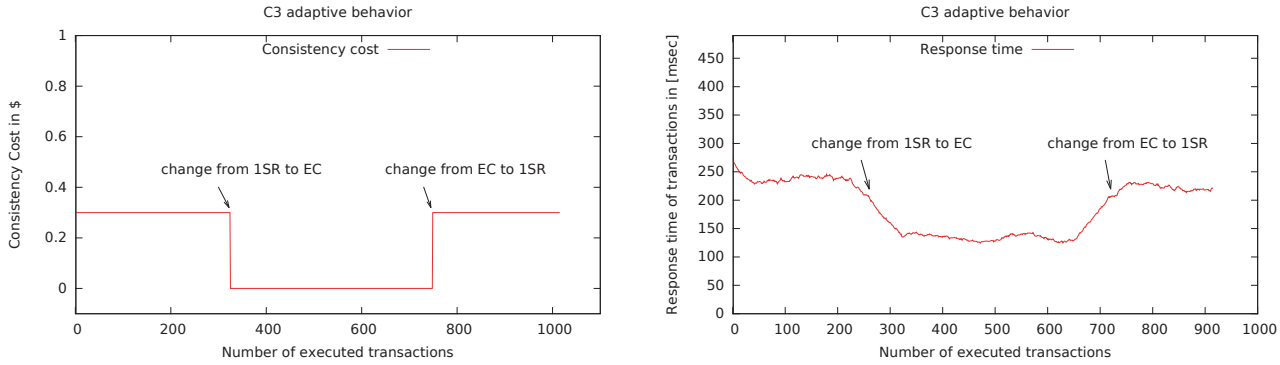
The results of this experiment are depicted in Figure 7.9. As shown there, C^3 generates less monetary cost compared to the baselines. The difference compared to the EC were rather small, as we have used a low inconsistency cost. However, compared to 1SR, C^3 generates almost 50% less cost in total. C^3 also leads to a decrease of response time compared to 1SR by almost 37%. This is explained one hand by the fact that C^3 executes many transactions with EC as that is the cost-effective consistency due to the low inconsistency cost, and on the other hand, due to the lower load generated by these EC transactions. Although there are no conflicts between the transactions of the common pools, the impact of the lower processing load is tangible in the overall performance. In case of conflicts, the impact of missing 2PC overhead would lead to an even higher performance gain, as shortening the lifetime of transactions (by removing the 2PC component) would decrease the lock duration, and with that, the resource contention.

7.4.5 C^3 Adaptiveness

In this experiment, we will assess the ability of C^3 to continuously adjust consistency level based on the workload given specific consistency and inconsistency costs. For that, we have created a set of transaction pools, consisting of update transactions only, that have different characteristics with regards to the number of inconsistencies. It is the pool size that controls the number of inconsistencies: the smaller the pool size, the higher the probability that the same transaction is submitted to different sites, and with that the higher the probability for inconsistencies. Both, $cost_{2pcmess}$ and *incCost* are set to \$0.01. A set of workers will continuously pool transactions from the current pool and submit them for execution. Each run is proceeded by a warm-up phase, which allows C^3 to gather data for the cost prediction. Workers will run for a specific duration, after which they will switch to another pool. Notice that the number of workers, i.e., the load remains constant. Each switch to another pool is proceeded by a warm-up phase, that does not appear in the statistics.

Expected Results

We expect C^3 to detect workload switches, predict the expected cost, and adjust the consistency level accordingly.



(a) Consistency costs of transactions (b) Response time of transactions

Figure 7.10: Consistency costs of transactions using C^3's adaptive consistency.

Experimental Results

The experimental results are depicted in Figure 7.10. As shown there, C^3 is able to detect workload changes and adapt consistency level at runtime. Two adjustments are visible in Figure 7.10, one from 1SR to EC and the other one in vice-versa. These two switches are also clearly visible in the response time of transactions depicted in Figure 7.10b. In this experiment, the reconfiguration overhead is determined by the communication overhead of the *ConsistencyManager* with the *TransactionsManager*s at the sites, that need to be informed about the new consistency level.

7.4.6 Summary

The conducted experiments show the ability of C^3 to adjust consistency at runtime. C^3 is able to handle different workload types by executing transactions with the most cost-effective consistency. Furthermore, it is multi-class aware, and it is able to extract costs on a per-class basis, and by that adjust consistency for each of the different classes separately. The reconfiguration process of C^3 ensures a safe adjustment of consistency at runtime and incorporates a reconciliation mechanism that applies when consistency is adjusted from EC to 1SR. Although it currently relies on manual intervention for removing existing inconsistencies, it nevertheless ensures that all sites have the same view on the data.

C^3 can be considered as an isolation level for NoSQL databases. It can, however, be also run in a mode that enforces a certain consistency level by setting the consistency or inconsistency cost to infinity. Its modular architecture allows for a seamless implementation of further consistency levels. Moreover, as it is implemented at the middleware layer, it can be used on top of different existing DBSs.

7.5 Cumulus Evaluation Results

The objective of the evaluation of Cumulus is threefold. First, we show the impact of distributed transactions on the overall system performance. For this, we compare Cumulus with a fully replicated system that uses the ROWAA approach. Second, we show the effects of the Cumulus workload analysis described in Section 5.2 on the quality of the resulting partition configuration. This is done by comparing Cumulus with a graph-based partitioning approach, such as Schism [CZJM10], that does not consider workload analysis. Third, we compare the on-the-fly and on-demand reconfiguration of Cumulus to a stop-and-copy approach [EDAE11]. Moreover, we compare Cumulus to a static protocol in order to depict the necessity of adapting to workload shifts.

7.5.1 ROWAA vs. Cumulus

The first series of experiments compares ROWAA with Cumulus in terms of distributed transactions and depicts the impact of distributed transactions to overall system performance. We have conducted three types of experiments for comparing the performance of Cumulus to that of ROWAA:

1. We have varied the r/w ratio in order to depict its impact on the percentage of distributed transactions, and also to depict the impact of the distributed transactions to performance. For that, we have created a predefined pool of 200 read-only and update transactions. The workers ($\#workers = 10$) will pull transactions from that pool according to the desired r/w ratio and submit them for execution. For each r/w ratio, Cumulus needs two runs. The first run is a warm-up phase for gathering the workload statistics that are necessary for creating a suitable partition configuration. The warm-up phase is not necessary for ROWAA. In the second run, the quality of the partitions is assessed.

2. We have conducted a *sizeup* [CST+10] experiment, in which, given the $r/w = 50\%/50\%$, *#workers* is continuously increased and the average response time is measured. For that, we have used the pool of transactions defined in the previous experiment. The goal is to show that the higher the load, the higher the impact of distributed transactions to the performance due to the high lock contention. The experiment should depict that it pays-off to invest resources in reducing or avoiding distributed transactions.

3. We have varied the *#sites* to depict that the load distribution and the collocation of objects are two competing goals. For this experiment, $\#workers = 10$ and $r/w = 50\%/50\%$, whereas the number of sites is varied.

Expected Results

Distributed transactions are expensive and incur an overhead due to the necessity of network communication. This additional overhead may considerably impact the overall performance as it increases the duration in which resources have to be kept. While in

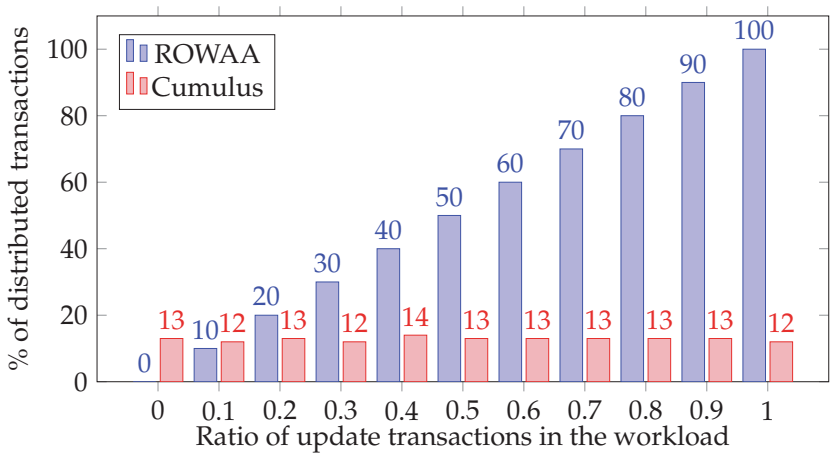


Figure 7.11: Percentage of distributed transactions.

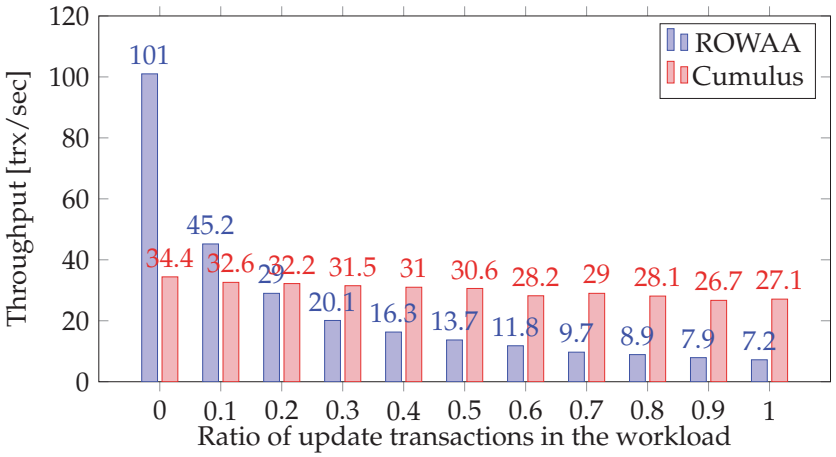


Figure 7.12: Transaction throughput.

ROWAA the percentage of distributed transactions is determined by the r/w ratio, the percentage of distributed transactions in Cumulus is determined by the quality of the partitions, and we expected it to be lower and, in the best case, independent of the r/w ratio. The reduction of distributed transactions in Cumulus should also lead to a considerable decrease of response time and increase of throughput compared to ROWAA.

As part of the introduction, we described two of the main goals of a partitioning algorithm, namely collocation of objects and even load distribution. The issue of load distribution can be tackled using two strategies. First, use existing sites and trigger repartitioning so that the load remains evenly distributed, which is of no help if the capacity limit of the existing sites is reached. Second, add additional sites and incorporate them into the partitioning process. However, load distribution and collocation are two competing goals; we expect an increase of distributed transactions with the increasing number of sites that are considered during the data partitioning.

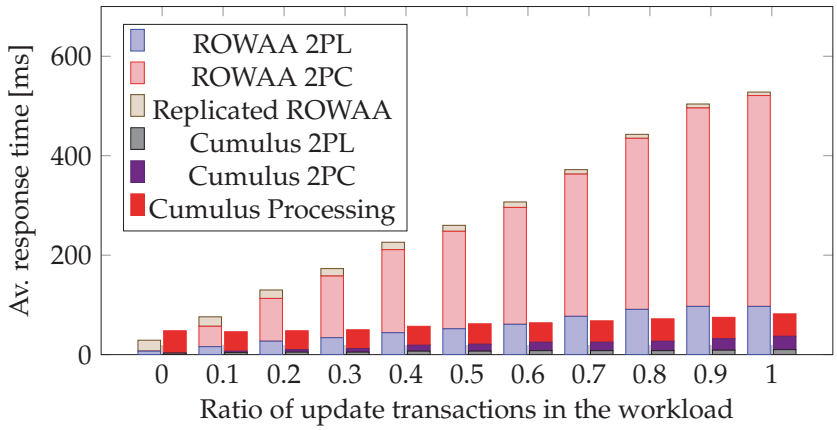
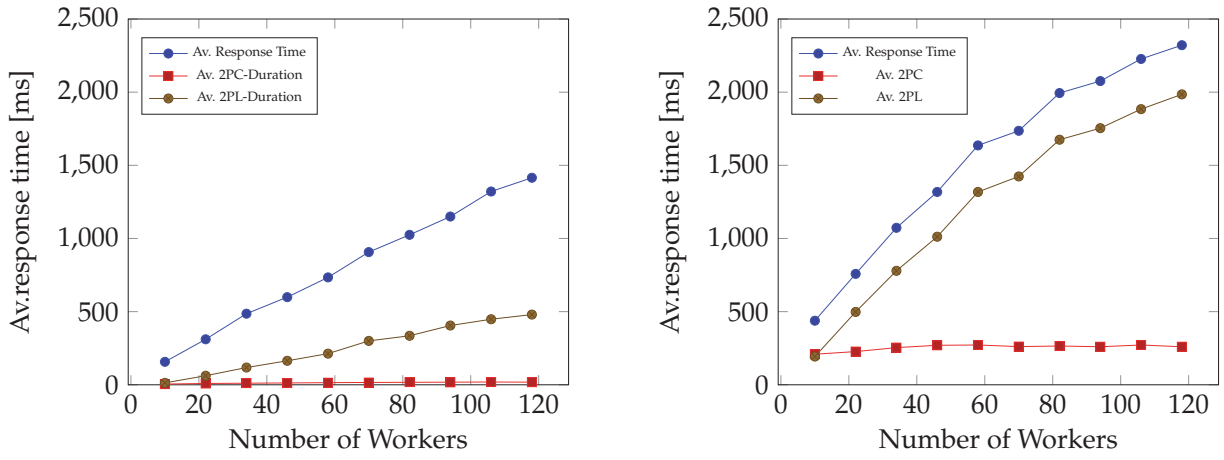


Figure 7.13: Response time of transactions.



(a) Cumulus (b) ROWAA

Figure 7.14: Comparing Cumulus with ROWAA based on a *sizeup* test.

Experimental Results

Impact of the Ratio of Update Transactions in the Workload to the Percentage of Distributed Transactions The experimental results are depicted in the Figures 7.11, 7.13 and 7.12. The results for Cumulus correspond to measurements during the evaluation phase, i.e., in the second run after the warm-up phase.

As shown in Figure 7.11, the percentage of distributed transactions in Cumulus remains more or less constant. This is in sharp contrast to the percentage of distributed transactions that occur in ROWAA, which increases with an increasing ratio of update transactions in the workload. This nicely shows that Cumulus remains agnostic to the percentage of update transactions in the transaction mix. The impact of distributed transactions to the system performance is depicted in Figures 7.12 and 7.13. As shown there, an increased number of distributed transactions leads to a decrease in the throughput and implies high response times.

ROWAA performs better only for a read-only workload, which corresponds to the expectations, as in Cumulus read-only transactions may also be distributed. This is

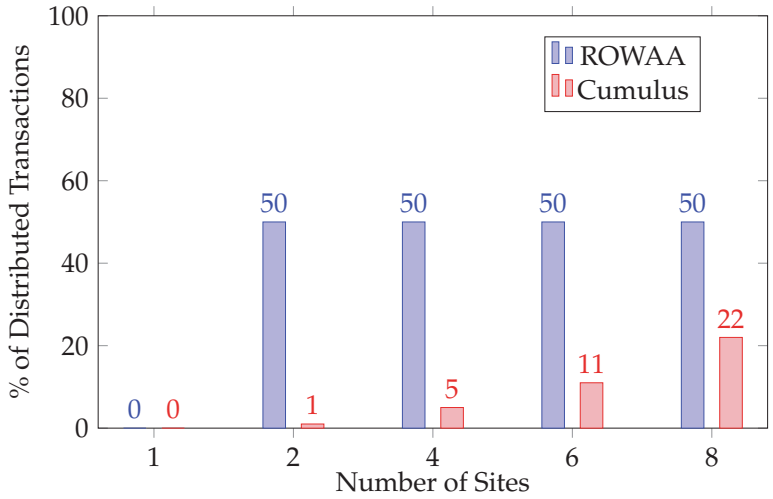


Figure 7.15: Percentage of distributed transactions with increasing number of sites.

a consequence of the Cumulus behavior that also considers load distribution, which implies object distribution.

Figure 7.13 depicts the overhead generated by each of the transaction components, namely processing, S2PL, and 2PC. The overall response time is the sum of the duration of each of these three components. The results show that the overall response time in the ROWAA setting is dominated by 2PC overhead especially for workloads with high update ratio.

Latency vs. Throughput Figure 7.14 depicts the average response time of transactions together with the overhead generated by the locking and commit phases as the load (*#worker*) increases. The results show that the response time of transactions in ROWAA increases much faster compared to Cumulus when the throughput increases. In case of ROWAA, the commit coordination costs are higher compared to Cumulus due to the higher number of distributed transactions (compare the 2PC overhead). This, in turn, leads to higher resource contention as locks need to be kept longer in case of ROWAA. With increasing workload, the contention becomes even stronger and thus significantly impacts the average response time.

Collocation vs. Load Distribution The results of this experiment are shown in Figure 7.15. As it can be seen, the more sites are added, the less collocation can be achieved, i.e., the number of distributed transactions increases with the increasing number of sites.

In summary, in a fully replicated system such as ROWAA, if strong consistency is required, the percentage of distributed transactions is determined by the ratio of update transactions in the workload. However, as data is fully replicated transactions are not bound to certain sites, and thus, they can be executed at any site. From the load balancing point of view, this is the ideal case. By partitioning the data, it is possible to influence the number of distributed transactions. However, that limits the load balancing capabilities as transactions are bound to specific sites. Thus, in order to provide scalability, both the number of distributed transactions and load distribution should be optimized by the partition protocol.

If we decompose 1SR transactions to the different components, namely the processing, locking (S2PL) and 2PC components, and map the optimization of Cumulus to these components, then we can conclude the following. Cumulus reduces the processing overhead by balancing the load, and the 2PC overhead by collocating objects that are frequently accessed together. These optimizations considerably impact the S2PL costs, which is known to be the limiting factor to the system throughput [BN96]. Thus, it pays off to "shorten" transaction lifetime and finish them as fast as possible.

7.5.2 Impact of the Workload Analysis on the Quality of the Partitions

This set of experiments evaluates the impact of workload analysis, described in Section 5.2, on the quality of the generated partition configuration. Filtering out noisy access patterns from the workload is crucial for avoiding the generation of a configuration that does generate no or low benefit for the workload. Moreover, the fewer patterns are considered, the lower the generated overhead when searching for a suitable configuration.

The experiment is set up as follows. We have defined two pools of access patterns, namely a pool containing significant patterns, i.e., patterns with a high occurrence, and another pool that contains patterns with a low frequency – denoted as noisy patterns. The transaction workers (#*workers* $= 10$) will generate a workload as a combination of the significant and noisy patterns based on the desired ratio of noise, which is varied for each experiment. For this experiment #*sites* $= 4$.

Another important feature of the Cumulus workload analysis is that only the objects accessed within the significant access patterns are used to determine the partition configuration – this is in contrast to similar approaches that incorporate all existing data objects into the generation of the partitions. We have conducted a *sizeup* (#*workers* is varied) test to depict the performance impact of the improved object (load) distribution to sites by Cumulus.

Expected Results

Cumulus incorporates a threshold-based workload analysis that can extract significant access patterns and generate a configuration that is tailored to the needs of the significant patterns, as that generates the highest benefit in terms of reduction in distributed transactions. Our expectations towards the results are thatCumulus can distinguish between significant and noisy pattern, and that it will generate a configuration that outperforms the approach without workload analysis. Moreover, the improvement should hold for different noise ratios up to the point in which no significant patterns can be extracted from the workload.

The Cumulus approach of considering only those objects accessed by the significant patterns during the generation of the configuration should avoid cases in which a certain site becomes a bottleneck (see Section 5.2). The performance advantage of Cumulus should increase as the load increases.

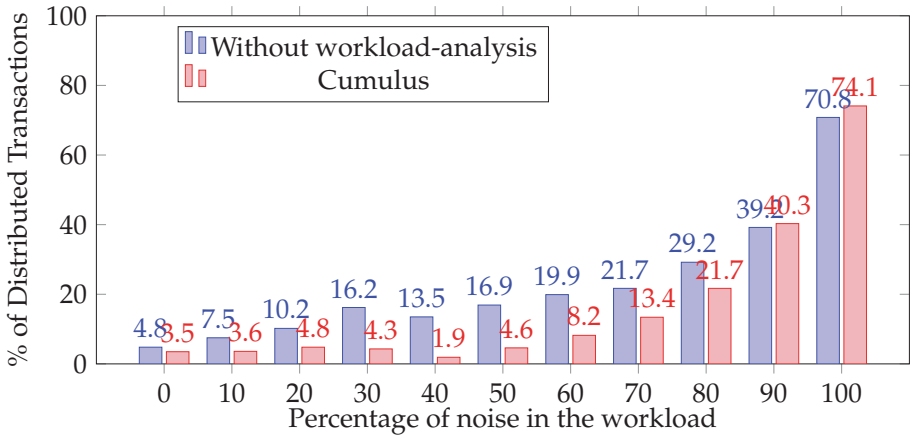


Figure 7.16: Percentage of distributed transactions.

Experimental Results

Figures 7.16 and 7.17 depict the results of the experiment. As shown there, the workload analysis of Cumulus outperforms in terms of quality the approach that does not analyze the workload. This improvement in the quality is reflected in the considerable lower percentage of distributed transactions in case of Cumulus. It is clear that if the noise level is very high (e.g., 100%), then, as no characteristic access patterns exist, workload analysis cannot lead to any improvement.

Figures 7.18a and 7.18b depict the performance results of three different partition approaches, namely the Cumulus approach that considers only objects that occur in the significant workload to determine the partitions (SUBSET), an approach in which all objects are used (ALL), and a hash-based approach as a control mechanism. As shown in the results, the Cumulus approach yields the best performance compared to the other two approaches, mainly due to the better load balancing. The ALL approach yields a 0% distributed transaction rate, which is expected as it better collocates objects accessed together. This rate was between 4% and 10% for Cumulus and over 90% for the hash-based approach. Hence, the better performance of Cumulus (SUBSET) comes from the better load balancing capability compared to the ALL approach, especially from the avoidance of hot spots. This shows that an important criterion for the overall system performance is to find a good trade-off between load balancing and minimization of distributed transactions; concentrating only on one aspect does not automatically lead to better system performance.

7.5.3 Adaptive Partitioning

Cumulus is an adaptive partitioning protocol, able to react dynamically to access pattern changes at runtime, and adjust its configuration to reflect these changes. As described in Section 6.3.2, Cumulus implements an on-the-fly and on-demand (OFD-R) reconfiguration approach, that, in contrast to the stop-and-copy approach (SC-R), incurs lower system unavailability. The goal of this experiment is to compare the runtime

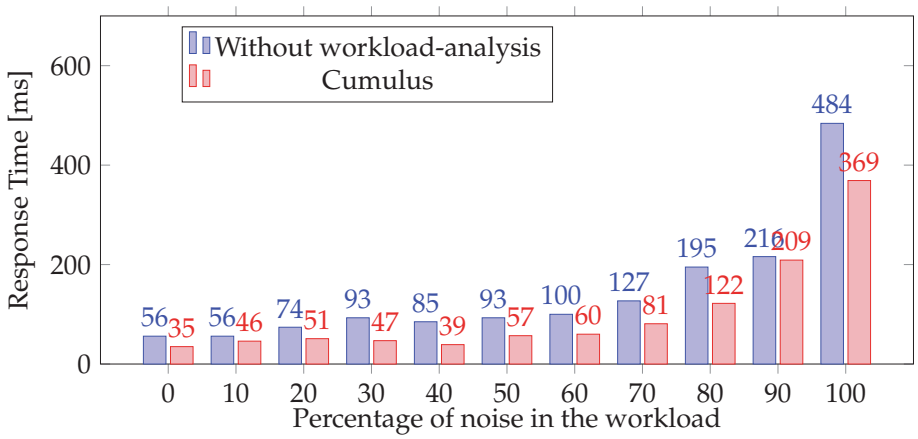
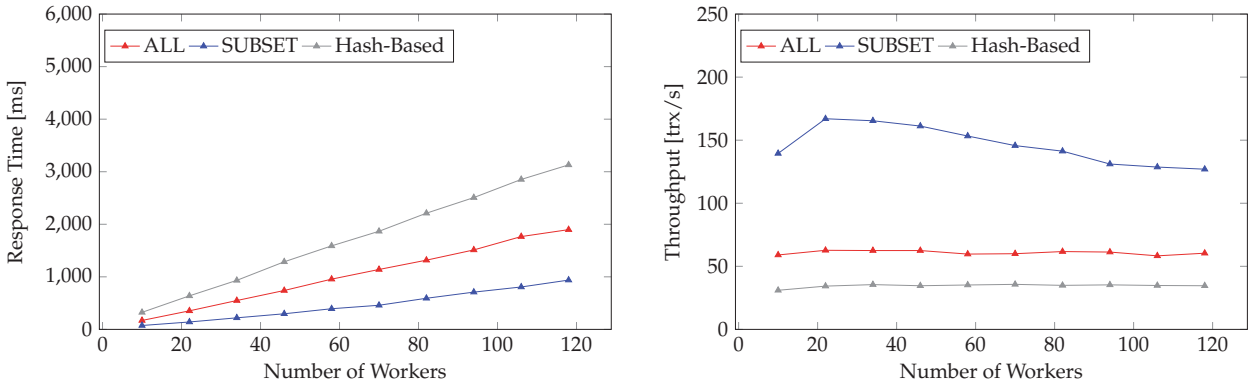


Figure 7.17: Impact of workload analysis on the percentage of distributed transactions.



(a) Response time of transactions

(b) Transaction throughput

Figure 7.18: Comparison of the different partitioning approaches.

behavior of both approaches. Moreover, the experiment shows the ability of Cumulus to detect changes in the workload and consequently adapt the configuration.

We have run two types of experiments with the goal of assessing the runtime adaptive behavior of Cumulus.

1. The first experiment compares the OFD-R approach to the SC-R reconfiguration. We have implemented the SC-R approach into Cumulus and have added the possibility to define at runtime which of the reconfiguration approaches should be used by Cumulus. This experiment is run twice, once for each of the two modes using the same single pool of access patterns. There is no warm-up phase as the goal is to show the adaptive behavior. Cumulus runs for a certain period of time, extracts the access patterns and then triggers a reconfiguration in the specified mode.

2. The second experiment evaluates the ability of Cumulus to adapt the configuration in reaction to changes in the workload. For this experiment, we have used different pools of access patterns and switch at runtime between them in order to simulate workload changes.

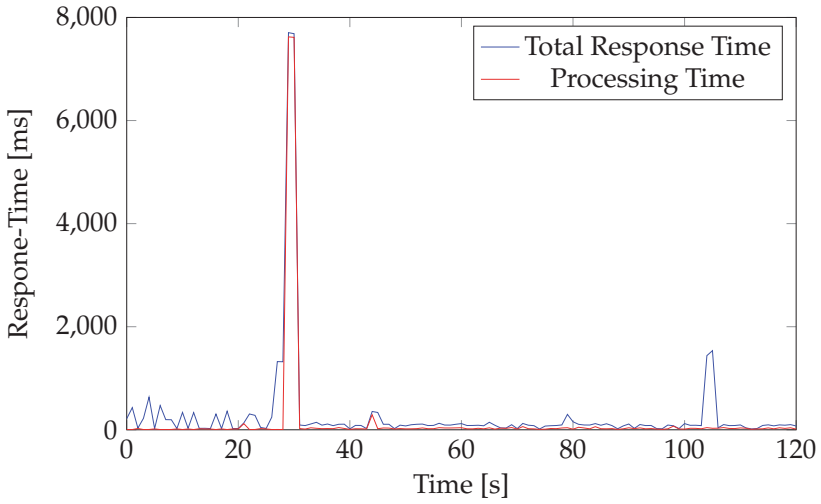


Figure 7.19: Stop-and-copy reconfiguration.

Both experiments use the following setup parameters: $\#sites = 4$, $\#workers = 50$, and $r/w = 0.5/0.5$.

Expected Results

We expect the OFD-R approach to lead to a lower system unavailability during the reconfiguration, which, in contrast to the SC-R approach, migrates objects only when accessed by user transactions. With regards to the adaptive behavior, we expect Cumulus to detect workload changes and timely adapt its configuration to the new workload.

Experimental Results

Runtime behavior of the Reconfiguration Figure 7.19 depicts the response time behavior of transactions using the SC-R approach, and Figure 7.20 the response time using the OFD-R approach. As it can be seen, the SC-R approach generates a huge spike in the response time, which is then stabilized once the reconfiguration is completed. During the reconfiguration, all incoming transactions are added to a wait-queue, and their execution is deferred. Depending on the number of the objects to be migrated, the stop and copy approach may lead to a considerable system unavailability that may destabilize the system, especially during high arrival rates as the wait-queue may be filled in completely.

In the case of the OFD-R approach, the immediate impact of reconfiguration is lower. However, there is an overhead incurring for individual transactions, as the effort for the reconfiguration of the objects they access will be added to these transactions. This overhead is visible in the higher response time compared to the SC-R approach after the reconfiguration. The overhead for individual transactions will decrease as after some time the reconfiguration will be completed.

Adaptive Behavior of Cumulus Figure 7.21 the behavior of Cumulus during the workload shifts and shows its ability to detect them and timely react by adapting its configu-

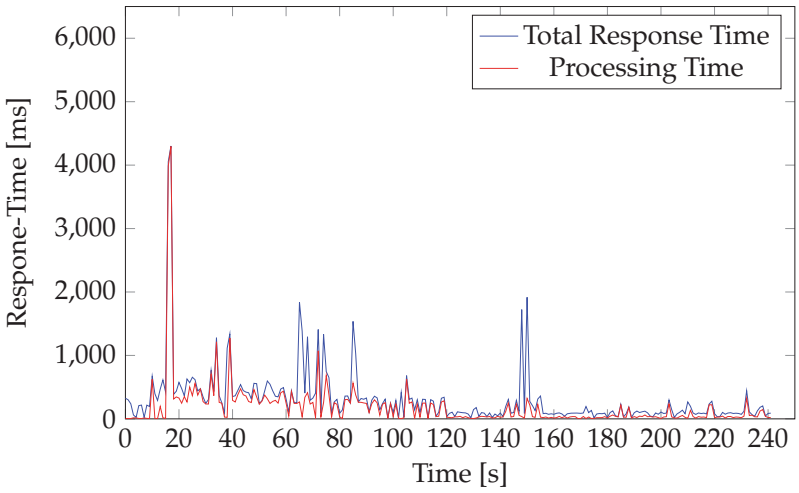


Figure 7.20: On-the-fly and on-demand reconfiguration.

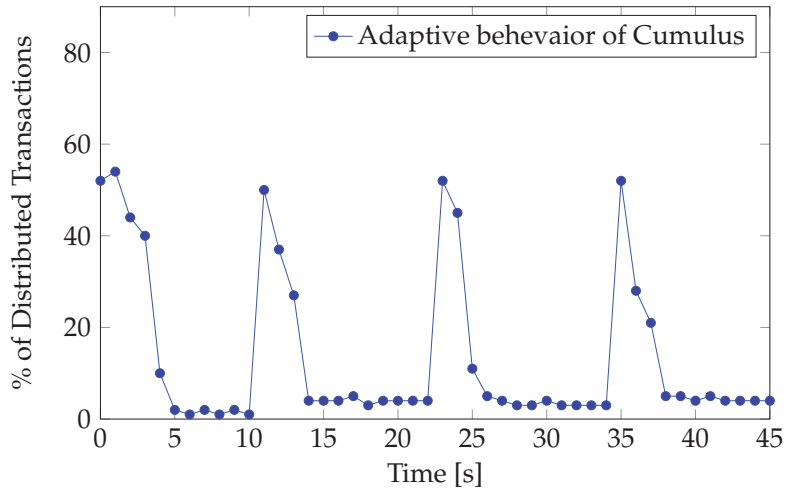


Figure 7.21: Percentage of distributed transactions over time with shifting access patterns.

ration. The four workload shifts are clearly visible, during which, there is a considerable increase in the percentage of distributed transactions. The percentage of distributed transactions decreases once Cumulus has adapted its configuration. This is sharp contrast to static partition protocols, in which the spike of distributed transactions would remain. Hence, in the presence of dynamically changing access patterns, Cumulus outperforms static approaches that can only be optimized for an initial access pattern but become suboptimal as the workload evolves.

7.5.4 Summary

In summary, the conducted experiments show that distributed transactions are expensive (due to commit coordination) and thus should be avoided. Data partitioning, if done properly, can considerably decrease the percentage of distributed transactions and

can thus impact the overall system performance. This is shown in a comparison of Cumulus with a fully replicated system configuration (ROWAA) where distributed update transactions cannot be avoided at all.

Workload analysis is of high importance for the quality of the generated partitions. However, distributed transactions are not the only source of overhead. It is crucial to avoid hot spots in the system as they decrease overall system performance. Cumulus, therefore, combines the avoidance of hot spots with the attempt to minimize distributed transactions. As access patterns of applications deployed in the Cloud are dynamic and may change over time, a partitioning protocol should be adaptive and, at the same time, should incur minimal reconfiguration overhead. Frequent reconfiguration events that do not provide sufficient gain should not take place as the cost of adaptiveness may overweight its advantage. Cumulus assess the gain of a reconfiguration before actually initiating it.

7.6 QuAD Evaluation Results

The goals of the QuAD evaluations are as follows. First, we show the importance of considering site properties when constructing the quorums. For that we have used a version of the MQ protocol, that neglects site properties, and depict the impact of that strategy on the overall performance. Second, we compare the performance of QuAD to that of MQ using round-robin and random quorum construction strategies, in a single-data and a multi-data center settings. Third, we compare the different construction strategies of QuAD and show their impact on the overall performance. And fourth, we analyze the necessity of adapting the quorums if site properties change at runtime and show that QuAD can adapt its quorums.

We have run two different types of experiments, depending on the concrete purpose, namely a *sizeup* and *speedup* test as described in [CST+10]. In the *sizeup* experiment, an initial number of 10 worker threads will be started and submit transactions for 30 seconds, and after that collect the statistics and increase the number of workers by 10, until the maximum of 150 workers is reached. Each *sizeup* experiment runs for 450 seconds and is repeated 10 times.

The goal of the speedup test is to analyze the performance improvement of QuAD compared to other approaches. The speedup is calculated as follows: $speedup = \frac{resptime(other-approach)}{resptime(QuAD)}$, and there is an improvement if $speedup > 1$. During a speedup experiment, the load remains constant, i.e., *#workers* does not change.

7.6.1 Impact of Site Properties on Performance

The goal of the quorum protocols is to reduce the overhead for update transactions as only a subset of sites is eagerly committed. This reduces the number of synchronization messages in the system and the load generated at the sites. However, to guarantee strong consistency, reads must also access a subset of sites, and this –in contrast to the ROWAA approach– increases the overhead for the reads. In Cloud environments, it is essential that quorums consider the properties of the available sites. In order to show the

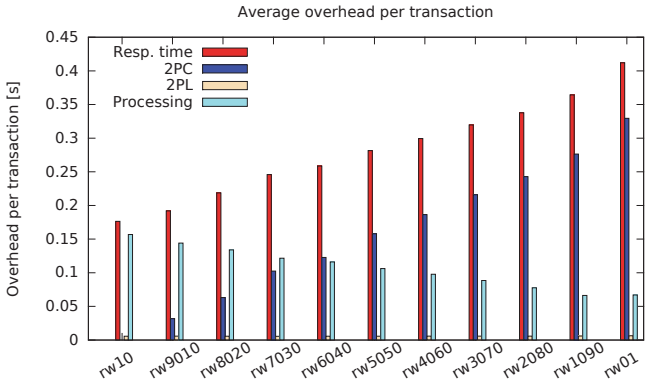
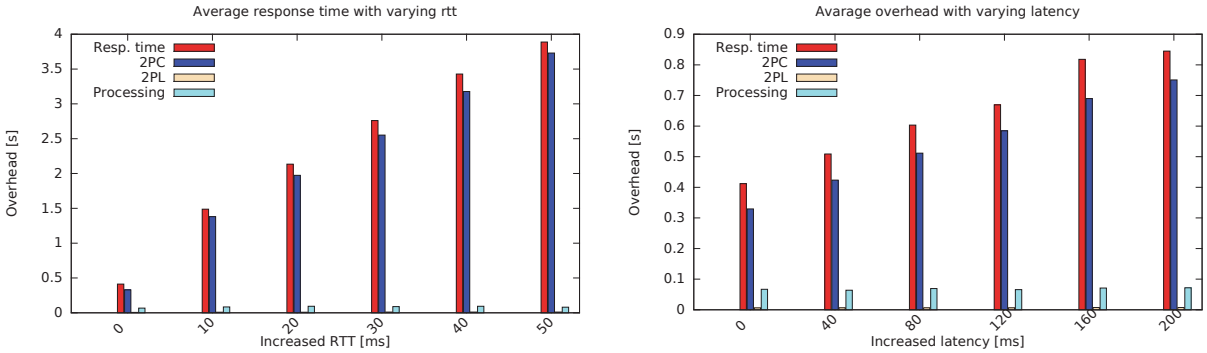


Figure 7.22: MQ: Transaction overhead with varying r/w ratio.



(a) Varying RTT

(b) Varying load

Figure 7.23: Transaction overhead in MQ for a workload consisting of update transactions only.

necessity of considering site properties, such as their RTT and load, when constructing the quorums, we have conducted two experiments on the basis of the MQ protocol with $\#sites = 4$.

In the first experiment, a test client with a single worker thread generates transactions for 450 seconds based on a specific transaction mix. These transactions are then submitted for execution to a dedicated site, which initially constructs a majority quorum by randomly picking from available sites.

In the second experiment, we have varied the RTT[4] and the load of a certain site. We ensured that the modified site is included in the read and write quorums.

Expected Results

When all sites have the same properties, we expect the transaction mix to be the determining factor for the performance. The more update transactions the workload contains the higher the probability of conflicts due to 2PL, which is known to be the limiting

[4]We used the `netem` tool to increase the RTT at a certain site: `www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

factor for the transaction throughput [BN96]. Moreover, update transactions incur an overhead for the 2PC, which additionally increases the lifetime of transactions and by that also the conflict probability (2PL overhead).

In a deployment in which sites differ in terms of their properties, we expect that the weakest site become the bottleneck and degrades the entire performance.

Experimental Results

Figure 7.22 shows the overhead per transaction when all sites have the same properties (i.e., the load of the sites and the RTT between sites is the same) for a varying r/w ratio. As shown there, with the increase of the update ratio in the workload, the 2PC costs also increase. The overhead generated by S2PL remains constant as there are no concurrent transactions. The goal was mainly to depict 2PC overhead. In summary, in the case of all sites having the same properties, the workload mixes determines the transaction overhead.

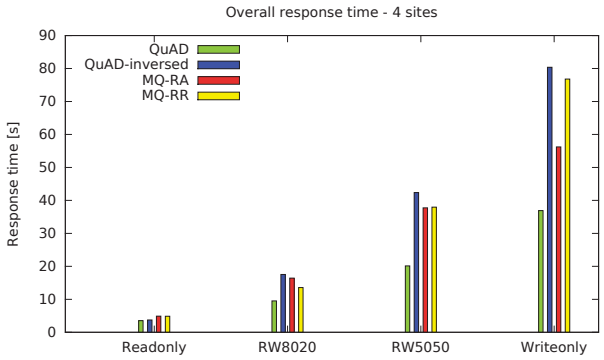
In Figure 7.23 we have depicted the results for a workload consisting of update transactions only. The increase of the RTT or load at one of the sites that is included in all read and write quorums has a considerable impact on the 2PC overhead. This is a consequence of the site being part of each commit path, and thereby slowing down the 2PC processing (in Figure 7.23a and Figure 7.23b, latency corresponds to the additional overhead generated compared to the latency when $rtt = 0$ and $latency = 0$). Thus, it is crucial to constructing the quorums in such a way so that the weak sites are avoided from the read and commit paths of transactions.

7.6.2 QuAD vs. MQ

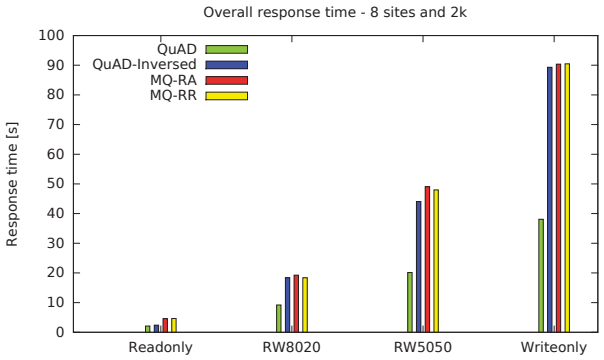
In the next series of experiments, we compare the performance of QuAD to that of the MQ that uses round-robin (MQ-RR) and random (MQ-RA) approaches for quorum construction. For that, we have run a *sizeup* test in a single-data center and a multi-data center setting. The worker threads ($\#workers = [10 : 10 : 150]$) will submit transactions to sites according to the desired r/w ratio. We run the experiments using four, eight and sixteen sites, with a subset of sites denoting the core sites.

In the single-data center environment, the load is the determining factor for the performance, as the network distance between the sites is negligible. The distribution of worker threads to sites determines the load generated at the sites, and by that their score. For the evaluation with four sites, one of the sites will get 40% of the overall load, the second one 30%, the third one 20% and the last one the remaining 10%. In the evaluation with eight sites, distribution is as follows: 30%, 15%, 15%, 10%, 10% and the remaining 20% are evenly distributed to the rest of the sites. The distribution of load in the case with 16 sites is similar to that with eight sites. The core sites are determined based on the model defined in Section 5.3.

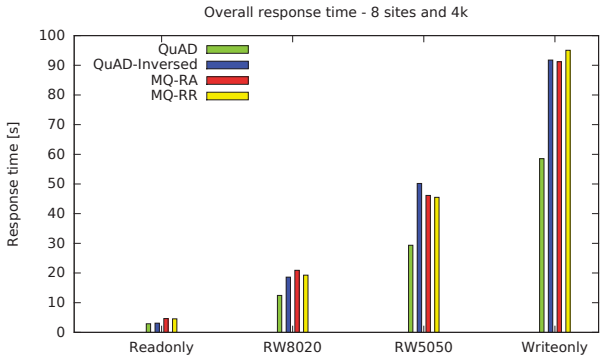
The goal of the *multi-data center* deployment is to evaluate the impact of RTT on the overall performance. Therefore, we have increased the RTT between three sites (with ratio of 4:2:1), so that the communication between them corresponds to the communication between sites located at three different data centers. As the load of all sites is the



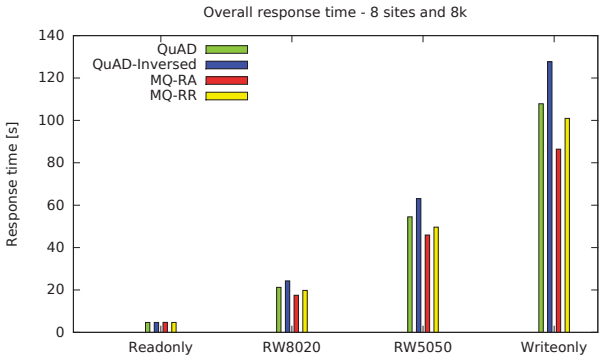
(a) 4 sites; $\kappa = 2$



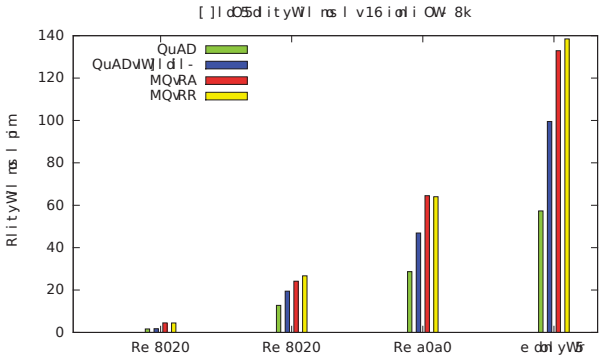
(b) 8 sites; $\kappa = 2$



(c) 8 sites; $\kappa = 4$



(d) 8 sites; $\kappa = 8$

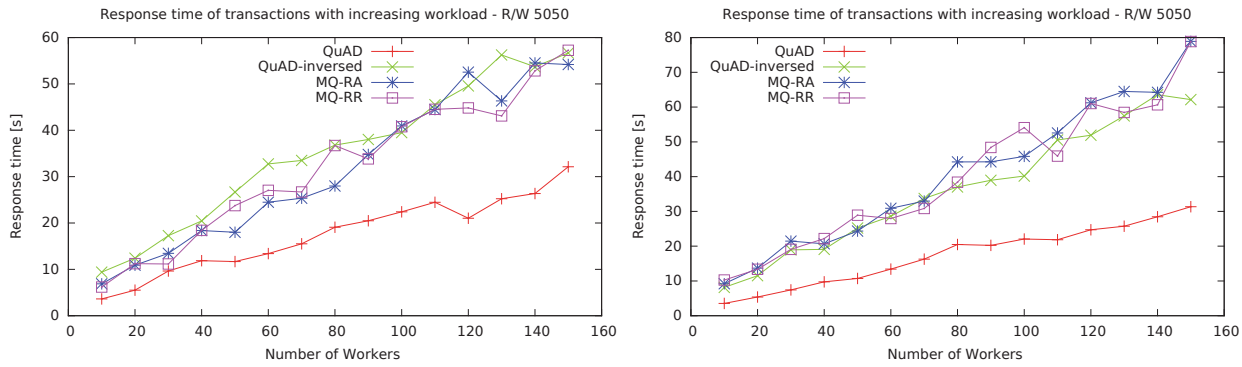


(e) 16 sites; $\kappa = 8$

Figure 7.24: Overall response time of transactions with varying site load (single-data center setting).

same (which is achieved by distributing transactions to sites in a round-robin approach), the RTT will be the determining factor for the construction of quorums.

We will also report the result for the QuAD-inversed, in which the weaker the site, the higher its score, simply for reasons of comparison.



(a) 4 sites; $\kappa = 2$ (b) 8 sites; $\kappa = 4$

Figure 7.25: Sizeup: QuAD vs. other quorum protocols for $r/w = 50\%/50\%$ (single-data center setting).

Expected Results

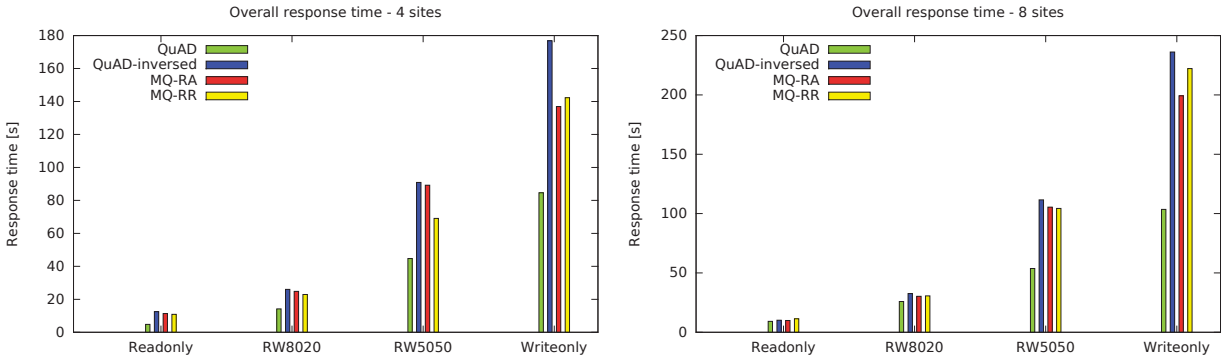
In a DDBS with varying site properties, we expect QuAD to exploit the advantages of its optimization model that considers the site properties when constructing the quorums. By avoiding weak sites from the read and commit paths of transactions, QuAD should considerably outperform the MQ protocol in terms of performance. This expectation applies to the single-data and multi-data center environments.

Experimental Results

Single-Data Center Figure 7.24 depicts the overall averaged response time of transactions in the *sizeup* test, and Figure 7.25 shows the response time behavior of transactions with increasing *#workers*. Based on the depicted results we can conclude that QuAD considerably outperforms both MQ approaches, namely the MQ-RR and MQ-RA, that neglect site properties when constructing the quorums. For update-heavy workloads, QuAD leads to a decrease of response time by more than 50%. The main reason is that quorums are constructed in such a way, so that weak sites are possibly avoided. The properties of the site that received the transactions should be the limiting factor and not the properties of the sites consisting the quorums.

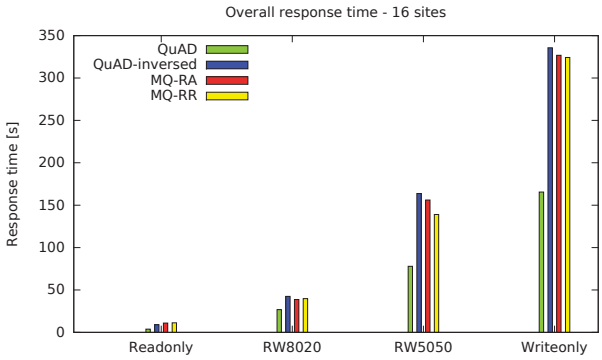
However, it is clear that QuAD, similar to other quorum protocols, have a higher overhead for reads compared to ROWAA. We run a simple speedup test for comparing the performance of QuAD to that of ROWAA for a read-only workload with all sites having same properties. The average response of ROWAA was about 0.08 seconds, and that of QuAD about 0.2 seconds, which means that using ROWAA for read-only workloads leads to a speedup of 2.5.

Multi-Data Center The experimental results for the multi-data center setup are depicted in Figure 7.26. From the results, we can conclude that QuAD significantly outperforms other approaches especially for update-heavy workloads by decreasing the average response time nearly by a factor of 3. The consideration of RTT is more significant especially for update-heavy workloads as they are mainly network bound due to



(a) 4 sites; $\kappa = 2$

(b) 8 sites; $\kappa = 4$



(c) 16 sites; $\kappa = 8$

Figure 7.26: Varying RTT (multi-data center setting).

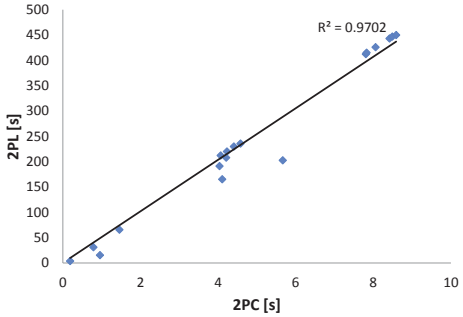


Figure 7.27: Correlation between the 2PC and S2PL overhead [SFS15].

the heavy 2PC communication. Note that as we use SOAP over HTTP for the communication between sites, the RTT is a crucial factor to the overall performance.

In [SFS15], which compares the performance of different replica protocols, we reported that an increase of the RTT by a factor of $20 - 40$ doubles the 2PC costs. This has a considerable impact on the overall performance as there is a strong correlation between the 2PC and S2PL costs, especially for update-heavy workloads (Figure 7.27).

7.6.3 QuAD Quorum Construction Strategies

The goal of the following experiments is to compare the QuAD quorum construction strategy to strategies that neglect one or more parameters from Equation 5.30. For that purpose we have run the following experiments:

1. We have compared the QuAD strategy of assigning slaves to core quorums which jointly considers both the load and the RTT to strategies that either only consider the load or the RTT[5]. For this evaluation, $\#sites = 4$ with $\kappa = 3$. This means that a single slave site needs to choose between three possible core quorums. In this evaluation the load remains the same, i.e., $\#workers$ remains constant. All transactions are submitted to the slave site.

2. When assigning the slaves to core quorums, QuAD tries to balance the assignment as much as possible to avoid that certain core quorums, and with that certain core sites, become a bottleneck if to many slaves are assigned to them. We have also compared the balanced assignment of slaves to cores with the unbalanced assignment based on a sizeup test.

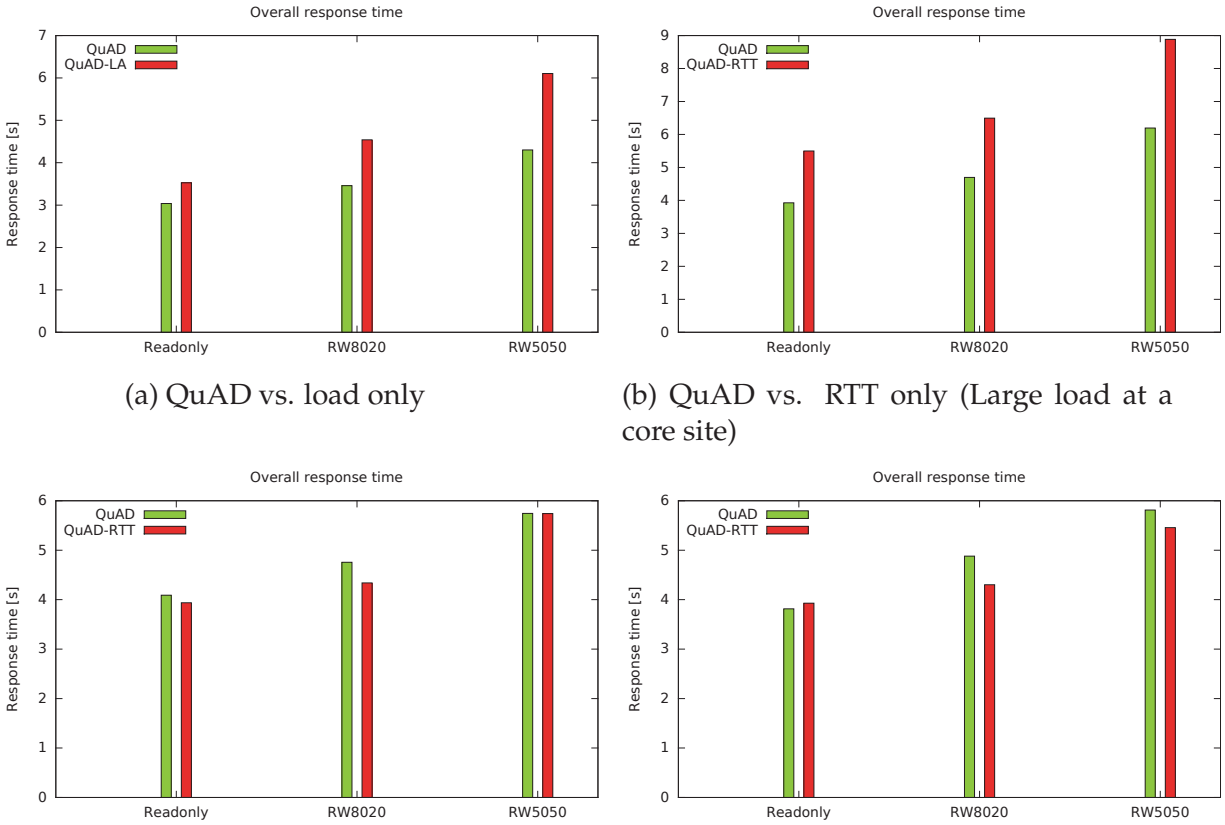
3. One of the crucial aspects in QuAD is the choice of κ as it impacts both its performance and availability. We have also evaluated the impact of κ to the overall performance based on a *sizeup* experiment with $\#sites = 8$ and a varying κ. The goal of this experiment was to show the trade-off between the availability, which increases with increasing number of κ sites, and the optimization capabilities, which may rapidly decrease if the properties of the sites are similar. Note that the decrease strongly depends on the site properties. If a subset of sites is significantly better than the rest, then there might even be an advantage in increasing the number of core sites (selected from this subset), as the more core sites available the more choices there are for assigning the slaves which, in turn, may be advantageous from a load balancing point of view. However, if there are many weak sites, the higher the number of cores the more weak sites are to be included. In this experiment, two sites were quite strong sites, and the rest was weak.

Expected Results

The QuAD strategy that jointly considers load and RTT when assigning slaves to cores should outperform strategies that neglect one of the two parameters. However, certain workload types may, for example, be network bound, and some others latency bound. In these cases it is more advantageous to optimize the load and RTT weights to consider these properties. As the QuAD' default assignment strategy gives equal weights, in the case of the aforementioned workloads, we expect it to perform worse compared to those strategies that consider only the load or the RTT.

Avoiding bottlenecks, i.e., sites that are more frequently accessed in the quorums than others, is crucial for the overall performance. We expect the balanced assignment

[5]This is achieved by specifying a zero weight for the load or RTT in Equation 5.30.



(a) QuAD vs. load only

(b) QuAD vs. RTT only (Large load at a core site)

(c) QuAD vs. RTT only (Medium load at a core site)

(d) QuAD vs. RTT only (Low load at a core site)

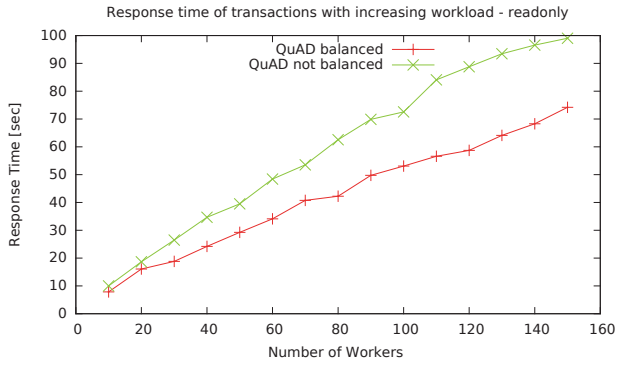
Figure 7.28: Comparison of strategies for the assignment of slaves to core sites.

of QuAD to provide an overall better performance compared to the non-balanced strategy. Furthermore, this improvement should become even more tangible as the load increases.

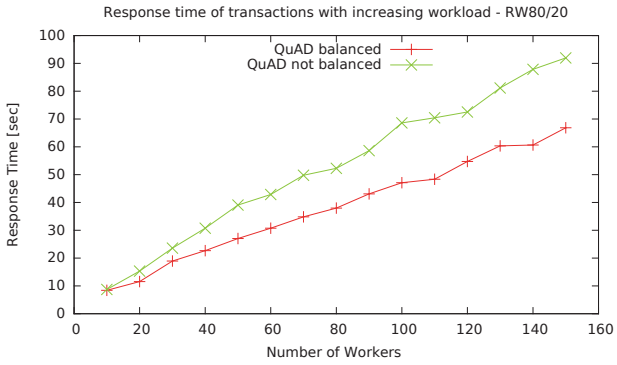
With regards to the impact of κ to the performance, if the DDBS consists of mostly weak sites, a large value of κ means that many of these weak sites need to be included in the quorums. In that case, we expect a high value of κ to be a disadvantage in terms of performance.

Experimental Results

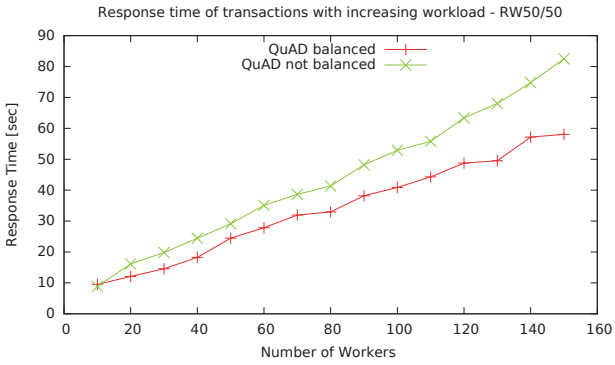
QuAD vs. Load only In the first experiment, we compare QuAD to an assignment which considers only the latency (QuAD-LA) when assigning the slave site to the cores. The latency at a site is determined by its load. The core site with the smallest load has the greater distance to the slave, and as the load of the cores increases, the distance decreases by the same factor. QuAD-LA chooses the core quorum that has the minimum maximum load, and assigns the slave to that quorum, whereas QuAD chooses that quorum that has the lowest cost by jointly considering the load and RTT. As depicted in Figure 7.28a, QuAD outperforms QuAD-LO for all r/w ratios. The performance advan-



(a) Readonly



(b) RW8020



(c) RW5050

Figure 7.29: Balanced vs. unbalanced assignment of slaves to core sites for varying r/w ratio.

tage of QuAD is considerable for update-heavy workloads as they are mostly network bound.

QuAD vs. RTT only In the second experiment, we compare QuAD to an assignment which only considers RTT (QuAD-RTT). The core site with the smallest RTT has the highest load, and as the RTT of the cores to the slave increases, their load decreases. We conduct three different experiments which differ in the load generated at the weakest core site from the load point of view. The first evaluation generates a load that corresponds to an average latency of $2,500[ms]$, the second to $500[ms]$ and the third one to an average latency of $100[ms]$. For the high load evaluation, QuAD outperforms the QuAD-RTT (Figure 7.28b). However, as the load of the weakest core site becomes smaller, the performance of QuAD-RTT gets better and it outperforms the performance of QuAD (Figures 7.28c and 7.28d). The reason is that in our SOAP/HTTP-based implementation, the network should have a higher weight compared to the load. Currently, in QuAD the weights can be configured by a user. However, it is planned to relieve the user from the burden of determining the appropriate weights by incorporating a machine learning approach to automatically determine the optimal weights.

Balanced vs. Unbalanced Assignment of Slaves to Cores The results of the comparison of the QuAD balanced strategy of assigning slaves to quorums to a non-balanced

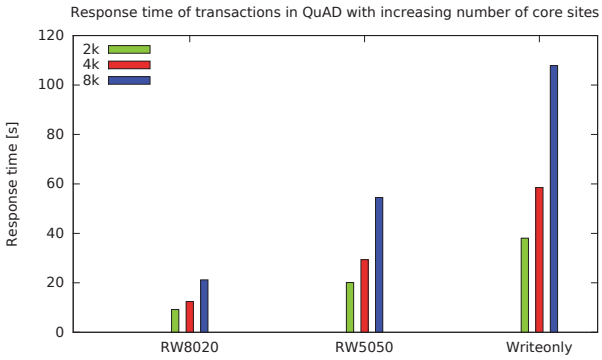


Figure 7.30: Varying κ (8 sites).

version are depicted in Figure 7.29. As the results show, as the load increases, the unbalanced version in which certain core quorums have more slaves assigned to them, will not only become bottlenecks from the point of view of the slave sites, but also transactions executed by them will be impacted by the high load generated from the slaves, so that the entire performance will degrade. The balanced version of QuAD leads to a decrease in response time between $20\% - 30\%$ compared to the non-balanced version.

Impact of κ to Performance The results of this evaluation are depicted in Figure 7.30. As the results show, the increase in the number of cores leads to a decrease of performance. κ determine the size of the quorums, and the smaller its value the higher the probability that the weak core sites are bypassed during the construction of quorums. If all sites are core sites, then the behavior of QuAD is similar to that of ROWAA for update transactions. It is well known that ROWAA has no choices whatsoever to optimize the commit of update transactions as all available sites are part of the commit path.

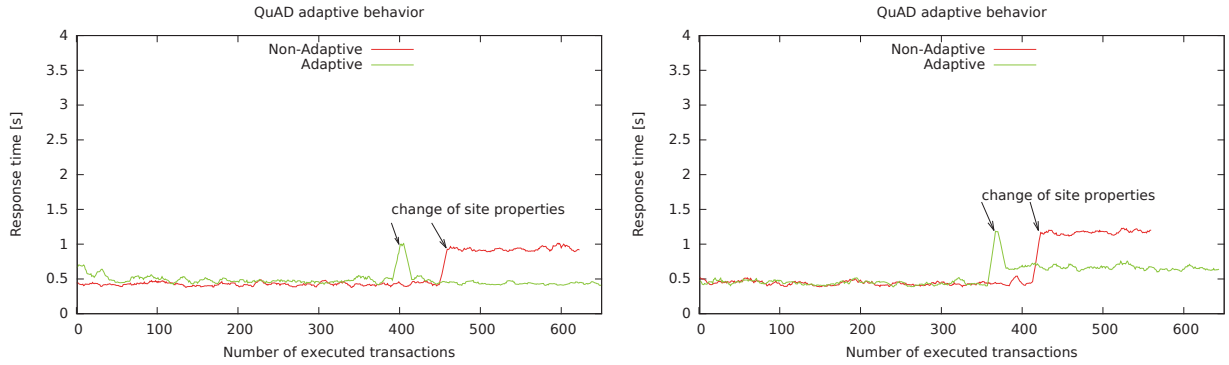
In addition to the availability and performance, the choice of κ also influences the monetary cost of applications. The size of quorums strongly correlates to the κ value: the higher the κ value the bigger the quorum size and thus the number of messages that have to be exchanged between the sites. As each message incurs a monetary overhead in the Cloud, the more messages are exchanged the higher the overall application cost.

7.6.4 Quorum Reconfiguration

In the following two experiments, we evaluate the ability of QuAD to adapt the quorums in reaction to changes of site properties. In the first experiment, the load of a core and a slave site are swapped, i.e., the core site gets the load of the slave site and vice versa, whereas in the second experiment, we increase the load at one of the core sites to a level that is between the load of the two slave sites.

Expected Results

We expect QuAD to adapt its quorums by promoting slaves to cores, and demoting cores to slaves. This adaption should be clearly visible in the response time of transactions.



(a) Swapping load of a core and a slave site

(b) Increasing load at one core site

Figure 7.31: QuAD adaptive behavior.

Experimental Results

The experimental results of the first experiment are depicted in Figure 7.31a and the results of the second experiment in Figure 7.31b[6].

For the first experiment, the results show clearly that QuAD reacts to the load swap and adapts the quorums. Compared to the non-adaptive quorum, this leads to a stabilization of the latency to the level before the change in the load. The same applies for the second experiment, with the difference that the latency will remain at a higher level after the adaption of quorums, as the load of the slave site that is promoted to a core does not decrease.

7.6.5 Summary

In summary, the conducted experiment show that the size of the quorums is not the only factor that determines the overhead in quorum-based replication. The properties of the sites are at least equally, if not more important, and should become first class citizens during the quorum construction. QuAD incorporates an advanced quorum strategy based on the κ-centers model, which, as shown by the experimental results, considerably outperforms (up to 50% gain in performance) approaches that neglect site properties. Moreover, QuAD considers also a balanced quorum construction by avoiding bottlenecks as far as possible. A strong site that is included in too many quorums will degrade the overall performance. Furthermore, QuAD is able to adapt the quorums at runtime if there is a significant change in the workload.

7.7 Discussion

The thorough experiments we have conducted prove that it is not only feasible but also necessary to build data management protocols that are able to adapt their behavior at runtime. Our CCQ protocols, by considering application workload and deployment

[6]Note that we initiate the adaption at different point is time to avoid an overlap in the charts.

characteristics, can lead to considerable savings in monetary cost and increase in performance.

Nevertheless, the adaptive models can benefit from extended evaluations in a real application deployment; that would run for a longer period of time. That would allow us to gain insights on more realistic access patterns, which can be useful to optimize further the reconfiguration approaches that we have implemented. As the online reconfiguration is one of the crucial components that determines the price of adaptiveness, it pays off to further optimize it, by, for example, considering the overall load of the system. If the load is too high, then the reconfiguration may be postponed, and resumed once the load decreases below a specific threshold.

The generated overhead is closely related to the reconfiguration period. The longer the period, the more activities need to be executed during the reconfiguration. Although the system may benefit by postponing a reconfiguration, that might lead to an overhead that overweights its gain. Thus, the optimal point in time for the reconfiguration is dependent on different parameters that need to be further evaluated.

Workload prediction is at the core of the CCQ protocols. Currently, we used time series prediction mainly due to their simplicity and low processing overhead. Additionally, they provided predictions of sufficient accuracy for our purpose. There are more advanced prediction models that can better tackle with workload skew and highly dynamic workloads, which is a typical characteristic of the applications deployed in the Cloud. However, they also generate more overhead, which may become an obstacle for adaptive protocols. The deployment in a real application together with a thorough evaluation of different prediction models would make it possible to identify those models that find the right balance between accuracy and acceptable overhead.

The CCQ protocols need to collect, manage and process system meta data to take a decision on a suitable configuration. All these activities generate additional overhead that is not explicitly quantified by current evaluations. We argue that the CCQ protocols can benefit from specific experiments that help to precisely asses the additional overhead and provide insightful information on eliminating the sources of that overhead.

8

Related Work

In this chapter, we summarize related work in the field of distributed data management. The chapter is structured in five sections. In the first section, we discuss general approaches and directions in the distributed management of data. The second section discusses related work in the context of modular DBSs. In the third section, we summarize related work in the field of data consistency for distributed databases together with a detailed description of approaches targeting cost-based consistency. The fourth section discusses related work for data partitioning with a focus on adaptive protocols. The fifth and final section discusses related data replication approaches.

8.1 Distributed Data Management

The development of distributed data management research has been mainly driven by the fundamental trade-offs captured by CAP/PACELC. The rise of companies, such as Google, Yahoo!, Facebook, and others, that started providing applications over the Internet, and the ever growing popularity of these applications, have led to a dramatical shift in the requirements towards database systems. These applications are characterized by a huge number of users and their high availability (possibly 24/7) and scalability demands.

The Cloud paradigm, with its pay-as-you-go cost model and virtually infinite scalability, has reduced the barrier for the development and deployment of large-scale applications, that were difficult or even impossible in traditional infrastructures due to the necessary upfront investment. These applications are characterized by a highly dynamic workload, which requires from the DBSs the ability to adapt to the varying load.

Relational Database Management System (RDBMS) have long been the database of choice in a traditional enterprise setting. While they can scale up, RDBMS are not suitable when it comes to horizontal scalability (scale out) on commodity hardware [ADE12]. Scale up is not an appropriate choice for applications deployed in the Cloud. While the dynamic nature of these applications demands adaption with possibly no delay, scale up requires hardware modifications, which considerably increases the reaction time.

Many of the aforementioned companies developed own DBSs that provide high availability, and are able to scale out on a commodity hardware. Examples include Bigtable [CDG+06], Pnuts [CRS+08], [LM10], and others.

The huge success of this first generation of NoSQL databases, which were mainly tailored to satisfy the needs of their creators, gave a rise to the development of a plethora of open source NoSQL databases, such as key-value databases (Redis [Car13], Voldemort [SKG+12], Riak [Klo10]), document databases (SimpleDB [Simb], CouchDB [ALS10], MongoDB [CD10]), and graph databases (Neo4J [Web12], OrientDB [Ori]).

Peer-to-Peer (P2P) data management systems have become quite popular mainly in the context of file sharing applications. They differ from traditional DDBSs in the degree of heterogeneity and autonomy of sites, as well as in the volatility of the system [ÖV11]. In a P2P DBS sites can join and leave the system at any time. Early P2P systems, such as Gnutella [Gnu], Kazaa [Kaz], Napster [Nap], CAN [RFH+01] and Chord [SMK+01], mainly focused on avoiding censorship and fast query processing.

Recent P2P systems also focus on providing rich query language similar to SQL for relational databases. The main difficulty in providing high-level query language is the unavailability of a centralized schema. Some approaches consider schema mapping on the fly during query processing [OST03], some others rely on the existence of super peers that provide schema description [NSS03]. Transaction management in P2P systems with replication remains an open issue [VP04].

8.2 Policy-based and Modular Data Management

Traditional (relational) DBSs were developed with the goal of supporting different application types that have diverging requirements. However, this approach, known as "one size fits all", has failed to satisfy the demands of emerging application types, such as large scale web applications, social networks, and others. In [SÇ05, Sto08] the authors provide a thorough analysis of why the "one size fits all' approach is no longer applicable to the database market. They conclude that new types of specialized DBSs will evolve, a trend observable in the NoSQL movement, tailored to the requirements of a specific type of application. This conclusion is not new and has been long ago observed in the mismatch between the requirements of Online Analytical Processing (OLAP) and OLTP applications towards the DBSs. As described in [Fre95] traditional DBMS architectures do not work well for data warehouse applications (OLAP) as their optimizations target OLTP workloads.

The PolarDBMS' approach fits very well in the context of this movement that goes away from the "one size fits all" approach. However, from the software engineering point of view, PolarDBMS follows the approach of having a thin DBMS framework, responsible for the management of modules implementing a certain data management protocol, instead of having a zoo of different specialized DBSs. Additionally, PolarDBMS treats the client as first class citizens by giving it the possibility to directly influence the DBS composition through a fully transparent interaction cycle as described in Section 2.2.

If we consider, for example, NoSQL databases, such as SimpleDB [Simb] or Oracle NoSQL [orab], which initially provided only weak consistency to applications, they have recently included strong consistency and allow applications choosing the appropriate consistency model. PolarDBMS, in contrast,s will autonomously decide and continuously adapt the provided guarantees, and the protocols implementing those guarantees by considering the full set of application requirements and workload. Each different PolarDBMS composition corresponds to what [SÇ05, Sto08] calls a "specialized engine".

OctopusDB [DJ11] is a DBS that allows to choose a suitable data model, depending on the application characteristics. The data model is however only a subset of data management properties considered by PolarDBMS. [KKL+10] introduces Cloudy, which, similar to PolarDBMS, is based on a modular architecture that enables its customization based on a concrete application scenario. In contrast to PolarDBMS it largely ignores the definition of requirements based on SLAs. [GSS+13] proposes an architecture based on co-design of the DBMS and operating systems, which allows the design of both by considering each other's requirements.

In [GFW+14] and [GBR14] the authors propose a unified Representational State Transfer (REST) interface for the access to NoSQL databases offered through the web (e.g., S3, Riak, etc.). The interface abstracts the different properties of these NoSQL databases, such as data consistency guarantees, data models, query languages, etc. The unified API is a perfect fit for PolarDBMS and would largely simplify the challenges related to the support of a wide range of different databases by PolarDBMS. Moreover, the authors propose a modular middleware called *Orestres* that provides an implementation of core database concepts, such as schema management or authentication, as modules. As these modules are accessible via a REST API, they can be easily incorporated into PolarDBMS, which, by that, is further decoupled from database specific functionality. Furthermore, this is in line with the goal of PolarDBMS to not only exchange database functionality at runtime, but even entire databases.

8.3 Workload Prediction

Load prediction is at the core of adaptive data management protocols, as it allows them to anticipate future workload and to adapt their behavior accordingly. For example, based on the predicted workload it is possible to determine the number of necessary sites so that certain SLOs are not violated.

Different prediction models exist that differ in terms of accuracy and cost. As these two goals are in conflict, it is crucial to find the right balance between them. The cost is especially a critical factor for adaptive systems, as high costs mean low flexibility.

The work in [AC06] provide a thorough analysis of the prediction accuracy and adaptability for different load predictors, such as simple moving averages, EMA and non-linear models, such as cubic splines. They conclude that EMA is the most accurate prediction model, at the cost of lower reactivity to load variations compared to cubic splines.

In [MCJM13] the authors provide a series of prediction models for resource utilization in OLTP workloads. The models are built offline based on collected logs during system operation. They have developed two types of models, namely black box and white box models. The former are based on statistical regression and assess future performance based on historical data, whereas the later consider major database components, such as the RAM, CPU, lock contention, and others, to provide more accurate predictions. The main difference between the two models is that black box models are more general but also less effective on unseen data.

In [ADAB11] the authors propose a new approach for estimating the query completion time for batch workloads by considering the interaction, i.e., the dependencies, between the queries. The presented approach samples the space of possible query mixes and fits statistical models to the observed performance and sampled queries.

In [GKD+09] a tool is presented, that is able to predict for short- and long-running queries. Short running queries have typically an execution duration of some milliseconds, whereas long-running queries may take up to some hours of execution time. The prediction tool can be used to determine the system size in terms of resources that is needed to execute a workload given a specific response time constraint. Moreover, it can also be used for capacity planning, i.e., given an expected workload, the tool provides a means to decide whether the system should be scaled up or down. The tool is trained with a set of real customer and commercial synthetic workloads. For the prediction model, the authors have used the canonical correlation analysis, which is an extension of the principal component analysis.

[DÇPU11] presents a model for estimating the impact of concurrency on query performance for OLAP workloads. Concretely, they describe a predictor that is able to determine when each OLAP query from a set of concurrently running queries will finish its execution. The predictor is based on a linear regression model that considers the I/O and CPU overhead to predict the query latency.

[HGR09] proposes a workload classification approach, which reduces the analysis overhead for DBS workloads by clustering similar workload events into classes. Its online nature makes the approach a perfect fit for the workload analysis of the CCQ protocols, which is considered one of the major sources of overhead.

8.4 Data Consistency

Data consistency remains an open and attractive research challenge. The goal of the research is on one hand the definition of new consistency models for emerging application types, and on the other hand, the design of low-cost protocols implementing a certain consistency model.

The trade-offs captured by CAP/PACELC have lead to a plethora of protocols that provide strong consistency, such as 1SR or SI, and incorporate different optimization with the goal of reducing the transaction overhead. The research in this area was pioneered by the protocols described in [KA00a, KA00b], which provide 1SR guarantees for replicated DBSs using efficient group communication. Their development is based on the observation that distributed commit protocols are expensive and generate con-

siderable performance penalty. Along the same lines, in [KPF+13] the authors propose Multi-Data Center Consistency (MDCC), a new distributed commit protocol, that guarantees strong consistency for geo-replicated data. MDCC is more fault tolerant and generates lower message overhead compared to 2PC. It is crucial to reduce message overhead, as in geo-replicated databases the network latency is the main limiting factor to performance [BFG+13, BDF+13].

Google's Spanner, which was motivated by frequent complaints of BigTable clients due to its relaxed consistency model, is a highly scalable distributed database that provides strong 1SR [CD+13]. Spanner's implementation of strong 1SR is based on S2PL and Paxos for the synchronous replication. It uses the novel *True Time* that assigns commit timestamp to transactions in a scalable way. Based on the timestamps it is possible to order globally transactions. Moreover, Spanner allows the implementation of lock-free read-only transactions, which has a considerable impact on the overall performance. F1, which stores data of Google's advertisement business, was one the first customers of Spanner [SOE+12].

Many commercial databases, such as Oracle [Cor] and MySql [MyS], also include weak models and allow clients to choose the appropriate one at the application level, as a consequence of the consistency-latency trade-off. This is in sharp contrast to the aforementioned approaches, which incorporate the optimizations at the protocol level without affecting the strong guarantees. However, if the large deployment of those databases is taken into account, then one can conclude that not every application requires strong consistency. As described in [BDF+13], which surveyed the provided consistency levels of eighteen popular databases, only three of those databases provided serializability as a default model, and eight did not provide that choice at all. It means that weak consistency has been there for quite a while and deployed in commercial applications.Weak models experienced a revival with the invent of the Cloud, which considerably decreased the barrier for the development of large-scale applications that demand always-on guarantees and low latency [ADE12].

Different weaker models than serializability have been proposed, such as 1SR [DS06, EZP05, BBG+95], causal consistency [BFG+12, BSW04, BGHS13, LFKA11], and EC [BLFS12]. Weak models reduce the performance overhead, and are compatible with the high availability requirements. However, at the cost of the semantics (understandability). As we described in Section 3.4, there is programmability-consistency trade-off, which has a high impact on the development and maintenance cost of applications [Ham10, BSW04, BGHS13].

Several approaches have been introduced that deal with the issue of avoiding the inconsistencies of a specific consistency model with a low or no overhead at all. For example, [FLO+05] defines a theory which allows arranging concurrent transaction so that the well-known SI anomalies are avoided, i.e., the resulting effects correspond to a serializable execution. Along the same lines of [FLO+05], in [CRF08] the authors present a new protocol on the basis of an existing SI implementation that is able to detect and prevent possible anomalies at runtime (read and write skew), by retaining the performance advantages of SI, i.e., reads and writes do not block each other.

[BFF+14] defines a formal framework that is able to decide if coordination is necessary for providing serializable execution of transactions, as coordination is the main

bottleneck even in single-copy DBSs. It does so by analyzing application defined invariants, such as integrity constraints, and enforces coordination only if there a possibility of violating the invariants.

In the context of data replication, the serializable generalized snapshot isolation (SGSI) protocol presented in [BHEF11b] allows the DBS to provide 1SR global guarantees when each replica site provides only SI locally. SGSI incorporates a *certifier* which is responsible for certifying the read and write sets of transactions with the goal of avoiding SI inconsistencies so that overall 1SR correctness is provided. SGSI can extract the read and write sets of transactions by applying automatic query transformation, and as they conclude, only the read set certification is necessary for avoiding the SI inconsistencies, which in contrast to common believe.

[SWWW99] defines an algorithm that provides strong 1SR guarantees in federated DBSs where sites provide strong SI locally. The approach enforces a global total order of transactions by controlling their execution order at each site.

In [TvS07b] the authors analyze the different layers (modules) of data consistency protocols and the interactions between the layers. Mainly, they analyze the concurrency control, atomic commitment, and consistent replication. Based on their analysis, they propose the transactional system called TAPIR, that provides strict serializable transaction execution. The novelty of TAPIR lies in the usage of a weakly consistent replication protocol, and is based on the observation that it is not necessary to enforce consistency at each layer of a data consistency protocol. This observation and the design of TAPIR are completely in line with the modular design of C^3. While TAPIR sticks to one consistency model, C^3 can adjust the consistency at runtime based on a cost model, and can choose the most suitable consistency implementation. From the perspective of C^3, TAPIR is a protocol that implements strict serializability. Thus, it would be taken into consideration during the search for the most optimal protocol if strict serializability is to be enforced.

8.5 Tunable and Adjustable Consistency

The development of the abstraction provided to application developers has roughly undergone following phases, namely the development of transactions as an abstraction, the development of NoSQL databases that initially did not provide any transactional guarantees, the development of databases with limited transactional support, such as ElasTras [DEA09], G-Store [DAE10], Relational Cloud, and the return of transactions with MegaStore, Spanner, and others. As we already described the main motivation that led to the development of Spanner was the lack of transactional support by BigTable, which forced developers to code around a lack of transactions [CD+13].

This programmability-consistency trade-off has been the main factor why NoSQL databases in the recent versions, have added support for stronger consistency models. This development is in line with what traditional databases such as Oracle and MySQL have ever since supported, namely a range of different consistency (isolation) models [BDF+13]. However, all have in common that the user is responsible for assigning the appropriate consistency model to the requests (transactions).

Adjustable consistency is a new term coined to describe consistency approaches that adjust both, the model and protocol implementing the model, at runtime automatically without any user intervention based on application requirements and workload.

[YV02] defines application-independent metrics for quantifying inconsistencies (numerical error, order error, and staleness). They present algorithms, included in their middleware called TACT, to bound each of the three metrics. TACT effectively balances the latency and semantics trade-off, by providing better performance compared to strong consistency, and stronger semantics compared to weak consistency.

In [KHAK09] the authors present an adjustable and cost-based consistency protocol, which is based on the assumption that not all data of an application need the same consistency guarantees. The approach introduces a consistency rationing model, which categorizes data in three different categories: A, B, and C. The A category contains data which require 1SR. The C category data are handled with session consistency, whereas B category data is handled with adaptive consistency. Adaptive consistency means that the consistency level changes between 1SR and Session Consistency at runtime depending on the specific policy defined at data level (annotation of the schema).

In [Sch01] a first approach is described to cost-based concurrency control that considers the cost of single operations for choosing the most suitable CCP. However, in contrast to our C^3 it does not consider any infrastructure-related costs for enforcing a selected CCP.

The work in [SKJ16] defines a high-level language called Quelea that standardizes the interface to eventually consistent data stores. Quelea aims at closing the chasm between the high-level consistency requirements of applications, and low-level tunable consistency levels. C^3 fits very will in this context as it allows applications to steer consistency based on costs, and reliefs the developers from the low-level details of the different consistency levels.

8.6 Data Partitioning

Several approaches have addressed the optimal placement of partitions with the goal of reducing expensive distributed transactions and evenly distributing the load [CZJM10, JD11, CLW02, CI93, NR89].

An optimal partition schema should completely avoid distributed transaction and should result in a shared-nothing scenario, in which each site of a DDBSs can completely execute transactions locally without any coordination with the other sites. Although hypothetical, shared-nothing databases are subject to virtually unlimited scale out [Sto86].

Schism [CZJM10] is a graph-based partition approach that was incorporated into Relational Cloud [CJP+11]. Similar to Cumulus, Schism represents the application workload as graph and partitions the graph to determine the schema. However, in contrast to Cumulus it does not consider workload analysis, which proved to be crucial not only for the schema quality, but also for the generated overhead. In contras to Cumulus, Schism is not able to adapt the schema (configuration) at runtime.

Autostore is a fully dynamic partitioning approach that contains that is similar to Cumulus in some regards as it also contains a workload prediction and is able to adapt the partition schema at runtime. Cumulus goes, however, one step further by also incorporating an on-demand reconfiguration, which does not only decrease the overhead for transactions, but it also provides higher availability compared to similar reconfiguration approaches.

In [BCD+11] propose Cloud SQL Server, which uses Microsoft SQL Server as its core, to provide scale-out guarantees to web applications. This combination of the ability to scale-out with a familiar programming model provided by the relational SQL Server is of considerable advantage to applications. The scale-out is achieved using data partitioning on a shared-nothing architecture. However, Cloud SQL Server puts strict constraints on the application data and provides only restricted transaction semantics in order to avoid distributed transactions.

ElasTras [DEA09] supports both a manual mode (based on a domain expert) and an autonomous mode for hash-based partitioning. It supports only so-called minitransactions which have a very rigid structure consisting of compare, read, and write operations which have to be executed in exactly that order [AMS+07]. The structure of minitransactions allows to piggyback their execution onto the first phase of 2PC. Compared to traditional transactions, which require two rounds of messages just to commit, minitransactions can be executed within the 2PC protocols. Cumulus, in contrast, provides full Atomicity, Consistency (ACID) support without limiting the size and structure of transactions.

Accordion [SMA+14] provides an elastic partitioning protocol which is particularly beneficial for the Cloud. It targets the optimal mapping of partitions to sites by considering the affinity of partitions, i.e., the frequency in which they are accessed together by transactions, and the capacity. Moreover, Accordion scales-out dynamically if the workload increases, and can scale-in to save costs if the workload decreases. The combination of Cumulus with Accordion would be the first step towards an integrated data management protocol as defined in Example 4.4, with the former defining the partition schema, and the latter the optimal mapping of partitions to sites.

The online reconfiguration is one the main sources of overhead in adaptive protocols, and there has been considerable research effort conducted in developing low overhead approaches under the term of live migration. In [BCM+12] the authors describe Slacker, that tackles the issue of minimizing the effect of the live migration to queries. It does so by throttling the migration rate of data in order to minimize interference.

Squall [EAT+15] is another reconfiguration approach for partitioned databases that has some commonalities with Cumulus. The main difference is that Cumulus' reconfiguration, driven by user transactions, will lock only the objects to be migrated, whereas Squall will lock entire partitions. The granularity of lock has a considerable impact to the transaction concurrency and thus to the overall performance.

[EDAE11] presents Zephyr, an approach that combines on-demand pull and asynchronous push for the live migration of data in shared nothing databases with support for scale-out and scale-in. It is a considerable improvement compared to other approaches, that either incur a long interruption or must abort transactions during the

migration period. Zephyr will only abort transactions that update the index structure. Cumulus, in contrast, will never abort transactions for the reason of the migration.

8.7 Data Replication

Replication of data serves two main purposes, namely to provide high availability by redundancy, and to distribute the load of read-only transactions. However, as already described, for update transactions data replication generates considerable overhead if strong consistency is a demand. This overhead is defined in terms of synchronization messages that need to be exchanged in order to keep the replicas up to date. As the synchronization requires network communication, it becomes the determining factor to performance in case of geo-replicated DBSs [BDF+13, SFS15].

Different replication protocols have been proposed in literature which mainly differ in the number of sites that are eagerly updated. While ROWA(A) protocols eagerly update all or all available sites, quorum protocols update only a subset of sites [JPAK03, KJP10, TT95]. This reduces the overhead for update transactions at increased cost for read-only transactions. Quorum protocols differ in the size of quorums, and the overhead generated for determining and maintaining the quorums.

In the MQ protocol, each site gets a nonnegative number of votes. The quorums are chosen in such a way so that they exceed half of the available votes. The tree-based quorum protocol defined in [AE90] organizes the sites in a tree structure. The read and write quorums are chosen based on predefined rules that guarantee the intersection property. However, the maintenance cost get rapidly increase in mobile environments as the tree must continuously be adapted to changing environment. Grid quorums organize the sites in a grid and exploit the properties of that structure to reduce the size of the quorums. For example, the rectangle protocol defined in [CAA90], organizes s sites in a grid of r rows and c columns, and defines that a read quorum consists of the sites of one column, whereas a write quorum must access one column and at least one site from the other quorums. A grid is optimal with regards to the size of quorums if it is a square ($r = c$) [JPAK03]. There are further grid protocols, such the ones defined in [KRS93] and [PW97], that either target the reduction of quorums size or the increase of availability.

Most of the existing quorum protocols are static, i.e., the quorums are fixed a priori and do not change. [JM87] defines a dynamic quorum protocol, in which the number of sites necessary to consist a quorum is defined as a function of up-to-date sites. The quorum is continuously adapted without any manual intervention. In [BGS86] policies for dynamic assignment of votes to sites are defined with the goal of increasing system availability and resilience to failures. Moreover, the votes are continuously adapted in case of failures. This is especially important in case of network partitions, as it allows to form majorities so that (a part of) the system remains available.

The number of sites that are accessed as part of read and write quorums is not the only aspect to be considered in order to overcome the PACELC trade-offs. Their properties, such as their cost, load, capacity and the network distance between them must be first-class citizens during the quorum construction [SFS15]. QuAD considers a subset

of these properties add is able to continuously adapt the quorums if these properties change. It does not only demonstrate the feasibility of adaptive quorum protocols, but also the necessity and high relevance of clever strategies for the definition on quorums based on site properties.

In [VS09], the authors propose a replication protocol for managing replicated data in Grids. They distinguish between updateable sites for executing of update transactions, and read-only sites for executing read-only transactions. While an update-transaction will eagerly commit all updateable sites, the updates to read-only sites is done in two modus, namely in the pull and push mode. In former, refresh transactions are initiated by a read-only site to pull versions of data to satisfy freshness demands of read-only transactions, whereas in the latter the updates are proactively propagated to read-only sites by considering their load. QuAD is a multi-master approach that does not put any restrictions on which site can execute which type of transactions.

The PDBREP protocol defined in [ATS+05] also distinguishes between read-only and updatable sites similar to the approach in [VS09]. However, PDBREP supports different data distribution models, from full replication to partial replication and partitioned data.

Lazy propagation algorithms differ in set of parameters they consider in their optimization model. [SS05] provides a survey of existing lazy propagation approaches, and describes a set of parameters for their classifications, such as conflict handling, consistency guarantees, propagation strategies, number of writers (single-master vs. multi-master), synchronization technique (push vs. pull), etc.

In general, there is a trade-off between propagation speed, performance, and availability [WRBK02]. Approaches, such as the ones defined in [PS00, RBSS02, BFG+06, GK85, PL91] tackle with these issue by allowing transactions to define explicitly their freshness requirements and adapt the synchronization process in order to fulfill these requirements (e.g., by reducing the propagation delay).

With regards to conflict detection and resolution, systems such as Bayou [TTP+95] and Coda [KS91] introduce the concept of a semantic conflict, which exploits the domain-knowledge of applications for an automatic detection and resolution of conflicts. It allows applications to specify the notion of conflicts (e.g., two users book the same room for different meetings that overlap in time) together with a resolution policy (e.g., moving one of the meetings to another free room).

The Mariposa system described [SAS+96] defines a replication protocol for processing queries based on an economical model. Each query has budget assigned to it, which depends on its importance. Sites can bid to execute a query or parts of it. Furthermore, the placement of replicas is also done dynamically steered by the economical model in which sites may sell or buy copies in order to maximize profit.

Similar to Mariposa, Skute, a self-organized replication protocol defined in [BPA10], introduces an economic model, which trades the replication (storage and maintenance) costs for the utility a replica provides to user queries. Skute is able to adapt dynamically to query load by finding the optimal location of replicas with regards to their popularity and client proximity.

9

Conclusions and Outlook

IN THIS CHAPTER, we conclude our contributions and point out to directions for future work that can further enhance our contributions.

9.1 Summary

In this thesis, we have presented our vision towards our policy based and modular PolarDBMS, that is able to extract application requirements from SLAs and determine that configuration that best satisfies the specified set of requirements. PolarDBMS is motivated by the fundamental trade-offs faced by distributed data management, by the ever increasing variety of application types, that have diverging requirements towards the data management, and by the dynamic behavior of both applications and the infrastructures that host these applications.

Our bottom-up approach towards the realization of the PolarDBMS' vision has resulted in a set of novel cost and workload-driven protocols that can adapt their behavior at runtime with the goal of always being in a best possible state with regards to the application requirements.

C^3 is based on the observation that applications typically do not care about the exact consistency guarantees, as their main goal is to minimize costs, which can incur do to constraint violations when using a weak consistency model and by excessive usage of Cloud resources in case of strong consistency. C^3 makes the monetary costs a first class citizen of data consistency and by that reliefs application developers from the burden of exactly determining which consistency to be used. Anyways, a decision taken at development time would be based on data that rapidly becomes outdated, and will thus not represent the optimal decision. Application providers have knowledge on what it means in terms of costs to compensate a violated constraint. This inconsistency cost is the only parameter required by C^3 in order to determine the optimal consistency and autonomously adapt consistency at runtime based on the workload.

Cumulus and QuAD are tailored to applications that demand 1SR consistency, and both target the optimization of performance under different availability constraints. While Cumulus is suited for applications whose main concern is performance, and that

neglect availability aspects, QuAD optimizes performance given a fully replicated DBS. QuAD is one of the first protocols to consider properties of the sites consisting the DBS when defining the quorums. Contrary to usual approaches that mainly target the reduction of quorum size, QuAD' primary goal is to avoid weak sites from read and commit paths of transactions as they are the main source of performance degradation.

All protocols have their adaptive behavior in common, a property crucial for applications deployed in the Cloud. The availability of applications to a huge number of users worldwide puts additional challenges to data management, and demands low overhead adaptiveness. Adaptive protocols based on a short term gain lead to a system that is continuously occupied with expensive reconfigurations, and that does not provide any utility to applications. Our protocols incorporate a cost model that trades the cost of adapting the behavior to its gain.

The CCQ protocols have been subject to a series of thorough qualitative and quantitative evaluations in a real Cloud environment. The evaluations demonstrate the feasibility of our concepts and that they lead to considerable gain in cost and performance for applications. Moreover, we have shown that the adaptive behavior of our protocols incurs only a minimal overhead for applications, and that it guarantees safety to applications even in presence of failures. Both properties make our CCQ protocols applicable for practical cases.

9.2 Future Work

When it comes to future work, we propose following directions for each of our contributions, as well as their integration into a cohesive whole.

9.2.1 C^3

When it comes to possible extensions of C^3, we suggest the incorporation of further consistency models and protocols, that would enlarge the configuration space, and by that provide more configuration options. The current version of the cost model neglects failures. It is however well known that failures might have a considerable impact on the 2PC cost. For example, in the case of site failures, the termination protocol might be necessary in order to avoid blocking. It is well known that the termination protocol generates considerable cost [BHG87]. Even without the termination protocol, if a site does not respond, the coordinator might need to resend messages. Thus, the cost model should be extended to consider also failures.

Transactions of different sizes differ in the cost they generate. For example, the AWS S3 cost model (see Section 4.1.1) charges clients (transactions) for each action, but also based on the objects read or written by the actions. The extension of the C^3 model to consider the transaction size would lead to the cost being captured at an even finer-grained level, and by that, provide an even more realistic view on costs.

The handling of multi-class transaction workloads remains an open challenge. In C^3 the classes are determined by their inconsistency costs, and the goal of C^3 is to choose for each class that consistency level that incurs the lowest cost. Other approaches, such

the one described in [Fek05], assign isolation levels to transactions, which in C^3 terms corresponds to the definition of classes, so that a certain consistency level is guaranteed. Depending on the workload, there might be more than one possible assignments of isolation levels that achieve the same consistency level. In that case, the cost would be another parameter that would influence the decision on which assignment should be chosen. Although [Fek05] considers only a limited set of consistency models and protocols, we argue that its combination with C^3 would provide further optimization possibilities from the cost and performance point of view. Moreover, the applications would get understandable and well-known consistency guarantees.

9.2.2 Cumulus

The Cumulus workflow for choosing an appropriate partition schema is steered by a number of parameters. Some of them, such as the frequency threshold that is during the workload analysis, currently need to be manually defined by the user. It means that the decision taken at deployment time for such a critical parameter may not be inline with application behavior at runtime. Moreover, the threshold needs to be continuously adapted to the application workload. For example, its value can increase or decrease based on the stability (volatility of the access patterns), the more stable the patterns the higher the threshold and vice-versa. The Cumulus workload analysis may in general benefit from a machine learning approach that determines optimal values for the different parameters.

Data partitioning does not provide any availability guarantees whatsoever. If a partition fails, the data hosted by that partition become unavailable. It is certainly interesting to enhance Cumulus with the ability to dynamically replicate certain partitions based on availability requirements. The mapping of partitions to sites and the site placement are one of the most challenging yet interesting aspects to be considered [SMA+14, BPA10].

9.2.3 QuAD

In its current version, QuAD considers the load of the sites and the network distance between them measured in terms of RTT to determine the quorums. It assumes that all sites have equal capacity, which is a realistic assumption for a wide range of OLTP applications. However, in context of PolarDBMS, the properties of the underlying infrastructure may also be determined by the application requirements, which may, for example, lead to certain sites being more resilient to failures and more powerful in terms of processing capacity. These aspects should also be considered during the quorum construction.

Similar to C^3 and Cumulus, QuAD also currently requires from the application the definition of some parameters, κ being the most important one. It is a critical parameter, as it has a considerable impact on the performance, availability and cost. The choice of the optimal κ is not trivial as it is highly dependent on the site properties, application workload and application requirements towards availability. Therefore, we plan to incorporate a machine learning approach to determine the optimal κ by considering all aforementioned aspects.

9.2.4 Integrated Data Management Protocols

The CCQ protocols are just an initial step towards the realization of the PolarDBMS vision. In context of PolarDBMS there are different aspects of interest, such as the definition of the policy language, the transformation of high-level SLAs to the policy language, and the definition of the most suitable configuration by considering all specified requirements in a multi-tenant environment. We have followed a bottom-up approach by first developing a set of protocols that currently are considered in isolation. As we discussed in Examples 4.4 and 4.6, the next step would be the definition of a comprehensive model for the I-CCQ protocol, that would integrate data consistency, partitioning, and replication protocols by jointly considering the different application requirements.

The I-CCQ protocol may use Cumulus to partition the data in order to satisfy the performance requirements of the application. Later, it might replicate a subset of data either due to an increased performance demand or stronger availability requirements. As soon as data objects are replicated, the I-CCQ protocol faces additional challenges related to the choice of the optimal consistency level and protocol, if the choice is not predefined by the application. In this case, C^3 can take over the task of choosing that consistency level and/or protocol that generates the lowest overhead in terms of monetary cost and performance.

In the approach described above, I-CCQ is a meta-protocol, which uses the capabilities of the CCQ protocols to choose the most optimal configuration given certain predefined requirements. However, applications may hand over full control to I-CCQ by specifying penalty costs for the violation of the different requirements, such as the availability, latency and consistency. In that case, I-CCQ will choose that configuration, which leads to the lowest overall penalty costs, by exploring the configuration space that is defined by different availability degrees and consistency models, and implementation thereof.

The CCQ protocols consider only a subset of data management properties. However, data management has also other very important properties, like for example data security, data models, data integration, just to mention a few, that can be steered based on cost and application workload. The analysis and the integration of these additional properties increases the problem space and introduces additional dependencies that need to be considered.

9.2.5 Distributed Meta-data Management and Autonomous Decision Making

The CCQ protocols would benefit from a distributed meta-data management in terms of resilience to failures, as the availability of meta-data is crucial for choosing the optimal configuration. High availability comes at a cost and generates considerable communication overhead in case the sites need to exchange meta-data. This is the case if all sites must take a consistent decision. Incorporating models that allow each site to decisions based on partial data would reduce this overhead, as in that case only a minimal or no communication between sites will occur. However, in that case, different sites may reflect different decisions, which might not be suitable for every application scenario.

9.2.6 The Cost of the Optimization

The CCQ protocols are based on the idea of incorporating an adaptive optimization model for reducing the overhead for transactions with the goal of satisfying application requirements. However, there is a cost for the optimization, which incurs due to the additional (meta-) data that needs to be collected, managed and processed. The generated overhead is also determined by the desired accuracy of the information (e.g., workload) that is derived from that data, and the configuration space. There are different approaches to tackle this issue ranging from the development of sophisticated yet low overhead machine learning algorithms, low-level programming optimizations, to giving the control to the client, so that it can trade between the cost and the degree of optimization.

A

C^3 On-the-Fly and On-Demand Reconfiguration

As described in Section 6.3.1, currently C^3 implements a stop-and-copy approach for the site reconciliation during the reconfiguration from EC to 1SR. The stop-and-copy approach has the nice property that once the reconfiguration is finished no further overhead incurs for the user transactions, which is in sharp contrast to the on-the-fly and on-demand approach. However, depending on the number of transactions executed with EC and the number of objects modified by EC transaction, the overhead generated by the stop-and-copy reconfiguration may have a considerable impact on the system availability. In what follows we will describe a possible implementation of an on-the-fly and on-demand approach.

The consistency information that is attached to the modified objects as part of the stop-and-copy approach can be reused to implement the on-the-fly and on-demand reconciliation approach. During the reconfiguration, the *ConsistencyManager* needs to collect the so called *change-set*, which contains all modified objects. The change-set is only relevant during a switch from EC to 1SR, and thus contains only objects modified by EC transactions. An entry in the change-set consists of the object id and the timestamp of its last update (Figure A.1). The *ConsistencyManager* will forward the collected change-sets to all *TransactionManagers*, which will merge them and add the location that contains the most recent value for each of the modified objects. All objects that are already up-to-date at the local site are deleted from the merged change-set. Each transaction that is executed by a specific site must check if any of its accessed objects are included in the change-set. If so, distributed refresh transactions are initiated that will retrieve the most recent values for the objects accessed by the user transactions from the remote sites. The refresh transactions are coordinated by the 2PC with the local site playing the role of the *coordinator* and the remote sites that of the *agents*. If any refresh transaction fails, the user transaction must be aborted, as in that case 1SR consistency cannot guaranteed. Each successfully refreshed object is removed from the change-set. If an update-transaction commits it will eagerly commit all sites, which leads to an implicit refresh of the remote sites. This means that at the commit of an update-transaction each site will remove any modified object from its change-set. Outdated objects may remain

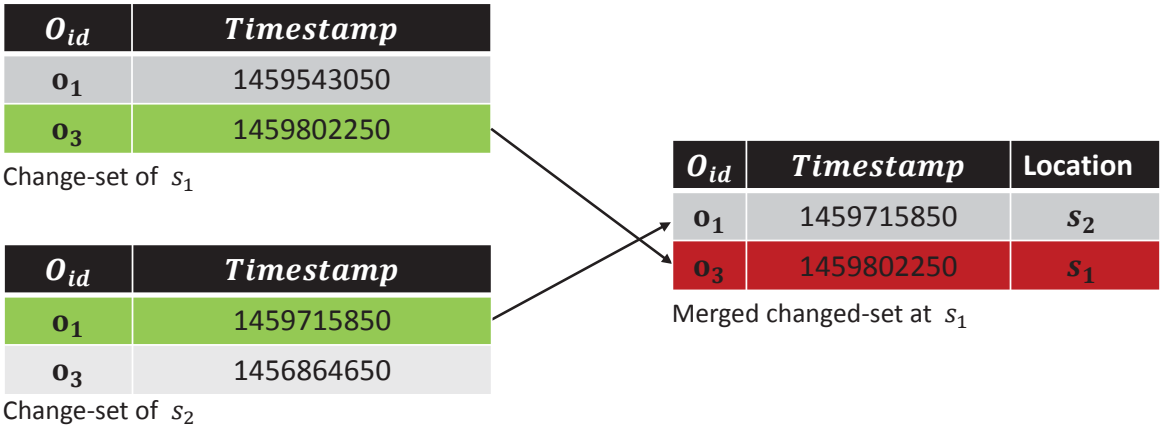


Figure A.1: Change-set of s_1 and s_2 consisting of objects modified by EC transactions and the resulting merged change-set at s_1 that includes the site containing the most recent value for each object. The entry for o_3 is deleted from the merged change-set as s_1 is up-to-date with regards to o_3.

in the change-set during subsequent reconfiguration events, which is a consequence of the on-the-fly and on-demand refresh of objects. This means that the existing merged change-set at each site needs to be merged with new change-sets that will be received from the *ConsistencyManager* in case of a switch from EC to 1SR. The mechanism is similar to what we described above, expect that now also the existing (if not empty) merged change-set needs to be considered, which will result in a new merged change-set that will replace the exiting one.

B

Test Definition File

Listing B.1: Example of a QuAD Test Configuration File

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:TestConfiguration xmlns:ns2="http://ClouTing.dbis.cs.unibas.ch/"
    testName="quad4-2k-assigned-latency">
    <environment defaultAwsAmi="ami-07d2ff70" defaultAwsInstanceType="c1
        .medium" defaultTomcatVersion="7.0.32" numberOfReplicas="4">
        <node name="Replica 1 +5" isAccessReplica="true">
            <module />
            <eazy responseLatency="[5,5]" />
        </node>
        <node name="Replica 2 +10">
            <module />
            <eazy responseLatency="[9,11]" />
        </node>
        <node name="Replica 3 +250">
            <module />
            <eazy responseLatency="[225,275]" />
        </node>
        <node name="Replica 4 +500">
            <module />
            <eazy responseLatency="[450,550]" />
        </node>

        <node name="Quad Manager">
            <module />
            <eazymanager strategy="QUAD_ASSIGNED">
            <quad numberOfKSites="2" scoreType="LATENCY" />
         </eazymanager>
        </node>
        <node name="Metadata Manager">
            <module />
            <metadata />
        </node>
        <node name="Lock Manager">
            <module />
            <lock />
        </node>
```

```
35          <node name="Timestamp Manager">
36              <module />
37              <timestamp />
38          </node>
39
40          <node name="Client" isClient="true" />
41      </environment>
42
43      <client defaultNumberOfTestRuns="1" waitXSecondsBeforeExecuting="30"
            >
44          <tpcc id="BM_1-10" workloadType="rw1000" workerRangeMin="10"
              workerRangeMax="150" singleRunDuration="30"
              workerIncrementStepSize="10" accessReplica="roundrobin"
              resultFolderNamePostfix="" />
45          <tpcc id="BM_3-82" workloadType="rw8020" workerRangeMin="10"
              workerRangeMax="150" singleRunDuration="30"
              workerIncrementStepSize="10" accessReplica="roundrobin"
              resultFolderNamePostfix="" />
46          <tpcc id="BM_6-55" workloadType="rw5050" workerRangeMin="10"
              workerRangeMax="150" singleRunDuration="30"
              workerIncrementStepSize="10" accessReplica="roundrobin"
              resultFolderNamePostfix="" />
47          <tpcc id="BM_11-01" workloadType="rw0100" workerRangeMin="10"
              workerRangeMax="150" singleRunDuration="30"
              workerIncrementStepSize="10" accessReplica="roundrobin"
              resultFolderNamePostfix="" />
48      </client>
49  </ns2:TestConfiguration>
```

Acronyms

Acronym	Description
1SR	One-Copy Serializability
2PC	Two-Phase Commit
2PL	Two-Phase Locking
3PC	Three-Phase Commit
AaaS	Archiving-as-a-Service
ACID	Atomicity, Consistency
AMI	Amazon Machine Image
API	Application Programming Interface
AR	Autoregressive Model
ARIMA	Auto-Regressive Integrated Moving Average
ARMA	Autoregressive Moving Average
AWS	Amazon Web Services
AZ	Availability Zone
B2PL	Basic Two-Phase Locking
C2PL	Conservative Two-Phase Locking
CC	Concurrency Control
CCM	Concurrency Control Model
CCP	Concurrency Control Protocol
CCQ	C3, Cumulus, QuAD
CDF	Cumulative Distribution Function
COCSR	Commit Order-Preserving Conflict Serializable
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
CSR	Conflict Serializable
CSV	Comma Separated Values
DaaS	Data-as-a-Service
DB	Database
DBMS	Database Management System
DBS	Database System
DDBS	Ditributed Database System
EC	Eventual Consistency
EC2	Elastic Compute Cloud
EMA	Exponential Moving Average

Acronym	Description
FIFO	First In, First Out
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IP	Internet Protocol
LWTQ	Log-Write Tree Quorum
MA	Moving Average
MAD	Mean Absolute Deviation
MoVDB	Monoversion Database
MQ	Majority Quorum
MuVDB	Multiversion Database
NoSQL	Not only SQL
OCSR	Order-Preserving Conflict Serializable
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
OWD	One-Way Delay
P2P	Peer-to-Peer
PaaS	Platform-as-a-Service
RAM	Random Access Memory
RC	Replica Control
RDBMS	Relational Database Management System
REST	Representational State Transfer
ROWA	Read-One-Write-All
ROWAA	Read-One-Write-All-Available
RP	Replica Protocol
RTT	Round-Trip Time
S2PL	Strict Two-Phase Locking
SaaS	Software-as-a-Service
SI	Snapshot Isolation
SLA	Service Level Agreement
SLG	Service Level Guarantee
SLO	Service Level Objective
SOA	Service Oriented Architectures
SOAP	Simple Object Access Protocol
SQL	Structured Query Language

Acronym	Description
SS2PL	Strickt Strong Two-Phase Locking
ToC	Terms and Conditions
TPCC	Transaction Processing Performance Council
TQ	Tree Quorum
TWR	Thomas' Write Rule
XML	Extensible Markup Language

Bibliography

[AA13] Ratnadip Adhikari and R. K. Agrawal. An introductory study on time series modeling and forecasting. *CoRR*, abs/1302.6613, 2013.

[Aba09] Daniel J. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12, 2009.

[Aba12] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[AC06] Mauro Andreolini and Sara Casolari. Load prediction models in web-based systems. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools, VALUETOOLS 2006, Pisa, Italy, October 11-13, 2006*, page 27, 2006.

[ACHM11] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[ADAB11] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 449–460, 2011.

[ADE11] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 530–533, 2011.

[ADE12] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. *Data Management in the Cloud: Challenges and Opportunities*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[Ady99] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999. AAI0800775.

[AE90] Divyakant Agrawal and Amr El Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pages 243–254, 1990.

[AE92] Divyakant Agrawal and Amr El Abbadi. The generalized tree quorum protocol: An efficient approach for managing replicated data. *ACM Trans. Database Syst.*, 17(4):689–717, 1992.

[AEM+13] Divyakant Agrawal, Amr El Abbadi, Hatem A. Mahmoud, Faisal Nawab, and Kenneth Salem. Managing geo-replicated data in multi-datacenters. In *Databases in Networked Information Systems - 8th International Workshop, DNIS 2013, Aizu-Wakamatsu, Japan, March 25-27, 2013. Proceedings*, pages 23–43. 2013.

[ALO00] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[ALS10] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB - The Definitive Guide: Time to Relax*. O'Reilly, 2010.

[Ama] Amazon. AWS EC2 User Guide. `http://tinyurl.com/zh8za5y`. Online; accessed on March, 3, 2016.

[AMS+07] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 159–174, 2007.

[ANB+95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[app] App Engine. `https://cloud.google.com/appengine/`. Online; accessed on February, 2, 2016.

[Asl11] Matthew Aslett. How will the database incumbents respond to nosql and newsql. *San Francisco, The*, 451:1–5, 2011.

[ATS+05] Fuat Akal, Can Türker, Hans-Jörg Schek, Yuri Breitbart, Torsten Grabs, and Lourens Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 565–576, 2005.

[awsa] Amazon EC2. `http://aws.amazon.com/de/ec2/`. Online; accessed 06-January-2015.

[awsb] Amazon Elastic Beanstalk. `http://aws.amazon.com/de/elasticbeanstalk/`. Online; accessed on February, 2, 2016.

[azu] Microsoft Azure Virtual Machines. `http://azure.microsoft.com/en-us/services/virtual-machines/`. Online; accessed on February, 2, 2016.

[BB+11] Jason Baker, Chris Bond, et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 223–234, 2011.

[BBG+95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels.

In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 1–10, 1995.

[BCD+11] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting microsoft SQL server for cloud computing. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1255–1263, 2011.

[BCM+12] Sean Kenneth Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüs, and Prashant J. Shenoy. "cut me some slack": latency-aware live migration for databases. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 432–443, 2012.

[BD] Phil Bernstein and Sudipto Das. Rethinking Consistency: A survey of synchronization techniques for replicated distributed databases. `http://tinyurl.com/jrlo756`. Online; accessed on February, 29, 2016.

[BDF+13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

[Ber99] Philip A. Bernstein. Review - A majority consensus approach to concurrency control for multiple copy databases. *ACM SIGMOD Digital Review*, 1, 1999.

[BFF+14] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3):185–196, 2014.

[BFG+06] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 599–610, 2006.

[BFG+12] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 22, 2012.

[BFG+13] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Hat, not CAP: towards highly available transactions. In *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*, 2013.

[BFS00] Peter Buneman, Mary F. Fernandez, and Dan Suciu. Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.

[BG84] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*, 9(4):596–615, 1984.

[BG13] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013.

[BGHS13] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 761–772, 2013.

[BGS86] Daniel Barbará, Hector Garcia-Molina, and Annemarie Spauster. Policies for dynamic vote reassignment. In *Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, USA, May 19-13, 1986*, pages 37–44, 1986.

[BHEF11a] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. One-copy serializability with snapshot isolation under the hood. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 625–636, 2011.

[BHEF11b] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. One-copy serializability with snapshot isolation under the hood. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 625–636, 2011.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[Bir12] Kenneth . Birman. *Guide to Reliable Distributed Systems - Building High-Assurance Applications and Cloud-Hosted Services*. Texts in Computer Science. Springer, 2012.

[BKP93] Judit Bar-Ilan, Guy Kortsarz, and David Peleg. How to allocate network centers. *J. Algorithms*, 15(3):385–415, 1993.

[BLFS12] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 67–86, 2012.

[BN96] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann, 1996.

[BO91] Mary Baker and John Ousterhout. Availability in the sprite distributed file system. *SIGOPS Oper. Syst. Rev.*, 25(2):95–98, April 1991.

[BPA10] Nicolas Bonvin, Thanasis G. Papaioannou, and Karl Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 205–216, 2010.

[BS15a] Filip-Martin Brinkmann and Heiko Schuldt. Towards archiving-as-a-service: A distributed index for the cost-effective access to replicated multi-version data. In *Proceedings of the 19th International Database Engineering & Applications Symposium, Yokohama, Japan, July 13-15, 2015*, pages 81–89, 2015.

[BS15b] Filip-Martin Brinkmann and Heiko Schuldt. Towards archiving-as-a-service: A distributed index for the cost-effective access to replicated multi-version data. In *Proceedings of the 19th International Database Engineering & Applications Symposium, Yokohama, Japan, July 13-15, 2015*, pages 81–89, 2015.

[BSW79] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Trans. Software Eng.*, 5(3):203–216, 1979.

[BSW04] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. From session causality to causal consistency. In *12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2004), 11-13 February 2004, A Coruna, Spain*, pages 152–158, 2004.

[CAA90] Shun Yan Cheung, Mostafa H. Ammar, and Mustaque Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*, pages 438–445, 1990.

[Car13] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[Cas] Apache Cassandra. `http://incubator.apache.org/cassandra/`. Online; accessed on February, 2, 2016.

[CD10] Kristina Chodorow and Michael Dirolf. *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010.

[CD+13] James C. Corbett, Jeffrey Dean, et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[CI93] Wesley W. Chu and Ion Tim Ieong. A transaction-based approach to vertical partitioning for relational database systems. *IEEE Trans. Software Eng.*, 19(8):804–812, 1993.

[CJP+11] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: a database service for the cloud. In *CIDR 2011, Fifth Biennial*

Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings, pages 235–240, 2011.

[CLW02] Chun Hung Cheng, Wing-Kin Lee, and Kam-Fai Wong. A genetic algorithm-based clustering approach for database partitioning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 32(3):215–230, 2002.

[Coo] John Cook. Random Inequalities. `http://tinyurl.com/h78x2ns`. Online; accessed on February, 2, 2016.

[Cor] Corporation. Data Concurrency and Consistency (10g Relese 1).

[Cou] Couchbase. Querying with N1QL. `http://www.couchbase.com/n1ql`. Online; accessed on February, 29, 2016.

[CRF08] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable isolation for snapshot databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 729–738, 2008.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: yahoo!'s hosted data serving platform. volume 1, pages 1277–1288, 2008.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154, 2010.

[CZJM10] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[DAE10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 163–174, 2010.

[DÇPU11] Jennie Duggan, Ugur Çetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 337–348, 2011.

[Dea] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. LADIS 2009 keynote: http://www.cs.cornell.edu/projects/ladis2009/talks/deankeynote-ladis2009.pdf. Online; accessed on February, 2, 2016.

[DEA09] Sudipto Das, Amr El Abbadi, and Divyakant Agrawal. Elastras: An elastic transactional data store in the cloud. In *Workshop on Hot Topics in Cloud Computing, HotCloud'09, San Diego, CA, USA, June 15, 2009*, 2009.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220, 2007.

[DJ11] Jens Dittrich and Alekh Jindal. Towards a one size fits all database architecture. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 195–198, 2011.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, pages 114–121, New York, NY, USA, 1975. ACM.

[DK11] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.

[d'O77] Cecilia R d'Oliveira. An analysis of computer decentralization. Technical report, Technical Memo TM-90, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. 7, 1977.

[DS06] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 715–726, 2006.

[Eat] Kit Eato. How one second could cost Amazon $1.6 billion in sales. `http://tinyurl.com/cxcfle3`. Online; accessed on February, 2, 2016.

[EAT+15] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 299–313, 2015.

[EDAE11] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 301–312, 2011.

[EZP05] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005),26-28 October 2005, Orlando, FL, USA*, pages 73–84, 2005.

[FBS14] Ilir Fetai, Filip-M. Brinkmann, and Heiko Schuldt. Polardbms: Towards a cost-effective and policy-based data management in the cloud. In *Workshops Proceedings of the 30th International Conference on Data Engineering*

Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014, pages 170–177, 2014.

[Fek99] Alain Fekete. Serializability and snapshot isolation. *In Proceedings of Australian Database Conference. Australian Computer Society*, 1999.

[Fek05] Alan Fekete. Allocating isolation levels to transactions. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, pages 206–215, 2005.

[FJB09] Shel Finkelstein, Dean Jacobs, and Rainer Brendle. Principles for inconsistency. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[FK09] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[FMS15] Ilir Fetai, Damian Murezzan, and Heiko Schuldt. Workload-driven adaptive data partitioning and distribution - the cumulus approach. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 1688–1697, 2015.

[fou] Foursquare. `https://de.foursquare.com/`. Online; accessed on February, 2, 2016.

[FR10] Alan David Fekete and Krithi Ramamritham. Consistency models for replicated data. In *Replication: Theory and Practice*, pages 1–17. 2010.

[Fre95] Clark D. French. "one size fits all" database architectures do not work for DDS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 449–450, 1995.

[FS12] Ilir Fetai and Heiko Schuldt. Cost-based data consistency in a data-as-a-service cloud environment. In *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 526–533, 2012.

[FS13] Ilir Fetai and Heiko Schuldt. SO-1SR: towards a self-optimizing one-copy serializability protocol for data management in the cloud. In *Proceedings of the fifth international workshop on Cloud data management, CloudDB 2013, San Francisco, California, USA, October 28, 2013*, pages 11–18, 2013.

[GBR14] Felix Gessert, Florian Bucklers, and Norbert Ritter. Orestes: A scalable database-as-a-service architecture for low latency. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 215–222, 2014.

[GD01] Andreas Geppert and Klaus R. Dittrich. Component database systems: Introduction, foundations, and overview. In *Compontent Database Systems*, pages 1–28. 2001.

[GFW+14] Felix Gessert, Steffen Friedrich, Wolfram Wingerath, Michael Schaarschmidt, and Norbert Ritter. Towards a scalable and unified REST API for cloud data stores. In *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, pages 723–734, 2014.

[GG09] Robert L. Grossman and Yunhong Gu. On the varieties of clouds for data intensive computing. *IEEE Data Eng. Bull.*, 32(1):44–50, 2009.

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 173–182, 1996.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*, pages 150–162, 1979.

[GK85] Dieter Gawlick and David Kinkade. Varieties of concurrency control in IMS/VS fast path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[GKD+09] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 592–603, 2009.

[GL02] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[Gnu] Gnutella. Gnutella. `http://www.gnutellaforums.com/`. Online; accessed on February, 12, 2016.

[Goo] Google. Google Apps. `https://www.google.com/work/apps/business/`. Online; accessed on February, 2, 2016.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154, 1981.

[GSS+13] Jana Giceva, Tudor-Ioan Salomie, Adrian Schüpbach, Gustavo Alonso, and Timothy Roscoe. COD: database / operating system co-design. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[Hal10] Code Hale. You can't sacrifice partition tolerance. `http://tinyurl.com/j9fhfnj`, 2010. Online; accessed on February, 2, 2016.

[Ham10] James Hamilton. I love eventual consistency but... `http://bit.ly/hamilton-eventual`, 2010. Online; accessed on February, 2, 2016.

[HGR09] Marc Holze, Claas Gaidies, and Norbert Ritter. Consistent on-line classification of dbs workload events. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 1641–1644, 2009.

[HLR10] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[Hof] Todd Hoff. Myth: Eric Brewer On Why Banks Are BASE Not ACID - Availability Is Revenue. `http://tinyurl.com/bpgrh3s`. Online; accessed on February, 2, 2016.

[HP94] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Trans. Comput. Syst.*, 12(1):58–89, February 1994.

[HS98] JW Harris and H Stocker. Maximum likelihood method. *Handbook of Mathematics and Computational Science*, 1:824, 1998.

[IH12] Fetai I. and Schuldt H. Cost-based adaptive concurrency control in the cloud. Technical report, University of Basel, February 2012.

[ins] Instagram. `https://instagram.com/`. Online; accessed on February, 2, 2016.

[JD11] Alekh Jindal and Jens Dittrich. Relax and let the database do the partitioning online. In *Enabling Real-Time Business Intelligence - 5th International Workshop, BIRTE 2011, Held at the 37th International Conference on Very Large Databases, VLDB 2011, Seattle, WA, USA, September 2, 2011, Revised Selected Papers*, pages 65–80, 2011.

[JG77] James B. Rothnie Jr. and Nathan Goodman. A survey of research and development in distributed database management. In *Proceedings of the Third International Conference on Very Large Data Bases, October 6-8, 1977, Tokyo, Japan.*, pages 48–62, 1977.

[JM87] Sushil Jajodia and David Mutchler. Dynamic voting. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 227–238, 1987.

[JPAK03] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.

[KA00a] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, A new way to implement database replication. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 134–143, 2000.

[KA00b] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, A new way to implement database replication. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 134–143, 2000.

[Kaz] Kazaa. Kazaa. `http://www.kazaa.com/`. Online; accessed on February, 12, 2016.

[KHAK09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[KJP10] Bettina Kemme, Ricardo Jiménez-Peris, and Marta Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1998.

[KK10] Donald Kossmann and Tim Kraska. Data management in the cloud: Promises, state-of-the-art, and open questions. *Datenbank-Spektrum*, 10(3):121–129, 2010.

[KKL+10] Donald Kossmann, Tim Kraska, Simon Loesing, Stephan Merkli, Raman Mittal, and Flavio Pfaffhauser. Cloudy: A modular cloud storage system. *PVLDB*, 3(2):1533–1536, 2010.

[KL02] Alexander Keller and Heiko Ludwig. Defining and monitoring service-level agreements for dynamic e-business. In *Proceedings of the 16th Conference on Systems Administration (LISA 2002), Philadelphia, PA, November 3-8, 2002*, pages 189–204, 2002.

[Klo10] Rusty Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[KPF+13] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: multi-data center consistency. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 113–126, 2013.

[KRS93] Akhil Kumar, Michael Rabinovich, and Rakesh K. Sinha. A performance study of general grid structures for replicated data. In *Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, Pennsylvania, USA, May 25-28, 1993*, pages 178–185, 1993.

[KS91] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *SIGOPS Oper. Syst. Rev.*, 25(5):213–225, September 1991.

[KS00] Samir Khuller and Yoram J. Sussmann. The capacitated *K*-center problem. *SIAM J. Discrete Math.*, 13(3):403–418, 2000.

[Kuh10] Harold W. Kuhn. The hungarian method for the assignment problem. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 29–47. 2010.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam02] Leslie Lamport. Paxos made simple, fast, and byzantine. In *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, pages 7–9, 2002.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 401–416, 2011.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[LZGS84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance - computer system analysis using queueing network models*. Prentice Hall, 1984.

[MCJM13] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 301–312, 2013.

[McW08] David T McWherter. *Sharing DBMS among Multiple Users while Providing Performance Isolation: Analysis and Implementation*. ProQuest, 2008.

[MD88] Mark S. Miller and Eric K. Drexler. Markets and computation: Agoric open systems. In Bernardo A. Huberman, editor, *The Ecology of Computation*. North-Holland, Amsterdam, 1988.

[Mel00] Jim Melton. *Understanding the New SQL: A Complete Guide, Second Edition, Volume I*. Morgan Kaufmann, 2000.

[Mon] MongoDB. `http://www.mongodb.org/`. Online; accessed on February, 2, 2016.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[MyS] MySQL. `https://www.mysql.com/`. Online; accessed on February, 2, 2016.

[MZ] Barbara Mutinelli and Sylvia Zwettler. What's behind these terms? `http://tinyurl.com/h7ugmb6`. Online; accessed on February, 29, 2016.

[Nap] Napster. Napster. `http://www.napster.com/`. Online; accessed on February, 12, 2016.

[Nau] Robert F. Nau. ARIMA models for time series forecasting. http://tinyurl.com/gsgd5hm. Online; accessed on February 11, 2016.

[NR89] Shamkant B. Navathe and Minyoung Ra. Vertical partitioning for database design: A graphical algorithm. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989.*, pages 440–450, 1989.

[NSS03] Wolfgang Nejdl, Wolf Siberski, and Michael Sintek. Design issues and challenges for rdf-and schema-based peer-to-peer systems. *ACM SIGMOD Record*, 32(3):41–46, 2003.

[oraa] Oracle 10g Data Concurrency and Consistency. http://tinyurl.com/zfuwmn9. Online; accessed on February, 2, 2016.

[orab] Oracle NoSQL Consistency. http://www.oracle.com/technetwork/database/nosqldb/overview/nosql-transactions-497227.html. Online; accessed on February, 2, 2016.

[Ori] OrientDB. OrientDB Manual. http://tinyurl.com/jyj3rd3. Online; accessed on February, 2, 2016.

[OST03] Beng Chin Ooi, Yanfeng Shu, and Kian-Lee Tan. Relational data sharing in peer-based data management systems. *SIGMOD Record*, 32(3):59–64, 2003.

[ÖV11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[PB93] Gustav Pomberger and Günther Blaschek. *Software Engineering - Prototyping und objektorientierte Software-Entwicklung*. Hanser, 1993.

[PL91] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991.*, pages 377–386, 1991.

[PS00] Esther Pacitti and Eric Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB J.*, 8(3-4):305–318, 2000.

[PST+97] Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. pages 288–301, 1997.

[PW97] David Peleg and Avishai Wool. Crumbling walls: A class of practical and efficient quorum systems. *Distributed Computing*, 10(2):87–97, 1997.

[Qia] Shawn Qian. Enumerative Combinatorics. http://tinyurl.com/h5q8c9w. Online; accessed on February, 29, 2016.

[RBSS02] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS - A freshness-sensitive coordination middleware for a cluster of OLAP components. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 754–765, 2002.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[Sal] Salesforce. Bring your CRM to the future. `https://www.salesforce.com/crm/`. Online; accessed on February, 2, 2016.

[SAS+96] Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker, and Andrew Yu. Data replication in mariposa. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 485–494, 1996.

[SÇ05] Michael Stonebraker and Ugur Çetintemel. "one size fits all": An idea whose time has come and gone (abstract). In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 2–11, 2005.

[Sch01] Heiko Schuldt. Process locking: A protocol based on ordered shared locks for the execution of transactional processes. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, 2001.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[SFS15] Alexander Stiemer, Ilir Fetai, and Heiko Schuldt. Comparison of eager and quorum-based replication in a cloud environment. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 1738–1748, 2015.

[sima] SimpleDB Consistency. `http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/ConsistencySummary.html`. Online; accessed on February, 2, 2016.

[Simb] Amazon SimpleDB. `http://aws.amazon.com/de/simpledb/`. Online; accessed on February, 2, 2016.

[siP] Serializable Isolation versus True Serializability. `http://tinyurl.com/zpr8eaf`. Online; accessed on February, 2, 2016.

[SK09] Marc Shapiro and Bettina Kemme. Eventual consistency. In *Encyclopedia of Database Systems*, pages 1071–1072. 2009.

[Ske81] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981*, pages 133–142, 1981.

[SKG+12] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 18, 2012.

[SKJ16] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Representation without taxation: A uniform, low-overhead, and high-level interface to eventually consistent key-value stores. *IEEE Data Eng. Bull.*, 39(1):52–64, 2016.

[SKS05] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005.

[SMA+14] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB*, 7(12):1035–1046, 2014.

[SMAdM08] Raúl Salinas-Monteagudo, Francesc D. Muñoz-Escoí, José Enrique Armendáriz-Iñigo, and José Ramón González de Mendívil. A performance evaluation of g-bound with a consistency protocol supporting multiple isolation levels. In *On the Move to Meaningful Internet Systems: OTM 2008 Workshops, OTM Confederated International Workshops and Posters, ADI, AWeSoMe, COMBEK, EI2N, IWSSA, MONET, OnToContent + QSI, ORM, PerSys, RDDS, SEMELS, and SWWS 2008, Monterrey, Mexico, November 9-14, 2008. Proceedings*, pages 914–923, 2008.

[SMK+01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[SOE+12] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, et al. F1: the fault-tolerant distributed rdbms supporting google's ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778. ACM, 2012.

[SPB+11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski, et al. A comprehensive study of convergent and commutative replicated data types. 2011.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[SS11] RobertH. Shumway and DavidS. Stoffer. Arima models. In *Time Series Analysis and Its Applications*, Springer Texts in Statistics, pages 83–171. Springer New York, 2011.

[Sto86] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[Sto08] Michael Stonebraker. Technical perspective - one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.

[SW00] Ralf Schenkel and Gerhard Weikum. Integrating snapshot isolation into transactional federation. In *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*, pages 90–101, 2000.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[SWWW99] Ralf Schenkel, Gerhard Weikum, Norbert Weißenberg, and Xuequn Wu. Federated transaction management with snapshot isolation. In *Transactions and Database Dynamics, Eight International Workshop on Foundations of Models and Languages for Data and Objects, Schloß Dagstuhl, Germany, September 27-30, 1999, Selected Papers*, pages 1–25, 1999.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[TMS+14] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *PVLDB*, 8(3):245–256, 2014.

[TPK+13] Pinar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: analyzing tpc's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 17–28, 2013.

[TPP] TPPC Benchmark. `http://www.tpc.org/tpcc/`. ONLINE; accessed January-2015.

[TSJ81] Robert R Tenney and Nils R Sandell Jr. Detection with distributed sensors. *IEEE Transactions on Aerospace Electronic Systems*, 17:501–510, 1981.

[TT95] Peter Triantafillou and David J. Taylor. The location based paradigm for replication: Achieving efficiency and availability in distributed systems. *IEEE Trans. Software Eng.*, 21(1):1–18, 1995.

[TTP+95] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 172–183, 1995.

[TvS07a] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms (2. ed.)*. Pearson Education, 2007.

[TvS07b] Andrew S. Tanenbaum and Maarten van Steen. Distributed systems - principles and paradigms (2. ed.). 2007.

[Vis93] Ramanarayanan Viswanathan. A note on distributed estimation and sufficiency. *IEEE Transactions on Information Theory*, 39(5):1765–1767, 1993.

[Vog09] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[VP04] Patrick Valduriez and Esther Pacitti. Data management in large-scale P2P systems. In *High Performance Computing for Computational Science - VECPAR 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004, Revised Selected and Invited Papers*, pages 104–118, 2004.

[VS09] Laura Cristiana Voicu and Heiko Schuldt. How replicated data management in the cloud can benefit from a data grid protocol: the re: Gridit approach. In *Proceedings of the First International CIKM Workshop on Cloud Data Management, CloudDb 2009, Hong Kong, China, November 2, 2009*, pages 45–48, 2009.

[W+74] W. Wulf et al. HYDRA: the Kernel of a Multiprocessor Operating System. *Commun. ACM*, 1974.

[Web12] Jim Webber. A programmatic introduction to neo4j. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, pages 217–218, 2012.

[WRBK02] An-I Wang, Peter L. Reiher, Rajive L. Bagrodia, and Geoffrey H. Kuenning. Understanding the behavior of the conflict-rate metric in optimistic peer replication. In *13th International Workshop on Database and Expert Systems Applications (DEXA 2002), 2-6 September 2002, Aix-en-Provence, France*, pages 757–764, 2002.

[WTK+08] Lizhe Wang, Jie Tao, Marcel Kunze, Alvaro Canales Castellanos, David Kramer, and Wolfgang Karl. Scientific cloud computing: Early definition and experience. In *10th IEEE International Conference on High Performance Computing and Communications, HPCC 2008, 25-27 Sept. 2008, Dalian, China*, pages 825–830, 2008.

[WV02] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[YSY09] Fan Yang, Jayavel Shanmugasundaram, and Ramana Yerneni. A scalable data platform for a large number of small applications. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[YV02] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

[ZP08] Vaide Zuikeviciute and Fernando Pedone. Correctness criteria for database replication: Theoretical and practical aspects. In *On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*, pages 639–656, 2008.

Index

Curriculum Vitae

Ilir Fetai

August 21, 1980	Born in Cellopek, Tetovo, R. Macedonia Son of Menduale and Imer Citizen of Switzerland
1987–1991	Primary School, Cellopek/Tetovo, R. Macedonia
1991–1995	Secondary School, Cellopek/Tetovo, R. Macedonia
1995–1999	High School, Tetovo, R. Macedonia
1999–2004	Bachelor of Science in Computer Science University of Applied Sciences, Bern, Switzerland
2008–2011	Master of Science in Computer Science University of Basel, Basel, Switzerland
2011–2016	Research and teaching assistant in the group of Prof.Heiko Schuldt Database and Information Systems, University of Basel, Switzerland