Clouder: A Flexible Large Scale Decentralized Object Store

Ricardo Manuel Pereira Vilaça

October 17, 2012

Agradecimentos

Foi em 2007 que começou esta aventura. É incrível como 5 anos passaram tão depressa. Tudo isto foi possível por duas razões. Primeiro, por estar a trabalhar numa área entusiasmante e que me dá muito prazer trabalhar. Segundo, porque tive o apoio pessoal e/ou cientifico de um conjunto de pessoas e instituições sem as quais não teria sido possível chegar a bom porto. Aproveito para expressar aqui os meus sinceros agradecimentos aos que mais significativamente contribuíram para que esta aventura fosse possível.

Gostaria de começar por agradecer ao meu orientador Rui Oliveira. Antes de mais por me desafiar a entrar nesta aventura e por ter excedido o papel de orientador cientifico desta tese. Ao longo destes 5 anos foi um pilar e orientou-me pelo rumo certo nos momentos mais difíceis. Para além disso, agradeço-lhe a oportunidade de alargar a minha formação académica em atividades não diretamente relacionadas com a tese: participação na elaboração de propostas de financiamento a projetos; revisão de artigos científicos; apoio à lecionação de aulas de sistemas distribuídos; e a co-orientação de alunos. Muito obrigado pela sua dedicação.

Ao longo deste tempo que passei no Grupo de Sistemas Distribuídos (GSD) passei por grandes experiências profissionais e pessoais. Isto deve-se ao excelente ambiente do GSD, proporcionado pelas pessoas que o compõe. Antes de mais gostaria de agradecer a todos os atuais e antigos colegas de gabinete: Alfrânio Correia, Ana Nunes, Bruno Costa, Filipe Campos, Francisco Cruz, Francisco Maia, Luís Ferreira, Luís Soares, João Paulo, José Marques, Miguel Borges, Miguel Matos, Nelson Gonçalves, Nuno Carvalho, Nuno Castro, Nuno Lopes, Paulo Jesus, Pedro Gomes e Ricardo Gonçalves. Depois, a todos os professores do grupo e em particular, gostaria de agradecer ao professor José Orlando pela total disponibilidade sempre que recorri à sua ajuda.

O trabalho torna-se mais agradável quando os nossos colegas são também nossos

amigos. Assim sendo gostaria de agradecer a todos os membros dos ”Os Sem Estatuto (OSE)” por todos os momentos de convívio. Para além dos colegas de gabinete, que também são elementos dos OSE, gostaria de agradecer à Alexandra Silva, ao Daniel Machado, ao Jácome Cunha, ao João Fernandes, ao Paulo Silva, e ao Tiago Alves.

Gostaria de agradecer também aos meus amigos fora do ambiente de trabalho pela sua presença nos grandes momentos da minha vida e pelo seu apoio incondicional. Gostaria de agradecer particularmente aos amigos da LESI, a todos os membros da família Garrafónica e a todos os membros da minha CVX. Obrigado a todos por fazerem parte da minha vida.

De forma muito especial gostaria de agradecer o apoio da minha família, os meus pais, José e Cândida, a minha irmã Leninha e o meu irmão Miguel. Os meus pais sacrificaram-se para que eu e os meus irmãos pudéssemos estudar. Não há palavras que descrevam a gratidão que eu sinto por tudo o que fizeram por mim. Sem eles nunca teria chegado aqui.

Finalmente, porque a vida é muito mais do que trabalho e tive a feliz oportunidade de durante este tempo para além do doutoramento ter vivido outro grande evento na minha vida, o meu casamento. À Ana por estar sempre ao meu lado e por me motivar nos momentos mais difíceis. Sem a sua energia e apoio incondicional esta tese não teria sido possível. Obrigado por tudo. AMO-TI pala sempre.

Algumas instituições apoiaram o trabalho apresentado nesta tese. A Fundação para a Ciência e Tecnologia (FCT) apoiou este trabalho através de uma bolsa de doutoramento (SFRH/BD/38529/2007) durante 4 anos. O Departamento de Informática e o HasLab providenciaram-me condições para o meu doutoramento. A HP Research atribuiu-me uma bolsa para participar na conferência SOSP 2011. Parte deste trabalho foi financiado por fundos FEDER através do Programa Operacional Fatores de Competitividade – COMPETE e por Fundos Nacionais através da FCT – Fundação para a Ciência e a Tecnologia no âmbito do projeto Stratus/FCOMP-01-0124-FEDER-015020 e pelo European Union Seventh Framework Programme (FP7) através da bolsa número 257993 do projeto CumuloNimbo.

FCT
Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

POPH
PROGRAMA OPERACIONAL POTENCIAL HUMANO

UNIÃO EUROPEIA
Fundo Social Europeu

Braga, Julho de 2012
Ricardo Vilaça

Clouder: A Flexible Large Scale Decentralized Object Store

Large scale data stores have been initially introduced to support a few concrete extreme scale applications, such as social networks. Their scalability and availability requirements often sacrifice richer data and processing models, and even elementary data consistency. In strong contrast with traditional relational databases (RDBMS), large scale data stores present very simple data models and APIs, lacking most of the established relational data management operations, and presenting relaxed consistency guarantees, providing eventual consistency.

Nowadays, with a number of available and mature alternatives, there is an increasing willingness to use them in a wider and more diverse spectrum of applications by skewing the current trade-off toward the needs of common business users, and easing the migration from current RDBMS. This is particularly so when used in the context of a Cloud solution, such as in a Platform as a Service (PaaS).

This thesis aims at reducing the gap between traditional RDBMS and large scale data stores by seeking mechanisms to provide additional consistency guarantees, and higher-level data processing primitives in large scale data stores. The devised mechanisms should not hinder the scalability and dependability of large scale data stores. Regarding higher-level data processing primitives, this thesis explores two complementary approaches: extending data stores with additional operations such as general multi-item operations; and coupling data stores with other existing processing facilities without hindering scalability.

We address these challenges with a new architecture for large scale data stores, efficient multi-item access for large scale data stores, and SQL processing on large scale data stores. The novel architecture allows one to find the right trade-offs among flexible usage, efficiency, and fault tolerance. To efficiently support multi-item access, we extend first generation large scale data store's data models with tags, and a multi-tuple data placement strategy, allowing to efficiently store and retrieve large sets of related data at once. For efficient SQL support on scalable data stores, we devised design modifications to existing relational SQL query engines, allowing them to be distributed.

We demonstrate our approaches with running prototypes and extensive experimental evaluation using proper workloads.

Clouder: Armazenamento e processamento de dados de forma flexível e descentralizada

Os sistemas de armazenamento de dados de grande escala foram inicialmente desenvolvidos para suportar um leque restrito de aplicações de extrema escala, como as redes sociais. Os requisitos de escalabilidade e elevada disponibilidade levaram a sacrificar modelos de dados e processamento enriquecidos e até a coerência dos dados. Em oposição aos tradicionais sistemas relacionais de gestão de bases de dados (SRGBD), os sistemas de armazenamento de dados de grande escala apresentam modelos de dados e APIs muito simples. Em particular, evidencia-se a ausência de muitas operações de gestão de dados relacionais existentes e o relaxamento das garantias de coerência, fornecendo coerência futura.

Atualmente, com o número de alternativas disponíveis e maduras, existe o crescente interesse em usá-los num maior e diverso leque de aplicações, orientando o atual compromisso para as necessidades dos típicos clientes empresariais e facilitando a migração a partir das atuais SRGBD. Isto é particularmente importante no contexto de soluções *cloud* como plataformas como um serviço (*PaaS*).

Esta tese tem como objetivo reduzir a diferença entre os tradicionais SRGBDs e os sistemas de armazenamento de dados de grande escala, procurando mecanismos que providenciem garantias de coerência mais fortes e primitivas com maior capacidade de processamento. Os mecanismos desenvolvidos não devem comprometer a escalabilidade e fiabilidade dos sistemas de armazenamento de dados de grande escala. No que diz respeito às primitivas com maior capacidade de processamento, esta tese explora duas abordagens complementares: a extensão de sistemas de armazenamento de dados de grande escala com operações genéricas de multi-objeto e a junção dos sistemas de armazenamento de dados de grande escala com mecanismos existentes de processamento e interrogação de dados, sem colocar em causa a escalabilidade dos mesmos.

Para isso apresentámos uma nova arquitetura para os sistemas de armazenamento de dados de grande escala, acesso eficiente a múltiplos objetos, e processamento de SQL sobre sistemas de armazenamento de dados de grande escala. A nova arquitetura permite encontrar os compromissos adequados entre flexibilidade, eficiência e tolerância a faltas. De forma a suportar eficientemente o acesso a múltiplos objetos estendemos o modelo de dados de sistemas de armazenamento de dados de grande escala da primeira geração com palavras-chave e definimos uma estratégia de colocação de dados para múltiplos objetos, que permite eficientemente armazenar e obter grandes quantidades de dados de uma só vez. Para o suporte eficiente de SQL sobre sistemas de armazenamento de dados de grande escala, analisámos a arquitetura dos motores de interrogação de SRGBDs e fizemos alterações que permitem que sejam distribuídos.

As abordagens propostas são demonstradas através de protótipos e uma avaliação experimental exaustiva recorrendo a cargas adequadas baseadas em aplicações reais.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Massive-scale distributed computing is a challenge at our doorstep. Digital data storage has reached unprecedented levels with the ever increasing demand of individuals and organizations for information in electronic formats, ranging from the disposal of traditional media storage media of music, photos and movies to the rise of massive applications. Such large volumes of data tend to disallow their centralized storage and processing, making extensive and flexible data partitioning unavoidable. Particularly, Web-scale applications manage large amounts of data that are generated continuously and systems processing such data run under rigid time constraints.

Relational Database Management Systems (RDBMS) have been the key technology for the management of structured data. RDBMS offer developers both a rich processing interface, Structured Query Language (SQL), and a transactional model that helps developers to ensure data consistency. SQL is the standard language for accessing and manipulating databases. Most database applications and tools that have been developed over the years are coupled to SQL. Transactions in RDBMS have the well-known ACID properties: atomicity, consistency, isolation and durability. ACID properties guarantee that the integrity and consistency of the data is maintained, despite concurrent accesses and faults (Garcia-Molina et al. 2008).

However, traditional relational database management systems are based on highly centralized, rigid architectures that fail to cope with the increasing demand for scalability and dependability, or are not cost-effective. High performance RDBMS invariably rely on mainframe architectures or clustering based on a centralized shared storage infrastructure. Although easy to setup and deploy, these often require large investments upfront and present severe scalability limitations.

This was the breeding ground for a new generation of elastic data management solutions that can scale not only in the sheer volume of data which can be held, but also in how required resources can be provisioned dynamically and incrementally (DeCandia et al. 2007; Cooper et al. 2008; Chang et al. 2006; Lakshman and Malik 2010). Furthermore, the underlying business model supporting these efforts requires the ability to simultaneously serve and adapt to multiple tenants with diverse performance and dependability requirements, which add to the complexity of the whole system. These first generation large scale data stores were built by major Internet players, such as Google, Amazon, Microsoft, Facebook and Yahoo, by embracing the Cloud computing model.

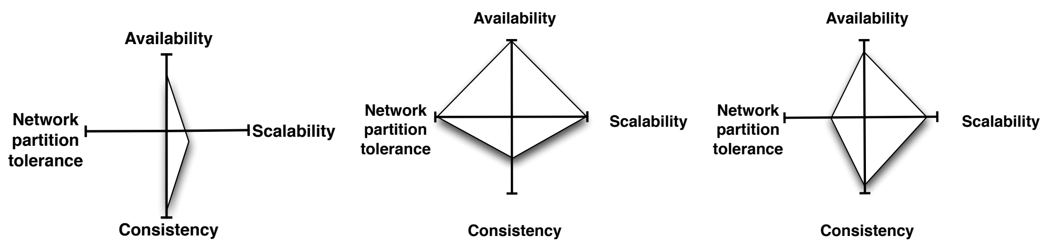
After these solutions of major companies several community or commercial implementations of large scale data stores emerged. They are a subset of the so called NoSQL databases[1]: Cassandra[2], HBase[3], Oracle NoSQL Database[4], MongoDB[5], CouchDB[6], RavenDB[7], Riak[8], among others.

NoSQL databases may be categorized according to the way they store the data and fall under categories, such as key-value stores, column stores, document store databases, and graph databases. Key-value stores are the most common and allow the application to store its data in a schema-less way, only defining a key per item.

In strong contrast with traditional relational databases, large scale data stores present very simple data models and APIs, lacking most of the established relational data management operations, and presenting relaxed consistency guarantees, providing eventual consistency (Vogels 2009). The reduced flexibility compared to traditional RDBMS is compensated by significant gains in scalability to hundreds of nodes, and performance for certain data models. However, most of the times this implies to think *a priori* about data access patterns, and having queries defined upfront.

Large scale data stores are often highly optimized to retrieve and append operations, and often offer little functionality beyond single item. All large scale data stores have simple data models with no rigid schema. They offer custom and simple key-value interfaces that allow applications to insert, query, and remove individual items

[1]http://nosql-database.org
[2]http://cassandra.apache.org
[3]http://hadoop.apache.org
[4]http://www.oracle.com/technetwork/database/nosqldb/overview/index.html
[5]http://www.mongodb.org
[6]http://couchdb.apache.org
[7]https://github.com/ravendb/ravendb
[8]http://wiki.basho.com



(a) RDBMS (b) Large Scale Data Stores (c) Clouder

Figure 1.1: CAP

or at most range queries based on the primary key of the item. Indeed, besides basic store, retrieve and scan primitives, current systems lack any filtering or processing capabilities. This is unfortunate as all of the most useful data management operations, such as filtering, joining, grouping, ordering and counting, are done outside the system. These systems require more general and complex multi-item queries to be done asynchronously and outside the system, using some implementation of the the Map Reduce (Dean and Ghemawat 2008) programming model: Yahoo's PigLatin (Olston et al. 2008), Google's Sawzall (Pike et al. 2005), Microsoft's LINQ (Meijer 2011).

Regarding consistency guarantees, large scale data stores rely on well-known distributed systems techniques to be elastic and highly available. The CAP theorem (Brewer 2000) states that in a distributed system it is impossible to simultaneously provide: network partition tolerance, strong consistency and high availability. While traditional RDBMS focus on availability and consistency, Figure 1.1(a), most large scale data stores focus on applications that have minor consistency requirements, Figure 1.1(b). They replace the traditional transactional serializability (Bernstein et al. 1987) or linearizability (Herlihy and Wing 1990) strict criteria by eventual consistency.

New generation large scale data stores are in strong contrast with traditional relational databases and are in disjoint design spaces. Large scale stores forfeit complex relational and processing facilities, and most strikingly, transactional guarantees common in traditional RDBMS. By doing so, they focus on a specific narrow trade-off among consistency, availability, performance, scale, and migration cost that tightly conforms with their very large motivating application scenarios. They focus on applications that have minor consistency requirements and can favor availability with an increasing complexity at the application logic.

However, in most enterprises where there isn't a large in-house research development team for application customization and maintenance, it is hard to add this complex layer to the application. As a result, it is hard to provide a smooth migration path for existing applications based on relational databases, and using a SQL-based interface, even when using modern Web-based multi-tier architectures. This is a hurdle to the adoption of Cloud computing by a wider potential market; thus, a limitation to the long term profitability of businesses model.

1.1 Problem statement and objectives

New generation large scale data stores and traditional RDBMS are in disjoint design spaces, and there is a huge gap between them. This thesis aims at reducing this gap by seeking mechanisms to provide additional consistency guarantees, and higher-level data processing primitives in large scale data stores, Figure 1.1(c). Regarding higher-level data processing primitives, this thesis explores two complementary approaches: extending data stores with additional operations, such as general multi-item operations; and coupling data stores with other existing processing facilities.

This approximation between large scale data stores provides the current trade-off to lead toward the needs of common business users, and eases the migration from current RDBMS. The devised mechanisms should not hinder the scalability and dependability of large scale data stores.

1.2 Contributions

The main contributions of this thesis are:

- A new architecture for large scale data stores - allowing to find the right trade-offs among flexible usage, efficiency, and fault tolerance by a clear separation of concerns between different functional aspects of the system, which should be addressed at different abstractions levels with different goals and assumptions.
- Efficient multi-item access for large scale data stores extending first generation large scale data store's data models with tags, and a multi-tuple data placement strategy that allows to efficiently store and retrieve large sets of related data at once. Multi-tuple operations leverage disclosed data relations to manipulate sets of comparable or arbitrarily related elements.

- Design modifications to existing relational SQL query engines allowing them to be distributed, and efficiently run SQL on a scalable data store while being able to scale with the data store.

1.3 Results

The thesis presents the following results:

- A prototype of a new large scale data store, following the proposed architecture and implementing the novel data placement strategy presented in this dissertation, is presented and evaluated.

- Extensive simulation and real results, under a workload representative of applications currently exploiting the scalability of emerging key-value stores, confirm the properties claimed both by the new large scale data store and by the data placement strategy.

- A prototype of a distributed SQL query engine running on a large scale data store.

- Experimental results, using standard industrial database benchmarks, show that the prototype is both scalable and efficient compared to a standalone version of a SQL engine.

1.4 Dissertation outline

The chapters of this dissertation are organized as follows:

- Chapter 2 presents the relevant related work for this thesis. It starts by describing DHTs and its most popular implementations. After describing DHTs, existing data and replica placement strategies in DHTs are described and compared. Then, a detailed comparison of existing large scale data stores regarding their data model, API, and architecture is presented. Finally, recent work, aiming to ease the transition path from relational databases to large scale data stores, is described and compared. Each section is concluded with a comparison of the solutions described, and an analysis focused on their limitations.

- Chapter 3 presents the novel architecture for large scale data stores; details the design of one of its components (DataDroplets) encompassing existing data placement strategies, and a novel correlation-aware placement strategy for efficient multi-items operations; a detailed description of DataDroplet's prototype; and the last section presents the extensive evaluation results.

- Chapter 4 starts by presenting the challenges for the efficient support of SQL on a large scale data store, and then presents the fundamental design modifications needed to be done to an existing SQL query engine. The chapter also includes the description of the current prototype developed in the context of the CumuloNimbo FP7 project and its experimental evaluation.

- Chapter 5 provides a conclusion and points to future research directions for large scale data stores with additional consistency and enriched APIs, and how to ease the migration from current RDBMS.

Related publications This thesis integrates the work of my PhD research done from 2007 to 2012. During this period, I was involved in a National research project, Stratus (PTDC/EIA-CCO/115570/2009), and two European research projects: CumuloNimbo (FP7-257993) and GORDA (FP6-IST2-004758). Preliminary versions of portions of this dissertation have been published as:

- Ricardo Vilaça, Francisco Cruz, José Pereira and Rui Oliveira, A Simple Approach for Executing SQL on a NoSQL Datastore (submitted), ACM Symposium on Cloud Computing, San Jose, CA, USA, October 14-17, 2012

  This paper presents a simple approach for running SQL queries on a large scale data store, while preserving the underlying flexible schema and transaction-less semantics. The experimental results show it provides good performance and scalability properties.

- Ricardo Vilaça, Rui Carlos Oliveira, and José Pereira. A correlation-aware data placement strategy for key-value stores. In Proceedings of the 11th IFIP WG 6.1 international conference on Distributed Applications and Interoperable Systems, DAIS'11, pages 214–227, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21386-1.

This paper presents a novel data placement strategy, supporting dynamic tags based on multidimensional locality-preserving mappings. Its evaluation under a realistic workload shows that the proposed strategy offers a major improvement on the system's overall response time and network requirements.

- Miguel Matos, Ricardo Vilaça, José Pereira, and Rui Oliveira. An epidemic approach to dependable key-value substrates. In International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV 2011), June 2011

  This position paper outlines the major ideas of a novel architecture designed to handle today's very large scale demand and its inherent dynamism.

- Ricardo Vilaça, Francisco Cruz, and Rui Oliveira. On the expressiveness and trade-offs of large scale tuple stores. In Robert Meersman, Tharam Dillon, and Pilar Herrero, editors, On the Move to Meaningful Internet Systems, OTM 2010, volume 6427 of Lecture Notes in Computer Science, pages 727–744. Springer Berlin / Heidelberg, 2010

  This paper introduces DataDroplets, a novel data store that shifts the current trade-off toward the needs of common business users, providing additional consistency guarantees and higher-level data processing primitives. Preliminary results of the system's performance under a realistic workload are also presented.

- Ricardo Vilaça and Rui Oliveira. Clouder: a flexible large scale decentralized object store: architecture overview. In WDDDM '09: Proceedings of the Third Workshop on Dependable Distributed Data Management, pages 25–28, New York, NY, USA, 2009. ACM

  This position paper presents preliminary ideas for the architecture of a flexible, efficient and dependable fully decentralized large scale data store able to manage very large sets of variable size objects, and to coordinate in place processing.

Chapter 2

Related work

The need for leveraging decentralized infrastructures to store and process large volumes of data was explicitly acknowledged by the research in preceding years. A large body of research has been recently dedicated to seek more flexible solutions based on the foundation of highly decentralized peer-to-peer systems that, to some extent, is feeding most of the ideas and proposals of large scale data stores.

This chapter is organized as follows. Section 2.1 starts by presenting Distributed Hash Tables (DHTs), and then reviewing the most popular DHT implementations that were the basis for the ones used in large scale data stores. Then, in Section 2.2 data placement strategies used in DHTs are reviewed. It starts by describing the single node strategies, and then the replicas placement when replication is used. Section 2.3 presents a detailed comparison of large scale data stores regarding: data model and programming interface and architecture. Then, Section 2.4 describes how query processing is done in traditional RDBMS, and presents some work aiming to ease the transition path from relational databases to large scale data stores. Finally, the most relevant conclusions to this thesis are summarized in Section 2.5.

2.1 Distributed Hash Tables (DHT)

Most distributed systems need data location and routing. Data location and routing deal with the problem of efficiently route the request, a key representing some data, to the closest node holding the information. Additionally, when a large number of nodes exists it must be able to find data, despite the dynamics on the system's membership due to network and node's failures.

Client-server based systems present limitations in an Internet scale distributed environment. In this type of systems, resources are concentrated on a small number of nodes, and sophisticated techniques must be used to provide high availability and dependability.

Peer-to-peer systems emerge as an alternative to traditional client-server systems. In a peer-to-peer system, all nodes play symmetric roles acting as clients and servers simultaneously, and being responsible for part of the information available in the system. Although circumventing many problems of traditional client-server systems, peer-to-peer systems present more complex placement and search methods, node organization and security issues.

Although initial peer-to-peer systems only provide file system-like capabilities, in recent years they have been enriched with storage and processing capabilities over structured data. Particularly, large scale data stores use peer-to-peer systems.

One of the major problems in peer-to-peer systems is to find nodes responsible to answer a given query. Nodes are organized in overlay networks, and depending on nodes maintaining or not a structure with specific topologies and properties to answer queries, peer-to-peer systems can be classified as unstructured or structured. The former do not impose any structure on the overlay network. They build a random graph with nodes, and use flooding or random walks on that graph to discover data stored by overlay nodes. The latter use key-based routing schemes, and assign keys to data items building a graph that maps each key to the node that stores the corresponding data. The most common type of structured P2P network is the Distributed Hash Table (DHT) (Lua et al. 2005). Similar to the functionality provided by hash tables (which store keys to values maps), DHTs are distributed data structures that enable peer-to-peer applications to efficiently store and retrieve data, although having higher maintenance costs. As peer-to-peer systems are highly dynamic, they store the values in several nodes to achieve availability.

Oppositely to traditional peer-to-peer systems, where nodes can leave and join the system freely leading to transient failures, large scale data stores run on stable environments. Additionally, large scale data stores run under rigid time constraints. For this kind of environments, structured peer-to-peer systems based on DHTs are most adequate (Castro et al. 2004). DHTs are a powerful abstraction for the design of new applications and communication models, and allow to guarantee a correct routing answer in a bounded number of hops in dynamic networks.

The first step to address the data location and routing problem in DHTs is to assign objects to the available nodes. One of the most adopted techniques for this purpose is *consistent hashing* (Karger et al. 1997). Each node has a Globally Unique IDentifier (GUID), and data items also have unique identifiers. Moreover, a virtual domain is defined by identifiers positioned in a virtual ring which are unique integers in the range $[0, 2^m-1]$. Then, both node and data identifiers are mapped into the ring virtual domain (using some hash function), and a key K is assigned to the node whose virtual identifier is equal or greater than virtual identifier of item with key K. Such node is known as the *successor(K)*. Figure 2.1 shows a ring with $m = 3$ (eight identifiers). The system has three nodes $N1, N2$ and $N3$, and five data items with keys A, B, C, D and E. Keys A, C and D are allocated to node $N1$; keys B and E are allocated to nodes $N2$ and $N3$, respectively.

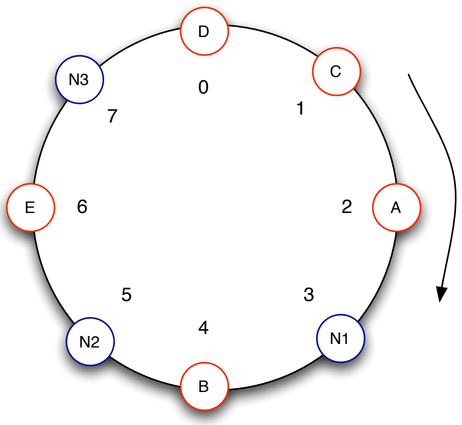


Figure 2.1: Consistent hashing with three nodes $N1, N2$ and $N3$ and five keys A, B, C, D and E

When a node fails its load is reallocated to adjacent nodes; when a node joins the system it splits the load with an existing node. In the example above, if $N2$ fails, data with key B will be reallocated to node $N3$. Since peer-to-peer systems are highly dynamic, this behavior is crucial as the reconfiguration of the system should be as fast as possible.

Subsequently, we present the most popular DHT implementations, which were the basis for the ones used in large scale data stores (DeCandia et al. 2007; Chang et al. 2006; Cooper et al. 2008; Lakshman and Malik 2010).

2.1.1 Chord

Chord (Stoica et al. 2001) is a DHT implementation where node and data are both mapped into a 160-bit identifier space. In Chord, a node in a network of size N only maintains information about $O(logN)$ nodes. The identifier space is the same used in consistent hashing providing a ringlike geometry.

Implementing consistent hashing in a distributed environment only requires each node to know its successor node in the ring. Thus, queries can be answered by passing them around the ring following these successor pointers. As this information implies inefficient routing, Chord maintains additional routing information.

Chord builds a routing table, known as the *finger table*, having at most m (number of bits in the key/node identifiers) entries. The i^{th} entry in the table, at node N, is the node $S = successor(N + 2^{i-1})$. This means that S is the first node succeeding N by at least 2^{i-1} positions in the ring. Each entry in the *finger table* contains the Chord identifier and the IP address of the node. The first entry in the *finger table* of node N is its immediate successor in the ring, and is further referred as *successor*. With this scheme, each node stores information about a small number of nodes and have more information about nearby nodes in the ring than nodes farther away.

As the generated *finger table* does not have enough information to directly route any arbitrary key K, Chord defines how the routing is done in such case. A node N, which has no knowledge of the successor of a key K, $successor(K)$, searches in its *finger table* for the node that immediately precedes K in the ring, and asks that node whose identifier is closest to K. The repetition of this process allows N to learn nodes closer to K.

Chord must be able to locate every key in the network, despite failures. As such, the *successor* of each node must be maintained, and for every key K the $successor(K)$ is responsible for K. Thus, it is able to maintain integrity of the ring despite failures of nodes. In order to simplify the joining and leaving of nodes, Chord also maintains in each node a *predecessor pointer* of the immediate predecessor of that node in the ring.

A new node N joining the system learns the identity of an existing node M by an external mechanism; then uses M to initialize its state and join the existing Chord network. Additionally, Chord needs to update the *fingers* of existing nodes and move the keys to which node N is the new successor.

2.1.2 Tapestry

Tapestry (Zhao et al. 2001) is a DHT implementation that uses the routing mechanisms introduced by Plaxton et al. (Plaxton et al. 1997). They propose a meshlike structure of neighboring maps that can be viewed as a large set of embedded trees in the network.

Those neighboring maps are used to incrementally route messages to the destination digit by digit. [1] A node N has a neighboring map with multiple levels where each level contains an entry per base of the identifier. The i^{th} entry in the j^{th} level (Lj) is the identifier and location of the closest node ending with "i"+suffix(N, j-1).

Figure 2.2 shows an example of Tapestry routing from node 12345 to node $123AE$ with five hops. Briefly, the routing is done as follows: it starts at node 12345 at level $L1$ being forward to node $6893E$ with routing at level $L2$, second hop. Finally, at level $L5$ node $F23AE$ forwards to destination node $123AE$.

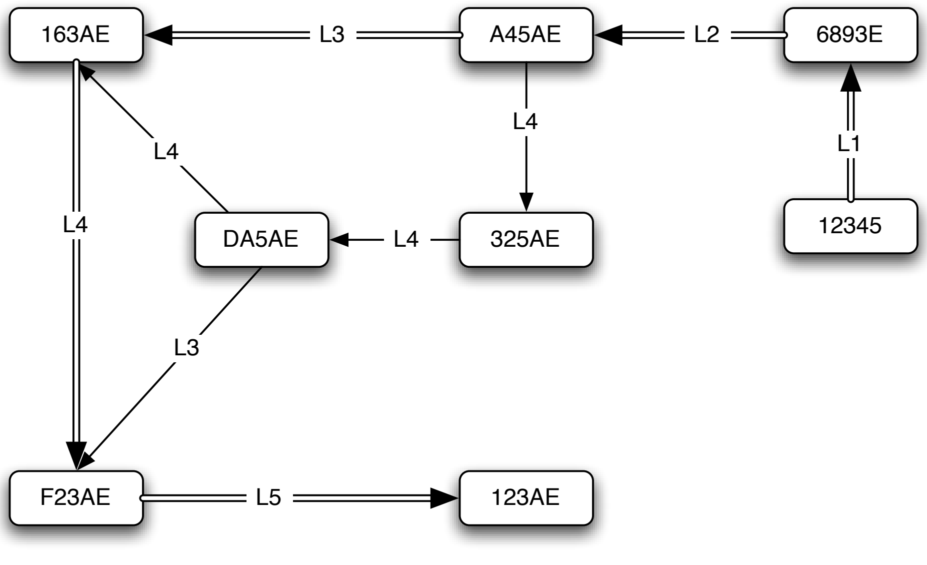


Figure 2.2: Tapestry routing example with hexadecimal bits of length five

This mesh can be used to store structured objects by generating identifiers to objects similar to the ones used by nodes. Thus, each object is mapped to the node having most common bits. Tapestry has been used in OceanStore (Rhea et al. 2003), a distributed storage application.

[1]In further examples, bits are processed from right to left.

2.1.3 Pastry

Pastry (Rowstron and Druschel 2001) is another DHT implementation combining techniques from Chord and Tapestry.

Each node in Pastry maintains three data structures: a routing table, a neighborhood set and a leaf set. The size of those data structures is configured using the following parameters: b, $|M|$, and $|L|$. The value of b involves a trade-off between the size of the routing table and the number of hops between any pair of nodes. Typical values for b, $|M|$, and $|L|$ are 4, 2^b, and 2^b, respectively. They may be configured depending on the target environment and desired behavior.

For routing purposes, Pastry node identifiers are handled as sequences of digits in base 2^b. The routing table has $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries in each row, in order to route messages with a maximum of $\lceil log_{2^b} N \rceil$ steps. The th row contains the identifiers and IP addresses of the closest $2^b - 1$ nodes that share with the current node the first n digits, but differ in the $n + 1$ digit. The distance is measured using a proximity metric. This proximity metric is given by the application and can be any scalar metric, such as IP routing hops or geographic distance.

The neighborhood set, M, contains the $|M|$ closest nodes, according to the proximity metric. It is used for maintaining locality properties. The leaf set, L, contains the $|L|$ numerically closest nodes: the first half are the numerically smaller nodes, and the other half the numerically larger nodes, all relative to the present node. Additionally to the routing table, the leaf set is used during message routing for optimization purposes.

When a node receives a message, it first checks if the key falls into the range of node's identifier covered by its leaf set. If so, the message is forwarded directly to the node in the leaf set, which is closest to the key. Otherwise, the routing table is used and the message is forwarded to the node whose identifier shares with the key a prefix, which is at least one digit longer than the prefix that the key shares with the current node's identifier.

It is possible that the appropriate entry in the routing table is empty or the associated node is not reachable, in which case the message is forwarded to a node that shares a prefix with the key at least as long as the present node's identifier.

Routing in Pastry can be random, meaning that the choice among multiple nodes can be made randomly. Thus, in the event of a malicious or failed node along the path, one can avoid the bad node by repeating the query several times.

2.1.4 One-hop lookup DHTs

Typical DHT implementations present a high lookup cost not appropriate for peer-to-peer applications in which the routing overlay is used to find the nodes responsible for a particular data item, and are then contacted directly as in data stores. Beehive (Ramasubramanian and Sirer 2004) and OneHop (Gupta et al. 2004) appeared as alternatives offering one hop lookup.

Beehive is a general replication framework operating on any ring-based DHT that uses prefix routing: Chord, Pastry, or Tapestry. The key idea behind Beehive is: if the object is replicated at all nodes preceding the node on all query paths, the *predecessor* and the *successor* of the node, on average the number of hops in lookup will be reduced to one. Therefore, Beehive controls the replication in the system by defining a replication level to each object. The replication level may vary from 0 to $log_m N$, 2^m being the ring size. The lookup for an object with replication level $log_m N$ costs one hop.

OneHop is a peer-to-peer routing algorithm where every node maintains a full routing table. The routing table has information about all nodes in the overlay, and the query success rate depends on the accuracy of the routing table. This is determined by the amount of time needed to notify membership changes in the network. Thus, the authors define a hierarchy forming dissemination trees to lower notification delays. The 128-bit circular identifier space, similar to the one used in Chord, is divided in k equal partitions called *slices*. The i^{th} *slice* contains the nodes whose identifiers are contained in interval $[\frac{i.2^{128}}{k}, \frac{(i+1).2^{128}}{k}]$. Each *slice* has a *slice leader*, which is dynamically chosen as the successor of key $\frac{(i+1/2).2^{128}}{k}$. In the same way, each *slice* is divided into *units*; each *unit* has also a *unit leader* determined in the same way as the *slice leader*.

In order to maintain the routing table, each node maintains local information and notifies changes to all nodes. The former is needed so each node knows the interval in the circular space it is responsible for, and the latter is needed to route requests in just one hop.

Local information is maintained by a stabilization process in which nodes send *keep-alive* messages to its *successor* and *predecessor*.

The notification of other nodes about membership changes is done by means of the dissemination tree, Figure 2.3, by the following process:

1. First the node sends a message to its *slice leader*.

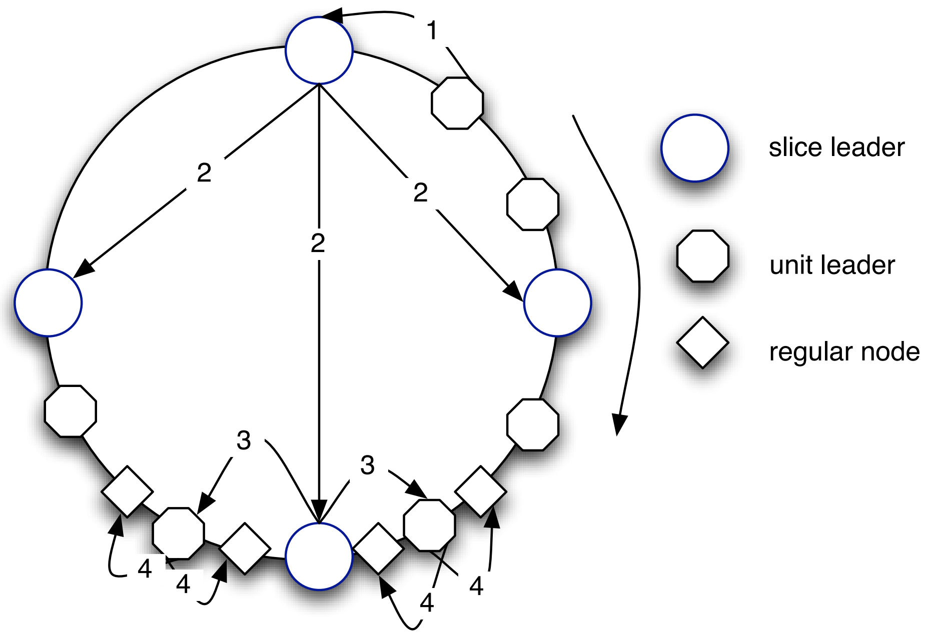


Figure 2.3: OneHop membership notifications

2. The *slice leader* aggregates all messages from its *slice* received in interval t_{big}, and then sends them to other *slice leaders*.

3. The *slice leaders* aggregate all messages received in interval t_{wait}, and then send them to *unit leaders*.

4. *Unit leaders* send this information to their *successors* and *predecessors* in the stabilization process.

5. Other nodes propagate information in the same direction they receive.

2.1.5 Discussion

Chord, Pastry and Tapestry have the same approach for routing, exploring the work by Plaxton et al. (Plaxton et al. 1997). They differ in the approach used to achieve network locality and in replication support.

In Chord, there is no natural correlation between the overlay namespace distance and network distance. On the contrary, Tapestry and Pastry have natural correlation between the overlay topology and the network distance. Furthermore, Chord makes no explicit effort to achieve good network locality.

A large scale data store is a direct contact application in which the routing overlay is used to find the nodes responsible for a particular data item. This type of applications need to maintain information about all nodes to reduce the lookup latency (Cruces et al. 2008). Moreover, the membership is more stable than in traditional peer-to-peer systems, where nodes can leave and join the system arbitrarily.

Typical implementations of DHTs favor a small memory and network overhead over lookup latency; thus, providing a higher cost in routing and searching of data. One-hop protocols presented in Section 2.1.4 question this trade-off and allow a lookup using only one hop. However, they must store the routing information for all nodes on each node, and this requires higher background traffic and appropriate mechanisms to maintain the routing tables up-to-date.

Large scale data stores, such as Dynamo (DeCandia et al. 2007) and Cassandra (Lakshman and Malik 2010), use a one hop DHT for data location and routing in order to avoid multi-hops for data lookup as it would increase variability in response times. While the average latency could not suffer much from these variations, the latency at higher percentiles (needed to satisfy some SLAs) would increase. Each node maintains enough local routing information to route a request to the appropriate node directly.

2.2 Data placement in DHTs

One of the major problems in distributed processing over massive-scale storage is the data placement problem.

The distributed placement of data is a well-known problem in distributed data management (Özsu and Valduriez 1999). It can be defined as a mapping from N items (all current items in the system) to the S available nodes. In an infrastructure with the dimension of a large scale data store (hundreds of nodes), hardware and software failures are common; therefore, nodes may fail or rejoin the system frequently (Schroeder and Gibson 2007). Furthermore, in order to achieve the desired elasticity of cloud computing, the system must be capable of easily add or remove nodes (scale in or out) in a quick and transparent way. This poses new challenges to the data placement problem. Over this environment, the placement must be dynamic to adapt to joining or leaving nodes, while having a minor impact on the system's performance, by minimizing the cost of the data transfer and rebalancing processes as the system scales in and out.

The optimal distribution of items should consider both the cost of the data distribution itself (of storing and moving the data to different nodes), and the performance of future operations that query the stored data. While for operations that add new data to the system, its performance depends directly on the data placement strategy; for future queries it is not clear the cost of the used data placement strategy. The solution to the placement problem is dependent on the applications running in the system and on their workloads. Moreover, in a multi-tenant environment, workloads with different requirements may coexist.

Particularly, by mapping keys to values in a distributed manner, DHTs also need a distributed data placement strategy to assign items to nodes. In ring-based DHTs, the data placement strategy can be decomposed in two components: the single node placement strategy, presented in section 2.2.1, and the replicas placement strategy, presented in section 2.2.2. The former is used to define the partition of data items across the ring, which nodes are responsible for each data item; the latter is used to define which nodes will have the additional replicas needed to achieve resilience.

2.2.1 Single node placement

This section presents and compares the strategies used by DHTs to partition data across all available nodes. For a given data item, the strategy must define which node is responsible for it, allowing every other node to discover the node responsible for that data item.

Random

One of the most used data placement strategies is the random strategy. In the random strategy, the data items' keys are, similar to node identifiers, pseudo-randomly hashed in the domain $[0, 2^m - 1]$.

The major advantage of this strategy lies in uniformly mapping data's identifiers to the identifier space, providing automatic load balancing. However, the hashing does not contemplate data locality, and can make range queries very expensive by retrieving continuous data items from many different nodes, and later sorting them in the proper order.

Ordered

The ordered strategy tries to improve range queries, it requires that items' keys define some partially ordered set, placing items according to this order. In this strategy, an order-preserving hash function (Garg and Gotlieb 1986) is used to generate the identifiers. Compared to a standard hash function, for a given ordering relation among the items, $<$, and items $I1$ and $I2$, an order-preserving hash function *hashorder()* has the extra guarantee that if $I1 < I2$, then $hashorder(I1) < hashorder(I2)$. This strategy preserves data locality, because the hash function preserves the partial order defined over the items.

However, it is useful to have some knowledge on the distribution of the items' keys (Garg and Gotlieb 1986) to make the order-preserving hash function uniform as well. There are many order-preserving hash functions that can be used, however, it is quite improbable that they also provide randomness and uniformity. The straightforward approach to the implementation of an ordered placement is to use the exact relative values of the items' keys, according to the maximum and minimum value of the expected item's key domain. This requires *a priori* knowledge of the expected domain for items' keys. Clearly, this estimate of the domain must be carefully done. While an optimistic estimate of the domain prevents from future reorganization of items, it may lead to a non-uniform distribution of data across nodes. On the contrary, a pessimist estimate of the domain leads to a more uniform distribution of data across nodes, but may trigger a costly reorganization of items.

The ordered strategy limits the optimization to queries with single attribute ranges or imposes a more complex overlay network, such as Mercury (Bharambe et al. 2004) or Chord# (Schutt et al. 2006), where multiple simple overlays, one for each attribute, are mapped onto the same set of nodes.

Space Filling Curves (SFCs)

One approach to avoid the restriction to single attribute queries is to preserve data locality, using locality preserving space filling curves (Sagan 1994). SFCs map multi-dimensional data to one-dimensional keys. While the ordered strategy takes into account a single attribute, data items' keys, this technique can consider multiple items' attributes.

Systems using this type of strategy (Schmidt and Parashar 2003; Ganesan et al. 2004; Chawathe et al. 2005) guarantee that a query is answered with bounded costs in terms of the number of messages and number of nodes involved. They support queries with partial keywords, wildcards and range queries.

These systems use an analogous approach to index data and distribute the index to nodes. However, the query engine is different. Place Lab (Chawathe et al. 2005) performs a lookup for each value in the specified range, which makes its performance suboptimal. The approach used by Ganesan et al. (Ganesan et al. 2004) increases false-positives (for example, non-relevant nodes may receive the query) to reduce the number of sub-queries. Squid (Schmidt and Parashar 2003) decentralizes and distributes the query processing across multiple nodes in the system. Recursive query processing can be viewed as constructing a tree. At each level of the tree the query is restricted to a set of nodes, which are refined at the next level. The tree is embedded into the overlay network: the first node performs the first refinement of the query, and each subsequent node further refines the query and forwards the resulting sub-queries down the tree to proper nodes in the system.

In SFCs, if the query contains partial keywords, wildcards or is a range query, the query identifies a set of points (data items) in the multi-dimensional space that corresponds to a set of points in the one dimensional identifier space.

Discussion

Each presented strategy has its trade-offs. While random strategy provides automatic load-balancing and behaves well for single item lookup queries, they don't preserve data locality; therefore, they have to forward multi item queries to all nodes.

Furthermore, all strategies described above, except for range queries, do not consider general multi-item operations, which are common over data-intensive workloads. A multi-item operation requests database operations (reads and/or writes) to a specific subset of items to accomplish a certain task. As shown by Gibbons et al. (Yu et al. 2006), the availability and performance of multi-item operations is highly affected by the data placement and partition strategies.

The probability for a pair of items to be requested together in a query (known as correlation) is not uniform, but it is often highly biased (Zhong et al. 2008). Additionally, correlation is mostly stable over time for most real applications. Therefore, if the data placement strategy places correlated items on the same node, then the communication overhead for multi-item operations tends to reduce.

To our knowledge, existing DHTs (Stoica et al. 2001; Ratnasamy et al. 2001; Rowstron and Druschel 2001; Zhao et al. 2001; Bharambe et al. 2004; Schutt et al. 2006) use a single strategy for data placement. However, the best strategy is dependent on the applications and their workloads. While some placement strategies can be optimal for some queries, it may have a huge cost for other types of queries. Therefore, each data placement strategy yields different trade-offs.

The use of a single strategy for data placement prevents the data placement to adapt and to be optimized to the clients' workloads, and, in the specific context of cloud computing, to suit the multi-tenant architecture.

2.2.2 Replica's placement

Data must be replicated to achieve data resilience or for performance reasons, such as load balancing of read-only operations. In a replicated environment, for a replication degree of R there are R replicas per item. One of the nodes is chosen using the single node placement strategy, and the others $R - 1$ replicas are chosen using the replica placement strategy. Multiple replica placement schemes can be defined, and they could have a high impact in the availability and performance of the system (Yu et al. 2006; Ktari et al. 2007; Leslie et al. 2006).

All replica placement strategies presented bellow can be applied to all DHT implementations presented above. However, some strategies may need additional state for replica management if applied to a given DHT implementation. Subsequently, we describe replica placement strategies defined and used by several DHTs, and discuss their advantages/disadvantages.

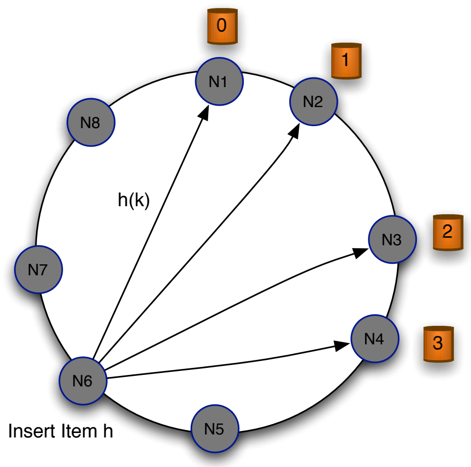
List replication

In list replication, the set of replicas is chosen from a prefix of an already existent list. Depending on the used list, it can be successor-list replication or leaf-set replication.

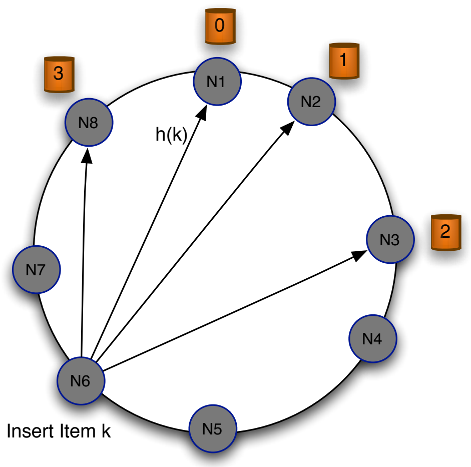
In successor-list replication (Stoica et al. 2001), Figure 2.4(a), the key of each item is hashed in the DHT and assigned to the R closest successors nodes of that key.

The leaf-set replication (Rowstron and Druschel 2001; Maymounkov and Mazières 2002), Figure 2.4(b), is similar to successor-list replication, but gathers both successor and predecessor nodes. Instead of storing an item on its closest R successor's, the item is stored on its $\lfloor \frac{R}{2} \rfloor$ closest successors and its $\lfloor \frac{R}{2} \rfloor$ closest predecessors.

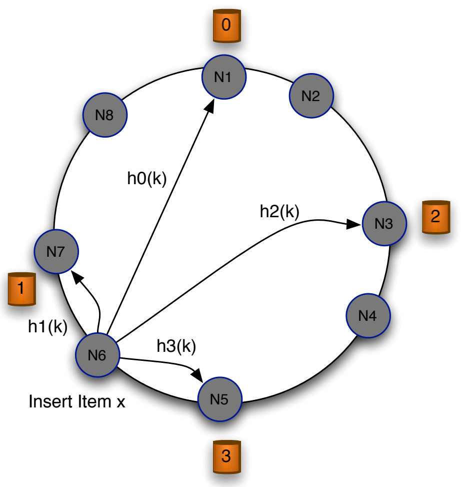
For systems using successor-list replication the routing always proceeds clock-



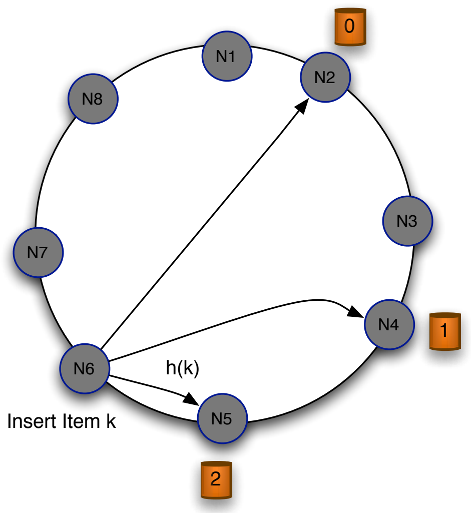
(a) Successor replica placement



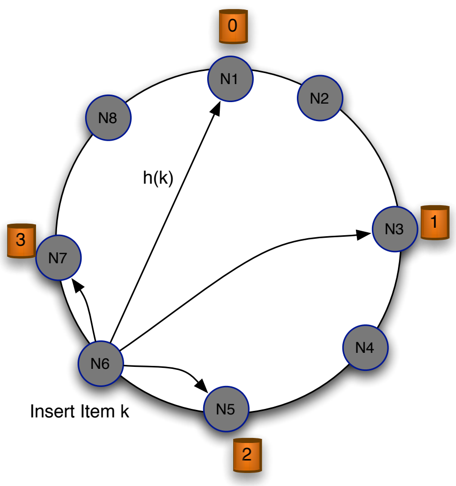
(b) Leaf-Set replica placement



(c) Multi Hashing replica placement



(d) Path replica placement



(e) Symmetric replica placement

Figure 2.4: Replica placement strategies

wise, while in systems using leaf-set replication routing is done in both ways, clockwise and anti-clockwise. However, as in one-hop DHTs each node may directly route messages to all nodes, the two strategies behave the same.

Multi hashing replication

Some DHTs, CAN (Ratnasamy et al. 2001) and Tapestry (Zhao et al. 2001), instead of a single hash function, use several hash functions to define in which nodes to place data, Figure 2.4(c). In such a scheme, R hash functions are used to achieve a replication degree of R. Each item in the DHT gets R identifiers by applying the respective functions to the key value. Therefore, the item is stored at $successor(h_0(k))$, $successor(h_1(k)), \ldots, successor(h_{R-1}(k))$.

A particular scheme of multi-hashing is rehashing, where the same hash is applied to the result of the previous hashing. Therefore, each item is replicated at nodes: $successor(h(k))$, $successor(h(h(k)))$, $successor(h(h(h(k))))$, and so forth.

Path replication

In path replication, used in Tapestry (Zhao et al. 2001), Figure 2.4(d), an item is replicated to nodes along the search path that is traversed during the routing of the message, from the source to the destination node.

However, for one-hop DHTs, where any node can directly route messages to all nodes, this strategy behaves as list replication strategy.

Symmetric replication

In symmetric replication (Ghodsi et al. 2007), Figure 2.4(e), each identifier in the system is associated with R other identifiers. If identifier i is associated with identifier j, then the node responsible for item i should also store the item j. Symmetrically, the node responsible for item j should also store the item i. The identifier space is partitioned into $\frac{N}{R}$ equivalence classes so that identifiers in an equivalence class are all associated with each other. Any such partition will work, but, for simplicity, the congruence classes modulo m is used, where N is the size of the identifier space and $m = \frac{N}{R}$ for R replicas. Therefore, each node replicates the equivalence class of all identifiers it is responsible for.

Discussion

The replica placement strategy has a high impact on both performance and availability (Yu et al. 2006; Leslie et al. 2006; Ktari et al. 2007). Each strategy has its trade-offs and no strategy is the best both in reliability and latency aspects.

While symmetric strategy has higher reliability it also has higher maintenance overhead and additional complexity (Ktari et al. 2007). The simple list replication strategy presents good results, while not requiring additional node state (Ktari et al. 2007). For workloads that must have complete answers (that is do not tolerate missing items), the multi-hashing strategy has higher availability than the symmetric strategy (Yu et al. 2006).

Moreover, the optimal solution is dependent on the workload. However, all existing strategies are biased to some kind of a particular workload; thus, they are not suited for a multi-tenant environment, where workloads with different requirements may coexist.

2.3 Large scale data stores

Web-scale applications need to manage very large volumes of structured data. This has led to the emergence of several elastic and scalable distributed data stores designed by major companies Google, Amazon, Yahoo! and Facebook.

In the following, we briefly present four available data stores. The chosen data stores are the most representative; although several open source projects exist, they are mostly implementations of some of the presented here.

Amazon's Dynamo (DeCandia et al. 2007) is a highly available key-value storage system. It has properties of both databases and DHTs. Although it isn't directly exposed externally as a web service, it is used as a building block of some of the Amazon Web Services[2], such as S3[3].

PNUTS (Cooper et al. 2008) is a massively scalable, hosted data management service that allows multiple applications to concurrently store and query data. Its shared service model allows multiple Yahoo! applications to share the same resources and knowledge. PNUTS is a component of the Yahoo!'s Sherpa, an integrated suite of data services.

[2]`http://aws.amazon.com/`
[3]`http://aws.amazon.com/s3/`

Google's Bigtable (Chang et al. 2006) is a distributed storage system for structured data that was designed to manage massive quantities of data and run across thousands of servers. Besides being used internally at Google for web indexing, Google Earth and Google Finance, it is also used to store Google's App Engine Datastore entities. The Google's App Engine Datastore API[4] defines an API for data management in the Google's App Engine (GAE)[5]. GAE is a toolkit that allows developers to build scalable applications in which the entire software and hardware stack is hosted at Google's own infrastructure, Software as a Service (SaaS).

Cassandra (Lakshman and Malik 2010) is a distributed storage engine initially developed by Facebook to be used at the Facebook social network site, and is now an Apache open source project[6]. It is a highly scalable distributed database that uses most of the ideas of the Dynamo architecture to offer a data model based on Bigtable's data model.

Sections 2.3.1, 2.3.2 present a detailed comparison of existing solutions regarding: data model, and programming interface and architecture. Finally, sub section 2.3.3 presents a comparison of the design trade-offs of each system focusing on their limitations.

2.3.1 Data models and Application Programming Interfaces (API)

The emergence of cloud computing and the demand for scalable distributed data stores are leading to a revolution on data models. A common approach in all recent large scale data stores is the replacement of the relational data model by a more flexible one. The relational data model was designed to store highly and statical structured data.

Most of existing data stores use a simple key-value store or at most variants of the Entity-Attribute-Value (EAV) model (Nadkarni and Brandt 1998). In the EAV, data model entities have a rough correspondence to relational tables, attributes to columns, tuples to rows and values to cells. However, despite each tuple being of a particular entity, it can have its own unique set of associated attributes. This data model allows to dynamically add new attributes that only apply to certain tuples. This flexibility of the EAV data model is helpful in domains where the problem is itself amenable to expansion or change over time. Another benefit of the EAV model that may help in

[4]`http://code.google.com/appengine/docs/datastore/`
[5]`http://code.google.com/appengine/`
[6]`http://cassandra.apache.org/`

the conceptual data design is the multi-value attribute in which each attribute can have more than one value.

In the following, a detailed description of the data model and programming interface for each of the data stores is introduced. Dynamo uses a simple key-value data model, while others use some variant EAV. In order to ease the comparison, for each data store we provide a standard representation of their data model and API. We use the following notation:

a) $A \times B$, product of A and B;

b) $A + B$, union of A or B;

c) A^*, sequence of A;

d) $A \rightharpoonup B$, map of A to B; and

e) $\mathcal{P}A$, power set of A.

Dynamo is modeled as:

$$K \rightharpoonup (V \times C) \ . \tag{2.1}$$

Each value has a key associated to it and a context, represented by C, which encodes system metadata, such as the tuple's version, and is opaque to the application. Dynamo treats both the key and the tuple, K and V, as opaque array of bytes.

```
$\mathcal{P}(V \times C)$ get($K$ key)
put($K$ key, $C$ context,$V$ value)
```

Figure 2.5: Dynamo's API

Dynamo offers a simple interface, Figure 2.5. The `get` operation locates the tuple associated with the `key`, and returns a single tuple or a list of tuples with conflicting versions. The `put` operation adds or updates a tuple also by `key`.

In PNUTS, data is organized into tables of tuples identified by a string, with dynamic typed attributes. Tuples of the same table can have different attributes,

$$String \rightharpoonup (K \rightharpoonup \mathcal{P}(String \times V)) \ .$$

Each tuple can have more than one value for the same attribute. The type for the attributes, V, and for the key, K, can be typical data types, such as integer, string, or the "blob" data type for arbitrary data. The type for the attributes is dynamically defined per attribute.

$\mathcal{P}(String \times V)$ `get-any(`$String$ `tableName,`K `key)`
$\mathcal{P}(String \times V)$ `get-critical(`$String$ `tableName,`K `key,` $Double$ `version)`
$\mathcal{P}(String \times V)$ `get-latest(`$String$ `tableName,`K `key)`
`put(`$String$ `tableName,`K `key,`$\mathcal{P}(String \times V)$ `value)`
`delete(`$String$ `tableName,`K `key)`
`test-and-set-put(`$String$ `tableName,`K `key,`$\mathcal{P}(String \times V)$ `value,` $Double$ `version)`
$K \rightharpoonup \mathcal{P}(String \times V)$ `scan(`$String$ `tableName,`$\mathcal{P}K$ `selections,`$\mathcal{P}String$ `projections)`
$K \rightharpoonup \mathcal{P}(String \times V)$ `rangeScan(`$String$ `tableName,`$(K \times K)$ `rangeSelection,`$\mathcal{P}String$ `projections)`
$String \rightharpoonup (K \rightharpoonup \mathcal{P}(String \times V))$ `multiget(`$\mathcal{P}(String \times K)$ `keys)`

Figure 2.6: PNUTS's API

In PNUTS, as the table's elements can be ordered or randomized, the available operations per table differ, Figure 2.6. All tables support `get-*`, `put`, `delete`, and `scan` operations. However, only ordered tables support selections by range: `rangeScan` operation. While selections can be done by tuple's key, `scan`, or specifying a range, `rangeScan`, updates and deletes must specify the tuple's key. PNUTS supports a whole range of single tuple `get` and `put` operations with different levels of consistency guarantees. Vary from a call where readers can request any version of the tuple, with highly reduced latency, to a call where writers can verify that the tuple is still at the expected version. Briefly, the `get-any` operation returns a possible stale version of the tuple, `get-critical` returns a version of the tuple that is at least as fresh as the `version`, `get-latest` returns the most recent copy of the tuple, and `test-and-set-put` performs the tuple modification if and only if the version of the tuple is the same as the requested `version`. Additionally, a `multiget` is provided to retrieve multiple tuples from one or more tables in parallel.

Bigtable is a multi-dimensional sorted map,

$$K \rightharpoonup (String \rightharpoonup (String \times Long \rightharpoonup V)) \,. \tag{2.2}$$

The index of the map is the row key, column name, and a timestamp. Column keys are grouped into *column families* and they must be created before data can be stored under any column key in that family. Data is maintained in lexicographic order by row key, where each row range is dynamically partitioned. Each cell in BigTable can have multiple versions of the same data indexed by timestamp. The timestamps are integers and can be assigned by Bigtable or by client applications. The type of the row key, K, and the value for columns V, is a string.

```
put($K$ key,$String \rightharpoonup (String \times Long \rightharpoonup V)$ rowMutation)
$String \rightharpoonup (String \times Long \rightharpoonup V)$ get($K$ key,$String \times String$ columns)
delete($K$ key,$String \times String$ columns)
$K \rightharpoonup (String \rightharpoonup (String \times Long \rightharpoonup V))$ scan($K$ startKey,$K$ stopKey,$String \times String$ columns)
```

Figure 2.7: Bigtable's API

The Bigtable API, Figure 2.7, provides operations to write or delete tuples (`put` and `delete`), to find individual tuples (`get`) or iterate over a subset of tuples (`scan`). For all operations, the string representing the column name may be a regular expression. Clients can iterate over multiple column families and limit the rows, columns, and timestamps. The results for both `get` and `scan` operations are grouped per column family.

Cassandra data model is an extension of the Bigtable data model,

$$K \rightharpoonup (String \rightharpoonup (String \rightharpoonup (String \times Long \rightharpoonup V))) \ . \qquad (2.3)$$

It exposes two types of column families: simple and super. Simple column families are the same as column families in Bigtable, and super column families are families of simple column families. Cassandra sorts columns either by time or name. In Cassandra, the type for rows key, K, is also a string with no size restrictions.

```
put($K$ key,$String \rightharpoonup (String \rightharpoonup (String \times Long \rightharpoonup V))$ rowMutation)
$String \rightharpoonup (String \rightharpoonup (String \times Long \rightharpoonup V))$ get(K key,$String \times String \times String$ columns)
$K \rightharpoonup (String \rightharpoonup (String \rightharpoonup (String \times Long \rightharpoonup V)))$ range($K$ startKey,$K$ endKey,
        $String \times String \times String$ columns)
delete($K$ key,$String \times String \times String$ columns)
```

Figure 2.8: Cassandra's API

The Cassandra API, Figure 2.8, is mostly the same of Bigtable's, except for the `scan` operation. The results of `get` are grouped both per super column family and column family, and ordered per column. Additionally, Cassandra's open source project introduced an additional `range` operation in version 0.6.

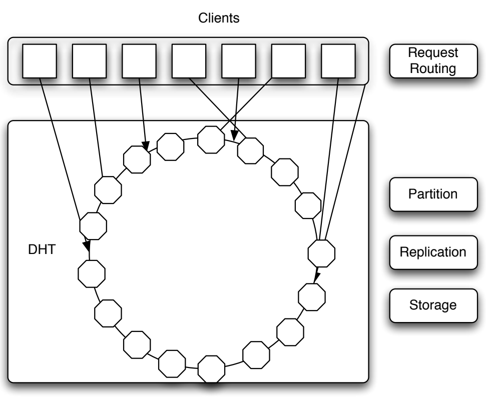
2.3.2 Architecture

Large scale data stores are distributed systems aiming at hundreds or thousands of machines in a multi-tenant scenario, and must be able to store and query massive quantities of structured data. At this scale, machine failures are frequent; therefore, data stores must replicate data to ensure dependability. Enabling distributed processing over this kind of massive-scale storage poses several new challenges: problems of data placement, dependability and distributed processing. Given these challenges and their different design requirements, all the systems under consideration came up with different architectures.

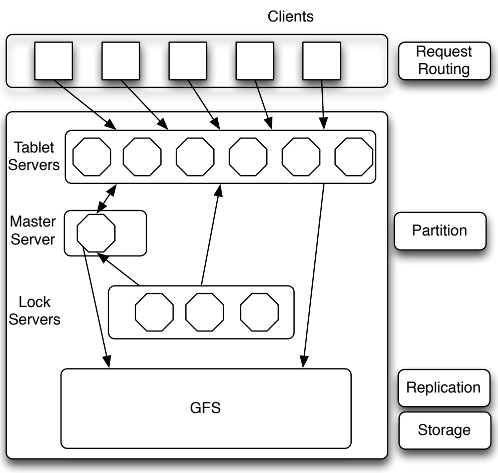
These architectures may be categorized in two types: fully decentralized and hierarchical. In the fully decentralized type, physical nodes are kept organized on a logical ring overlay, such as Chord (Stoica et al. 2001). Each node maintains complete information about the overlay membership therefore, being able to reach every other node. Dynamo and Cassandra fall in this category, Figures 2.9(d) and 2.9(a). In the hierarchical type, a small set of nodes is responsible for maintaining data partitions, and to coordinate processing and storage nodes. Both Bigtable and PNUTS, Figures 2.9(b) and 2.9(c), follow this type; thus, they partition data information into tablets, which are horizontal partitions of tuples. Bigtable is composed of three different types of servers: master, tablets, and lock servers. Master servers coordinate the tablet servers by assigning and mapping tablets to them, and redistributing tasks as needed. The architecture of PNUTS is composed of regions, tablet controllers, routers, and storage units. The system is divided into regions where each region has a complete copy of each table. Within each region, the tablet controller coordinates the interval mapping that maps tablets to storage units.

Despite having different architectures, all of the presented data stores share common components (request routing and storage) and use common techniques (data partitioning and replication). Figure 2.9, shows the architecture of each system and highlights the layers responsible for each component. In the following, we focus on those components pointing out the similarities and differences of their implementation in each data store.

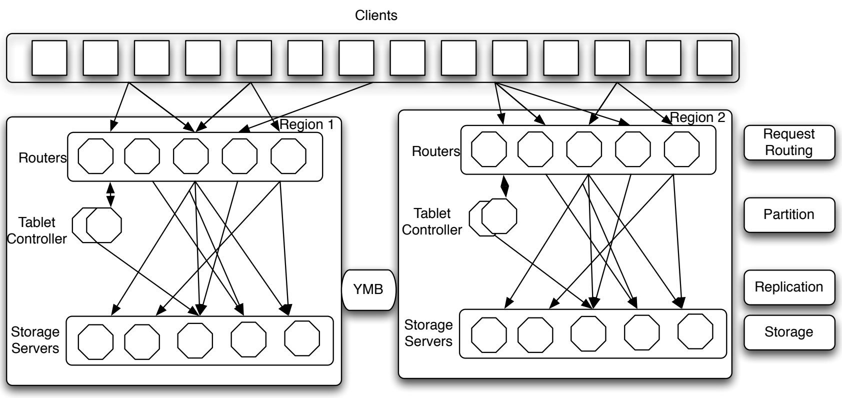
The characterization of the architecture is directly related to the way each system performs data partitioning, an aspect of major importance. While in data stores, using a fully decentralized architecture, the data partition is done in a fully decentralized manner through consistent hashing, in the hierarchical-based architecture a small set of nodes is responsible for maintaining the data partitions. In PNUTS, data tables are partitioned into tablets by dividing the key space in intervals. For ordered tables, the division is at the key-space level, while in hashed tables it is at the hash-space level. Each tablet is stored into a single storage unit within a region. In Bigtable, the row



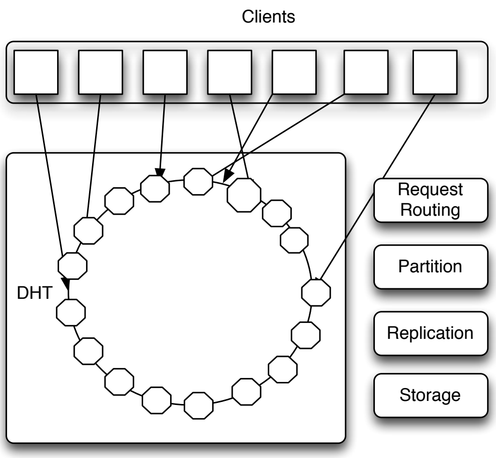
(a) Dynamo



(b) Bigtable



(c) PNUTS



(d) Cassandra

Figure 2.9: Large scale data store architectures

range is dynamically partitioned into tablets distributed over different machines.

Another mandatory aspect of these data stores is replication, which is used not only to improve read operations' performance by means of load balancing, but also to ensure fault tolerance. Cassandra and Dynamo use a successor-list replication strategy. In PNUTS, the message broker ensures inter-region replication, using asynchronous primary backup, while in Bigtable it is done at the storage layer by GFS (Ghemawat et al. 2003).

Besides replication, all data stores use a component to ensure that write operations are made durable, persistent. While Dynamo and Cassandra rely on local disks for persistency, PNUTS and Bigtable use a storage service. However, Bigtable uses the storage service in a transparent manner regarding its location, while PNUTS maintains information about the mapping from tuples to storage nodes, in a per tablet granularity.

Due to the multiple nodes used and the partition of data, every time a request is issued to the data store, it has to be routed to the node responsible for that piece of data. In Cassandra, any incoming request can be issued to any node in the system. Then, the request is properly routed to the responsible node. In PNUTS, when a request is received, the router determines which tablet contains the tuple and which storage node is responsible for that tablet. Routers contain only a cached copy of the interval mapping that is maintained by the tablet controller. In Dynamo, the routing is handled by the client (that is the client library is partition aware), directly sending the request to the proper node. Bigtable also needs a client library that caches tablet locations; therefore, clients send the requests directly to the proper tablet server.

2.3.3 Discussion

Table 2.1: Comparison of data stores

	Dynamo	**PNUTS**	**Bigtable**	**Cassandra**
Data Model	key value, row store	EAV, row store	column store	column store
API	single tuple	single tuple and range	single tuple and range	single tuple and range
Data Partition	random	random and ordered	ordered	ordered
Optimized for	writes	reads	writes	writes
Consistency	eventual	atomic or stale reads	atomic	eventual
Multiple Versions	version	version	timestamp	timestamp
Replication	quorum	async message broker	file system	quorum
Data Center Aware	yes	no	yes	yes
Persistency	local and pluggable	storage service and custom/MySQL	replicated and distributed file system	local and custom
Architecture	decentralized	hierarchical	hierarchical	decentralized
Client Library	yes	no	yes	no

Cloud-based data stores must adapt to multiple tenants with diverse performance, availability and dependability requirements. As stated by the CAP theorem, it is impossible to provide at the same time: network partition tolerance, strong consistency and high availability. However, different combinations of two of these properties are possible; thus, cloud-based data stores must establish reasonable trade-offs.

As described in previous sections, each data store, depending on its internal design requirements, has specific trade-offs. Table 2.1 presents a brief comparison of the analyzed data stores.

Data Model The data model type of a data store, in addition to determine its expressiveness, also impacts how data is physically stored in disks. In a row-based data model (Dynamo and PNUTS), tuples are stored contiguously on disk. While in a column oriented storage (Bigtable and Cassandra) columns may not be stored in a contiguously fashion. For that reason, column-oriented storage is only advantageous if applications only access a subset of columns per request.

API The API of data stores is not only highly coupled with their data model, but also with the supported data partition strategies. Single tuple operations are highly related to the data model, but the availability of a range operation is dependent on the existence of an ordered data partition strategy. Therefore, Dynamo doesn't offer a range operation like the other proposals. Still regarding the API, none of the presented data stores distinguish between inserts and updates, and do not consider general multi-item operations which are common over data-intensive workloads. The `put` operation stores the tuple with a given unique key, and if a tuple with the same key already exists in the system, it gets overwritten.

Reads vs Writes Another important trade-off is the optimization either for read or write operations. A key aspect for this is how data is persistently stored. In write optimized storage (for example Bigtable and Cassandra) records on disk are never overwritten, and multiple updates to the same tuple may be stored in different parts of the disk. Therefore, writes are sequential and thus fast, while a read is slower, because it may need multiple I/O operations to retrieve and combine several updates. Dynamo is also optimized for writes, because the conflict resolution (as further explained) is done in reads, ensuring that writes are never rejected.

Consistency All data stores only offer single tuple consistency operations. However, they differ from each other in the consistency given per tuple. The system with weaker consistency guarantees is Dynamo. It exposes data consistency and reconciliation logic issues to the application developers through conflict resolution methods, which leads to a more complex application logic. Moreover, the application must tune the number of tuple replicas `N`, read quorum `R` and write quorum `W`. Therefore, stale data can be read and conflicts may occur, which must be handled by the application. The conflict resolution can be syntactic or semantic based on the business logic. As multiple versions of the same data can coexist, the update of a tuple in Dynamo explicitly specifies which version of that tuple is being updated.

PNUTS also chooses to sacrifice consistency. PNUTS offers methods with different levels of consistency guarantees. Thus, final consistency guarantees depend on the specific calls made to the system. Although every reader will always see some consistent version of a tuple, it may be outdated. Therefore, the burden of strong consistency is left to the application that must take into account the item's version, and use it to ensure consistent updates and avoid stale reads.

In Bigtable, every read or write of a single tuple is atomic. However, every update on a given column of the tuple specifies a timestamp; therefore, creates a new version. Cassandra's consistency is similar to Dynamo with the value's timestamp defining its version.

Replication Regarding replication, in order to tolerate full data center outages, data stores must replicate tuples across data centers. While Bigtable isn't data center aware, Dynamo, Cassandra and PNUTS are. Dynamo is configured such that each tuple is replicated across multiple data centers. PNUTS's architecture was clearly designed as a geographically distributed service, where each data center forms a region and a message broker provides replication across regions. Cassandra also supports a replication strategy that is data center aware, and allows to ensure that each data item is replicated in different data centers, using Zookeeper.

Common Trade-Offs All existing large scale data stores provide two major trade-offs: no table joins, and single tuple consistency. The reason all share this first trade-off is that making a relational join over data, which is spread across many nodes, is difficult, because it may imply that every node would have to pull data from all nodes for each tuple. However, most applications still need joins, and they must be done manually.

Regarding the second trade-off, offering full database consistency through global transactions would restrict scalability, because nodes would have to achieve global agreement.

Almost all existing large scale data stores focus on applications that have minor consistency requirements and choose network partition tolerance and high availability properties of the CAP triangle. They are in strong contrast with traditional relational databases forfeiting complex relational and processing facilities, and most strikingly, transactional guarantees common in traditional RDBMS.

However, a midway between the two opposites (large scale data stores and traditional RDBMS) is possible. There's no reason why a scalable and replicated database cannot be used with SQL. Recent work on offering more complex processing on large scale data stores is reviewed in the next section.

2.4 Query processing

DBMS are a powerful tool to create and manage large amounts of data in an efficient manner, while safely persisting it. DBMS derive from a a large body of knowledge and technology developed over several decades (Stonebraker and Hellerstein 1998).

DBMS changed significantly after the seminal paper by Codd (Codd 1970), where he proposed relational databases. In relational databases, data is organized as tables called relations, and programmers are not concerned with the storage structure, but instead express queries in a high-level language. The most important query language based on the relational model is the Structured Query Language (SQL).

Relational Database Management Systems (RDBMS) are complex software systems with several components. Figure 2.10 summarizes the main components of a relational database management system. Query processing in a relational and transactional database processing system can be regarded as two logical processing stages, making use of different functional blocks: query and storage engines.

Briefly, the query engine is responsible for compiling, planning, and optimizing each query, thus, obtaining a physical plan ready for execution. The plan is prepared from a set of available physical operators, including both generic implementations of

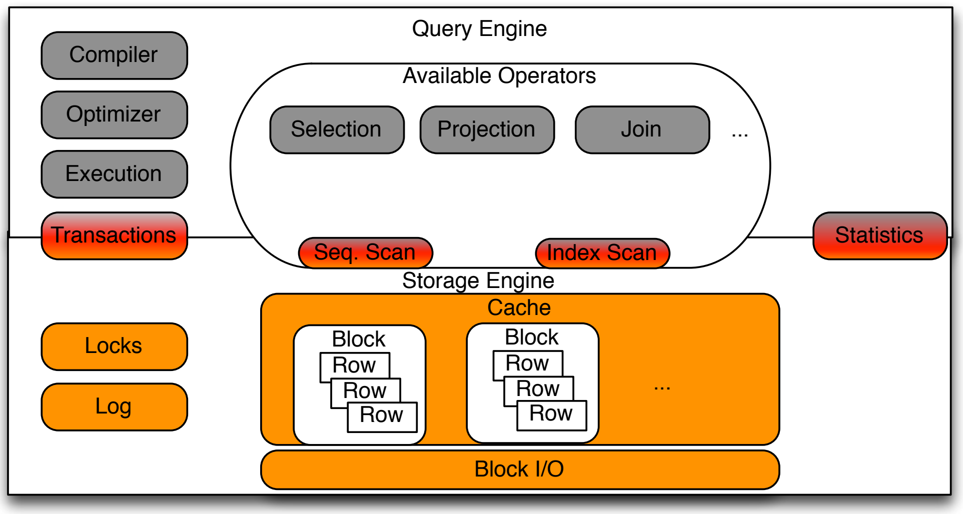


Figure 2.10: Query engine architecture.

relational operators and scan operators provided by the storage manager. The storage engine is responsible for providing scan operators that encapsulate physical representation of data and indexing. Moreover, I/O and caching, recovery and isolation are also responsibilities of the storage engine. In traditional RDBMS, providing such scans efficiently makes use of a cache that keeps a portion of the data stored on disk, and optimizes the usage of block I/O. By implicitly using locks or keeping multiple versions of data items, the storage engine enforces the isolation criteria. By using a persistent log, it also enforces transactional atomicity and durability.

The query engine offers a relational SQL based API for applications. A query engine instance comprises two main stages: compilation and execution of the query. The compilation stage is divided into three phases:

1. Starting from an initial SQL query, the query is parsed, and a parse tree for the query is generated;

2. The parse tree is converted to an initial implementation plan, represented by an algebraic expression, which is then optimized using algebraic laws - the logical query plan;

3. The physical query plan is then constructed from the logical query plan, which implies the choice of the physical operators (that is, algorithms) to be used and

the order in which they must be executed based on the metadata statistics, provided by the storage layer in order to select an efficient execution plan.

In the second stage, the physical plan with the expected lowest cost is executed. However, it should be noted that at phase three of the compilation stage the algorithms for each relational operator are chosen; although the execution of a query is as previously determined by the physical query plan, the execution stage is responsible for the actual implementation of algorithms that manipulate the data of the database.

The interface exposed to the query engine by the storage engine, framed in red in Figure 2.10, can be summarized as follows:

- A set of scan operators and row abstractions, encapsulating all details regarding physical data organization, I/O, caching, versioning and locking. In particular, the row abstraction must allow pinning and updating.
- Meta-information, specifically, logical data layout and low-level statistics.
- Primitives for transaction demarcation, namely,the setup of a transactional context and termination.

Traditional relational database management systems are based on highly centralized, rigid architectures that fail to cope with the increasing demand for scalability and dependability, and are not cost-effective. High performance RDBMS invariably rely on mainframe architectures or clustering based on a centralized shared storage infrastructure. These, although easy to setup and deploy, often require large investments upfront and present severe scalability limitations. Even distributed and parallel databases that have been around for decades (Özsu and Valduriez 1999), build on the same architecture and focus on the problem of guaranteeing strong consistency to replicated and distributed data. Much of the work has focused on distributed query processing (Kossmann 2000) and distributed transactions. The recent trend has been to develop techniques to ease the transition path from relational databases to large scale data stores.

2.4.1 Rich processing in large scale data stores

Recently, there have been some projects that aim to ease the transition path from relational databases to large scale data stores. On one hand, some of those projects intend to mitigate the constraint of simplicity and heterogeneity of the query interface by providing an interface based on SQL, namely BigQuery[7], Hive (Thusoo et al. 2010) and PIQL (Armbrust et al. 2010).

BigQuery is a Google web service built on BigTable. It aims at providing a means to query large datasets in a scalable, interactive and simple way. It uses a non-standard SQL dialect; therefore, it only offers a subset of operators, such as selection, projection, aggregation and ordering. Joins and other more complex operators are not supported. In addition, once data is uploaded it cannot be changed (only more data can be appended), so it does not support update statements.

The Hive system was initially developed at Facebook and is now an Apache project[8]. Hive is built on Hadoop, a project that encompasses the HDFS and the MapReduce framework. Similarly, Hive also defines a simple SQL-like query language to query data. In contrast, it offers more complex operators, such as equi-joins, which are converted into map/reduce jobs, and unions. However, like BigQuery, Hive is aimed at data analysis (OLAP) of large datasets, that is, asynchronous processing.

PIQL allows more complex queries on large scale data stores, while maintaining predictable performance on those complex operations. Providing strict bounds on the number of I/O operations that will be performed by some complex operations is useful, but applications must be rewritten using the new language. PIQL also restricts the set of available operations and aggregate functions, and joins (only equi-joins are supported).

On the other hand, there are a few projects that aim at offering features typical from RDBMS, such as complex operations and secondary indexing, as well as transactional guarantees (ACID) over a large scale data store, specifically: CloudTPS (Wei et al. 2011) and MegaStore (Baker et al. 2011).

CloudTPS chooses to provide strong consistency, but may become unavailable in case of network failures. It offers full transactional guarantees over any NoSQL database, and in order to do so it introduces Local Transaction Managers (LTM). According to the load in the system, there may be several instances of LTM each holding a copy of a partition of data stored in the NoSQL data store. Transactions across several LTM are implemented using a Two-Phase Commit protocol (2PC). In addition to the simple operations provided by a NoSQL datastore, CloudTPS also supports join operations, but with several restrictions on the join and transaction types. Only a subset

[7]`http://code.google.com/intl/pt-PT/apis/bigquery/`
[8]`http://hive.apache.org/`

of join queries are supported, and they are restricted to read-only transactions.

MegaStore approach is very similar to CloudTPS. It is built on BigTable and implements some of the features of RDBMS, such as secondary indexing. Nonetheless, join operations must be implemented on the application side. Therefore, applications must be written specifically for MegaStore, using its data model, and queries are restricted to scans and lookups. Differing from CloudTPS, data is only consistent if in the same partition.

2.4.2 Discussion

Current work on offering some of the traditional database features on large scale data stores aims to ease the transition path from relational databases to large scale data stores. However, they only offer a subset of the processing facilities offered by SQL and define new query languages similar to SQL. This implies that applications must be written specifically for them using their data model, and some queries must be written in application code. This is due to the lack of support for complex processing as general joins are not offered by those systems.

However, most Web-scale applications, such as Facebook, MySpace, and Twitter, remain SQL-based for their core data management, and without full SQL support, it is hard to provide a smooth migration from RDBMS to large scale data stores. This is a hurdle to the adoption of large scale data stores by a wider potential market.

While traditional RDBMS architectures include several legacy components that are not suited to modern hardware, and impose an important overhead to transaction processing (Harizopoulos et al. 2008), limiting both performance and scalability, some RDBMS components can be reused when using a large scale data store.

While the storage engine must be replaced, most of the query engine components can be reused. Reusing the query engine from an existing relational database management system requires that a set of abstractions (scan operators and row abstractions; meta-information, specifically, logical data layout and low-level statistics; and primitives for transaction demarcation) are provided, that the corresponding interfaces are identified and re-implemented.

2.5 Summary

DHTs are a building block for large scale data stores, and they must define data placement strategies to define how data items are assigned to nodes: single node strategies and replica placement strategy.

Regarding the former, existing single node data placement strategies are: random, ordered and SFC-based. Each presented strategy has its trade-offs and none consider general multi-item operations, which are common over data intensive workloads. Moreover, to our knowledge, existing DHTs use a single strategy for data placement. This prevents the data placement to adapt and to be optimized to the clients' workloads, and, in the specific context of cloud computing, to suit the multi-tenant architecture.

The replica placement strategy has a higher impact on both performance and availability (Yu et al. 2006; Leslie et al. 2006; Ktari et al. 2007). Each strategy has its trade-offs and no strategy is better both in reliability and latency aspects. The best solution depends on the workload. However, all existing strategies are biased to some kind of workload, and are not suited for a multi-tenant environment, where workloads with different requirements may coexist.

Cloud-based large scale data stores must adapt to multiple tenants with diverse performance, availability and dependability requirements. However, in face of the impossibility stated by the CAP theorem, Cloud-based data stores must establish reasonable trade-offs. Almost all existing large scale data stores focus on applications that have minor consistency requirements and choose network partition tolerance and high availability properties of the CAP triangle. They are in strong contrast with traditional relational databases forfeiting complex relational and processing facilities, and most strikingly, transactional guarantees common in traditional RDBMS.

However, a midway between the two opposites (large scale data stores and traditional RDBMS) is possible. Recent work on offering some of the traditional database features on large scale data stores focuses on a subset of the processing facilities offered by RDBMS, and defines new query languages similar to SQL. The lack of full SQL support is a hurdle to the adoption of large scale data stores by a wider potential market.

Typical RDBMS components can be reused when using a large scale data store. While the storage engine must be replaced, most of the query engine components can be reused to provide full SQL support on large scale data stores.

Chapter 3

DataDroplets

The management of distributed objects on a large peer-to-peer network raises many and interesting challenges. Some are inherent to the large scale of the system, which is not forgiving of centralized algorithm or flooding protocols, and require judicious control of recurring, system wide, *a priori* negligible tasks. Others are due to the dynamics of the system's membership and each node current capabilities.

Existing approaches to large scale data stores lie at the edges of the strong consistency versus scalability trade-off. The goal of this thesis is to extend the currently disjoint design space by enabling large scale data stores to scale, while offering a well-defined consistency model. Moreover, we expect to find the right trade-offs among flexible usage, efficiency, fault tolerance and quality of service.

In this chapter, we present DataDroplets a key-value store targeted at supporting very large volumes of data, leveraging the individual processing and storage capabilities of a large number of well connected computers. First, Section 3.1 presents Clouder, a novel architecture for large scale data stores, where DataDroplets plays a major role. Then, Section 3.2 details the design of DataDroplets encompassing the supported data placement strategies, and a novel correlation-aware placement strategy for efficient multi-item operations. Section 3.3 describes DataDroplet's prototype; finally, Section 3.4 presents an extensive evaluation of DataDroplets.

3.1 Clouder

We seek to devise a flexible, efficient and dependable large scale data store able to manage very large sets of variable size objects, and to coordinate in place processing.

Our target is local area large computing facilities composed of tens of thousands of nodes under the same administrative domain. The service should be aligned with the emerging Cloud Computing conceptual model. Nodes are not necessarily dedicated, but can be shared by enterprise or academic common tasks. We intend to design a large scale data store that seamlessly fragments and replicates application objects. The system should be capable of leveraging replication to balance read scalability and fault tolerance.

3.1.1 Assumptions

Most medium to big enterprises and universities have large computing facilities composed of tens of thousands of desktop machines that are used to support their collaborators. These machines are underused and have available resources, such as storage, memory, network bandwidth and processing power. Furthermore, these machines are interconnected by a high-bandwidth network, and as they are under the same administrative domain they are trustworthy. We think that these machines are a valuable resource, which can be used to provide a distributed data store thus leveraging machines' capabilities.

The target scenario of Clouder ranges from the typical data center to organizations with a large existing computing infrastructure, such as universities and global enterprises, where the common denominator is large scale and high dynamism. It should be able to leverage large computing facilities composed of tens of thousands of nodes under the same administrative, or at least, ownership domain. Unlike current deployments (DeCandia et al. 2007), some of these nodes are not expected to be functionally or permanently dedicated to the data store, but can be commodity machines mainly dedicated to common enterprise or academic tasks.

At the same time, the boom of the Web 2.0 led to the emergence of new Web applications or services, such as social network applications that need to store and process large amounts of data. These applications require flexible data stores that easily evolve and scale to their growing demands. Although these applications can have lower consistency requirements, most of them cannot afford to deal with conflicts that arise from concurrent updates. Those applications require richer client APIs than existing data stores, which lead to data stores with in-place processing.

To achieve high availability, all the existing data stores rely on high levels of service-dedicated infrastructures, which make them unsuitable to leverage other very large, but less reliable, networks with underused storage and processing resources, such as those found in banks and universities.

Oppositely, Clouder is designed to leverage these large computing facilities composed of tens of thousands of nodes sharing a reliable high-bandwidth network under the same administrative domain. As nodes are interconnected by a high-bandwidth network, bandwidth consumption is not a concern. However, we expect a controlled churn rate typical of an institutional, well administered, computing network. Furthermore, we assume the machines have available resources (storage, memory, network bandwidth and CPU). Applications and users of Clouder machines are trustworthy, rather than malicious.

Nonetheless, we also assume a smaller set of nodes, from tens to hundreds that are in a controlled environment, representing an upfront investment; therefore, are more reliable than the nodes in the large computing facilities. These machines are interconnected by a more reliable network infrastructure that has small latency and little variance over time. This set of nodes is the target environment for DataDroplets, as further explained.

3.1.2 Architecture

The proposed architecture for Clouder is driven by a careful analysis of client requirements and expectations that could be broadly divided in two major domains: i) client interface and concurrency control, and ii) data storage itself and low level processing. Most strikingly, this is the approach taken by traditional RDBMS and sidestepped by new data store proposals. As a matter of fact, in RDBMS the client has access to a clearly defined interface that allows the specification of details, such as isolation guarantees and primitives for performance tuning such as indexes, but hides the details of how data is actually stored, maintained and processed (data independence). This is in sharp contrast with popular data processing approaches, such as MapReduce (Dean and Ghemawat 2008).

To manage very large distributed sets of data, two main approaches prevail: structured network overlays (Ratnasamy et al. 2001; Stoica et al. 2001; Rowstron and Druschel 2001), in which all nodes are logically organized in a well-known structure; and unstructured network overlays (Pereira et al. 2003; Jelasity and Babaoglu 2005; Carvalho et al. 2007), in which nodes are not (*a priori*) structured but are probabilistically managed. The structured approach heavily depends on the attributes and adjustment of the logical overlay to the underlying physical network. A correct overlay contributes to an efficient communication, leveraging nodes and links with higher capacity, locality, and so forth. Furthermore, as the overlay provides approximate views of the whole system, it is possible to attain reasonable guarantees of message delivery and ordering. On the other hand, the usability of a structured overlay is limited by the dynamics of the system's membership. Changes on the set of participants lead to recalculating the overlay which, in the presence of high churn rates, can be impractical. On the contrary, an unstructured network does not rely on any *a priori* global structure, but on basic gossip communication. Each node uses local information about its neighborhood, and relays information to a random subset of its neighbors. These protocols can ensure a high probability of message delivery. Due to their unstructured properties, gossip communication is naturally resilient to churn and faults and simple to maintain.

Clouder adopts both approaches for the management of two collaborating layers, a soft and a persistent state subsystem of distinct structural and functional characteristics, Figure 3.1.

On top, a soft-state layer is responsible for handling client requests and avoiding issues, such as conflicts that eventually arise due to concurrent accesses. These problems require a careful ordering of requests; thus, coordination among participating nodes. This is best achieved by a structured DHT-based approach where nodes partition the key-space among themselves, in order to achieve load-balancing and unequivocal responsibility for partitions (Stoica et al. 2001). With potentially conflicting requests ordered, the remaining procedure of performing the actual read or write over the data is delegated to the persistent-state layer below. As the soft-state layer only carries simple and lightweight operations over metadata, which can be maintained in memory, we expect it to be moderately sized, up to hundreds of nodes; thus, manageable with a structured approach. In fact, on the event of a catastrophic failure or when a new node joins this layer, metadata can be reconstructed from the data reliably stored at the underlying persistent-state layer.

A more efficient design could take advantage of spare capacity at the soft-state layer to serve as a tuple cache and take advantage of relations among tuples to improve the performance of the underlying layer, for instance by adopting workload-aware data placement strategies.

This chapter focuses on the major problems of the soft-state layer while the work on the underlying persistent-state layer is left for future work. Therefore, Section 3.1.3

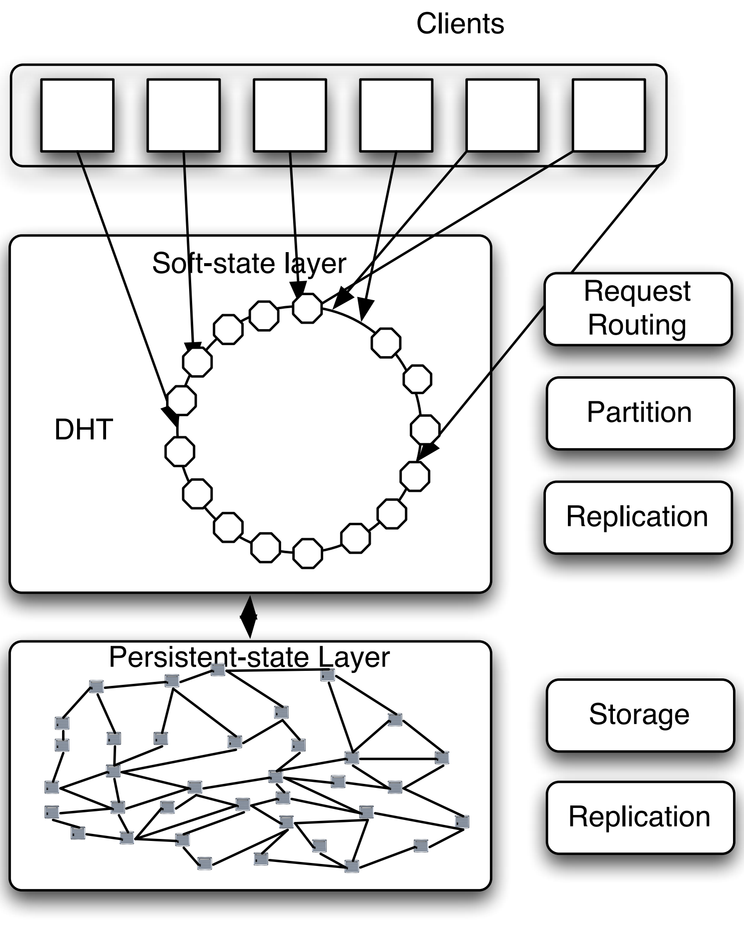


Figure 3.1: Clouder architecture

describes the main design ideas for the persistent-state layer and the remaining sections of this chapter focus on the soft-state layer.

3.1.3 Epidemic-based persistent-state layer

In this section, the main design ideas to the epidemic-based persistent-state layer are described. More detailed design ideas for the persistent-state layer can be found in a position paper (Matos et al. 2011).

The essential properties of a low-level storage system are: i) data availability and ii) performance. Without the first, the system is unusable; the second makes it desirable. The initial approach for the persistent-state layer assumes simple read and write operations. They are ordered and identified with a request version. Any node in the system may receive requests to perform such operations.

Data availability is mainly affected by failures and churn, as data present in offline nodes becomes inaccessible. The only way to cope with unavailability, due to offline nodes is by redundancy. Thus, the main concern is how to achieve and maintain adequate redundancy levels. Despite the redundancy strategy used this directs us to the question: how many nodes need to replicate an item.

Upon a write request, it is necessary to ensure that several copies of the item are spread throughout the system for durability. Due to the low capacity of individual nodes and high churn rates, it is impossible to track down the state of individual nodes, and make globally informed data placement decisions. The strategy is instead based on a global dissemination/local decision approach. The key idea is to spread data in an epidemic fashion (Eugster et al. 2003; Pereira et al. 2003; Birman et al. 1999), and have nodes locally decide if they need to store that data.

Restricted by the impossibility to store all data in a single node and by the required redundancy level, each node needs to locally decide if it will keep the data. The idea is to address this by means of local sieves that should retain only small fractions of data. Thus, upon reception of a new message, nodes locally decide if the message falls into the sieve range and relay it to fanout neighbors. This is in fact similar to what is done in structured DHT approaches, where each node is responsible for a given portion of the key space (Stoica et al. 2001; Rowstron and Druschel 2001).

The sieve function can be computed locally in a random fashion or can take into account some similarity metric, either computed by the node itself or as hinted by the soft-state layer. The only correctness requirement is that all possibilities in the key space are covered in order to avoid data-loss. This also gives enough flexibility to cope with nodes with disparate storage capabilities, as it is only a matter of adjusting the sieve grain in order to impact the amount of stored data. A simple sieve function could simply store an item locally with probability given by $1/number\ of\ nodes$. The number of nodes could be also estimated in an epidemic manner as in Cardoso et al. (Cardoso et al. 2009). Using replication, the sieve function could be simply extended to take into account the replication degree, r, as $r/number\ of\ nodes$.

The other issue to address is the maintenance of the redundancy levels according to the redundancy strategy chosen, which should ensure that a minimal number of copies of every item exists in the system. Again due to scale and churn, a centralized deterministic approach is infeasible; thus, we must rely on probabilistic epidemic-based methods. Those methods, based on random walks (Gkantsidis et al. 2006; Massoulié et al. 2006), allow each node to obtain a uniform sample of the data stored at other nodes, and eventually determine how many copies of the items it holds exist in the system.

Whereas data availability is mainly concerned with *how many* nodes need to hold replicas of a given item, data performance is related to *where* those items actually are. There are several strategies to improve performance in such a system, here we consider the following: i) collocation of related items, and ii) ordering of items.

The most straightforward approach to item co-location is by using smarter sieve functions that are able to take advantage of tuple correlation, instead of blindingly keep items based on a key; thus, co-locate related items.

The other essential issue to improve performance is the ordering of items, which would enable efficient range scans of items and the construction of advanced abstractions, such as indexes; an essential performance feature of traditional relational database management systems that is still lacking on new generation data stores. This problem becomes more evident when data needs to be drawn from several nodes, which is, at first, beneficial due to the ability to perform parallel reads, but raises several interesting challenges as node organization is essential to obtain adequate performance.

To attain such ordering, as nodes cannot store all the data locally, the natural approach is to order nodes such that each node knows the *next* node from which data needs to be retrieved/processed. In an overlay network, this reduces to establishing the appropriate neighbor relations among nodes taking into account the values they store. This semantic organization of nodes clearly calls to an approach based on content-based systems, which are precisely suited to arrange nodes taking into account the values of items they hold (Tran and Pham 2009; Chand and Felber 2005).

The persistent-state layer of Clouder can be exploited to offer simple summaries, such as counts or maximums. Interestingly, it is straightforward to offer simple aggregations to clients with minimal overhead. In fact, basic distributed computations are already done in order to estimate the data distribution of a given parameter; thus, it is simply a matter of exposing such results to the soft-state layer. This is attractive as those computations are likely to be required for attributes where an index or range scan is already built; thus, can be obtained at no cost.

3.2 DataDroplets

DataDroplets is a key-value store targeted at supporting very large volumes of data, leveraging the individual processing and storage capabilities of a large number of well connected computers. It offers a simple application interface providing the atomic manipulation of key-value items, and the establishment of arbitrary relations among items.

In existing proposals, contrary to traditional RDBMS, there is no standard API to query data in data stores. Most of existing Cloud based data stores offer a simple data store interface that allows applications to insert, query and remove individual tuples or at most range queries based on the primary key of the tuple. Regardless of using a simple key-value interface or flavors of the EAV model (Nadkarni and Brandt 1998), thus disclosing more details on the structure of the tuple, current systems require more *ad hoc* and complex multi-tuple queries to be done outside of the data store, using some implementation of the Map Reduce (Dean and Ghemawat 2008) programming model: Yahoo's PigLatin (Olston et al. 2008), Google's Sawzall (Pike et al. 2005), or Microsoft's LINQ (Meijer 2011).

Although this unveils the possibilities of what can be done with data, it has negative implications in terms of ease of use, and in the migration from current RDBMS-based applications. Even worse, if the data store API does not have enough operations to efficiently retrieve multiple tuples for the *ad hoc* queries they will incur in a high performance hit. These *ad hoc* queries will mostly access a set of correlated tuples. Zhonk et al. have shown that the probability of a pair of tuples being requested together in a query is not uniform, but often highly skewed (Zhong et al. 2008). They have also shown that correlation is mostly stable over time for real applications. Furthermore, it is known that when involving multiple tuples in a request to a distributed data store, it is desirable to restrict the number of nodes which actually participate in the request. Therefore, it is more beneficial to couple related tuples tightly, and unrelated tuples loosely, so that the most common tuples to be queried by a request would be those that are already tightly coupled. An important aspect of DataDroplets is the multi-tuple access that allows to store and retrieve large sets of related data efficiently at once.

3.2.1 Data model

Multi-tuple operations leverage disclosed data relations to manipulate sets of comparable or arbitrarily related elements. Therefore, the DataDroplets data model supports tags that allow to establish arbitrary relations among tuples,

$$String \rightharpoonup (K \rightharpoonup (V \times \mathcal{P}String)) \ . \tag{3.1}$$

In DataDroplets, data is organized into disjoint *collections* of tuples identified by a string. Each tuple is a triple consisting of a unique key drawn from a partially ordered set (K), a value that is opaque to DataDroplets (V), and a set of free-form string tags. It is worth mentioning that the establishment of arbitrary relations among tuples can be done even if they are from different collections.

DataDroplets use tags to allow applications to dynamically establish arbitrary relations among items. The major advantage of tags is that they are free form strings; thus, applications may use them in different manners.

Developers of applications based on traditional RDBMS are used to a system of primary keys, foreign keys and constraints between the two. In DataDroplets the item's key K and the set of tags $\mathcal{P}String$ are used as an index for the collection, so when defining them, the developer must consider how the data will be queried.

Moreover, the developer must consider more than querying when defining the set of tags, because it is also used for physically partitioning the tables, which provides load balancing and scalability. Depending on the data placement strategy, the rows with the same set of tags can be kept together in the same node.

In a relational database, we rely on foreign keys and constraints to define relationships. This impacts how queries and updates (including inserts and deletes) from collections are performed. In DataDroplets, foreign keys can be used as tags and the items from different collections will be be correlated, and co-located. For example, an application migrated from a relational database with two tables, *contact* and *address* and a foreign key *address* in *contact*, referencing an existing *address*. In DataDroplets, the items from the collection *contact* should add a tag with its address attribute, and items from the collection *address* should also add a tag with its primary key, the address.

Similarly, social applications may use as tags the user's identifier and the identifiers of the user's social connections, allowing most operations for the same user to be restricted to a small set of nodes. Also, tags can be used to correlate messages of the same topic.

3.2.2 Application Programming Interface (API)

$\texttt{put(}String\ \texttt{collection,}K\ \texttt{key,}\ V\ \texttt{value,}\ \mathcal{P}String\ \texttt{tags)}$
$V\ \texttt{delete(}String\ \texttt{collection,}K\ \texttt{key,}\mathcal{P}String\ \texttt{tags)}$
$V\ \texttt{get(}String\ \texttt{collection,}K\ \texttt{key)}$
$K \rightharpoonup V\ \texttt{getByRange(}String\ \texttt{collection,}\ K\ \texttt{min,}\ K\ \texttt{max)}$
$\texttt{multiPut(}\ \mathcal{P}(String \times (K \rightharpoonup (V \times \mathcal{P}String))\ \texttt{mapItems)}$
$String \rightharpoonup (K \rightharpoonup V)\ \texttt{multiGet(}\mathcal{P}(String \times K)\ \texttt{keys)}$
$String \rightharpoonup (K \rightharpoonup V)\ \texttt{getByTags(}\mathcal{P}String\ \texttt{tags)}$

Figure 3.2: DataDroplets' API

The system supports common single tuple operations, such as `put`, `get` and `delete`, multi-tuple put and get operations (`multiPut` and `multiGet`), and set operations to retrieve ranges (`getByRange`), and equally tagged tuples (`getByTags`), Figure 3.2.

The first four operations in the API are restricted to a single collection. The `put` operation determines where the replicas of the tuple of the given collection should be placed based on the associated key and tags, and stores it. According to the key and value types, DataDroplets encodes them as opaque array of bytes. Then, in consonance with the single node data placement strategy, the item is given a position in the circular ring identifier, and, according to it, the node responsible for serving that tuple is found and contacted. Similar to other large scale data stores, DataDroplets does not distinguish between inserts and updates; therefore, if a tuple from the same collection with the same key already exists, it gets overwritten. It is worth mentioning that if the data placement strategy takes into account the set of tags, it may imply a move of the tuple to a new responsible node (when a tuple is updated the set of tags is modified). In addition, the `delete` operation specifies the tuple's key and collection. Optionally, the tuple's tags may be passed in order to ease DataDroplets locate the tuple, when using a tag-based single node data placement strategy. The `get` and `getByRange` operations can be, respectively, used for searching individual tuples or iterate over a subset of tuples in a given range, defined by the key of the minor tuple, min, and the key of the major tuple, max.

Regarding the three remaining operations, they are multiple collection operations. The `multiGet` and `multiPut` operations retrieve or update multiple tuples from one or more collections in parallel. The `getByTags` gets all items from any collection whose set of tags intersect with the tags in the set passed as parameter.

DataDroplets also provides operations for creating and deleting collections. As the replication level and data placement strategies (both single node and replicas) are configured on a per collection basis, collections must be explicitly created and configured before being used.

3.2.3 Request handling

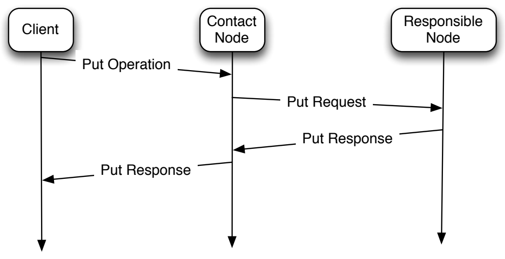
The clients route the requests through a generic load balancer that will select any node based on load information (the contact node). With this approach, the client does not have to link any code specific to DataDroplets in its application.

The contact node will then redirect, taking into account the data placement strategy (both for a single node and replicas), the request to the responsible or set of responsible nodes. For single item operations (Put, Get and Delete), the contact node sends the proper request to the responsible node, waits for the answer, and then redirects it to the client node, Figures 3.3(a), 3.3(b) and 3.3(c).

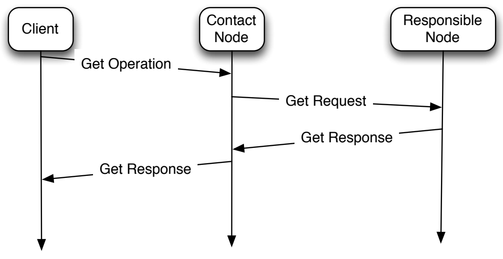
For multi-item operations (MultiPut, MultiGet, GetByRange, GetByTags), the contact node must find the set of nodes needed to answer the request, splits the request into multiple request, and initiates those requests in parallel at each responsible node. As the requests return success or failure, the contact node assembles the results, and then passes them to the client, Figures 3.3(d), 3.3(e), 3.3(f) and 3.3(g).

For metadata management operations, such as the operations for creating and deleting collections, all nodes must receive the requests and process them in the same order. Therefore, these operations cannot be forward to any node, but instead the node with the lowest identifier receives them, and then redirects them to all other nodes, and waits for their acknowledgements before answering to the client, Figure 3.3(h).

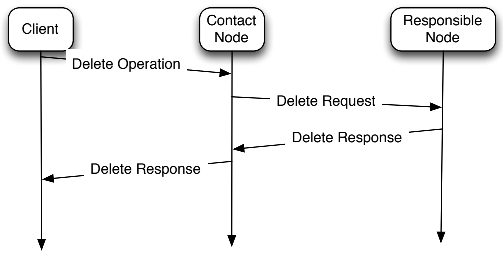
Some details about the request processing and the definition of the set of responsible nodes are tightly dependent on the collection's placement strategy. Therefore, more details about the request handling are given on Section 3.2.7.



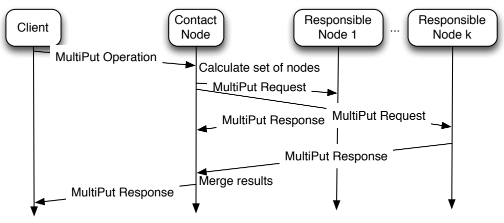
(a) Put



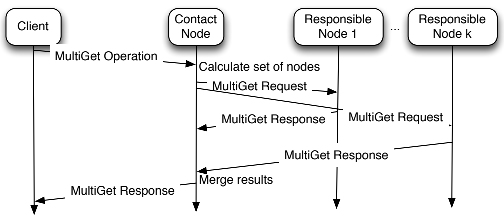
(b) Get



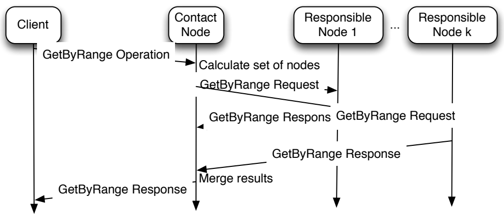
(c) Delete



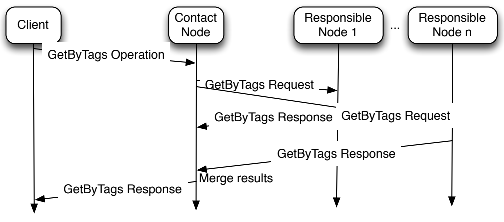
(d) Multi Put



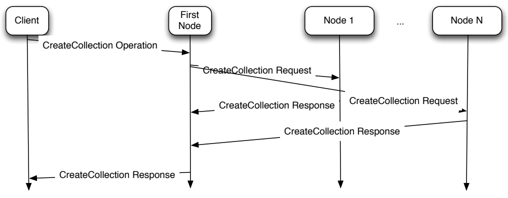
(e) Multi Get



(f) Get by range



(g) Get by tags


Client
First Node
Node 1
...
Node N
CreateCollection Operation
CreateCollection Request
CreateCollection Response
CreateCollection Request
CreateCollection Response
CreateCollection Response

(h) Create Collection

Figure 3.3: Request handling

3.2.4 Overlay management

DataDroplets builds on the Chord's structured overlay network (Stoica et al. 2001). Physical nodes are kept organized on a logical ring overlay, where each node maintains complete information about the overlay membership as in Gupta et al. (Gupta et al. 2004; Risson et al. 2006). This conforms to the assumptions about the size and dynamics of the target environments (tens to hundreds of nodes with a reasonably stable membership), and allows efficient one-hop routing of requests (Gupta et al. 2004).

Nodes have Globally Unique Identifier (GUID) and are hashed in the ring's virtual overlay, with identifiers uniformly picked from the $[0, 1[$ interval. Each node is responsible for the storage of buckets of a Distributed Hash Table (DHT) also mapped into the same interval.

On membership changes (due to nodes that join or leave the overlay), the system adapts to its new composition updating the routing information at each node, and readjusting the data stored at each node according to the redistribution of the mapping interval. In DataDroplets, this procedure follows closely the one described in OneHop (Gupta et al. 2004).

Membership changes raise two important issues. First, local information about the membership change must be updated, in order for each node in the system to determine precisely which interval in the identifier space it is responsible for. The second issue is conveying information about the change to all the nodes in the ring so that these nodes will maintain correct information about the system membership, and consequently manage to route in a single hop.

To maintain correct local information (that is, information about each node's successor and predecessor nodes), every node n runs a stabilization routine periodically, wherein it sends keep-alive messages to its successor S and predecessor P. Then, node P checks if N is indeed its predecessor, and if not, it notifies it of the existence of another node between them. Similarly P checks if N is indeed its successor, and if not it, notifies N. If either S or P does not respond, N pings it repeatedly until a time-out period, when it decides that the node is unreachable or dead.

A joining node contacts another existing node to get its view of the current membership; this protocol is similar to the Chord protocol. The membership information enables it to get in touch with its predecessor and successor, thus informing them of its presence.

To maintain correct full routing tables, notifications of membership change events must reach every node in the system within a specified amount of time. This is achieved by superimposing a well-defined hierarchy on the system. This hierarchy is used to form dissemination trees, which are used to propagate event information (Gupta et al. 2004).

Besides the automatic load redistribution on membership changes, because some workloads may impair the uniform data distribution even with a random data placement strategy, the system implements dynamic load-balancing as proposed in Karger and Ruhl (Karger and Ruhl 2004).

The protocol is randomized, and relies on the move of nodes in the DHT. The protocol receives the ϵ parameter that is any constant, $0 < \epsilon < 1$. The ϵ defines the sensitivity of the protocol to load changes. In the following $l_j$denotes the load on node j.

Each node i occasionally contacts another node j at random. If $l_i \leq \epsilon \times l_j$ or $l_j \leq \epsilon \times l_i$, then the nodes perform a load balancing operation (assume without loss of generality that $l_i > l_j$), distinguishing two cases:

- Case 1: $i = j + 1$: In this case, i is the successor of j and the two nodes handle address intervals next to each other. Node j increases its identifier in the circular ring so that the $(l_i - l_j)/2$ items with lowest addresses get reassigned from node i to node j. Both nodes end up with load $(l_i + l_j)/2$.

- Case 2: $i \neq j+1$: If $l_{j+1} > l_i$, then we set $i = j+1$, and go to case 1. Otherwise, node j moves between nodes $i - 1$, and i, to capture half of node i's items. This means that node j's items are now handled by its former successor, node $j + 1$.

3.2.5 Bootstrapping

Some DataDroplets' nodes play the role of seeds, entry points within the cluster. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Seeds can be obtained either from static configuration or from a configuration service, such as ZooKeeper (Hunt et al. 2010). Seeds are fully functional nodes.

In the bootstrap case, when a node needs to join a cluster, it reads its configuration file which contains a list of a few seeds, it chooses a random token for its position in the ring and contacts a seed that will propagate the information around the cluster.

3.2.6 Fault tolerance

In DataDroplets, the consistency is given per item. Thus, it is sufficient to have a local view of the ring. The contact node, the one that was contacted by the client and has issued the requests to the proper nodes, defines a timeout for each request. If the request fails on its first attempt (when the timeout has expired) it does not return an error to the application. Instead, requests are rerouted. If a request from contact node $n1$ to responsible node $n2$ fails, because $n2$ is no longer in the system, $n1$ can retry the query by sending it to $n2$'s successor. If the request failed, because a recently joined node, $n3$, is the new successor for the key that $n1$ is searching for, $n2$ can reply with the identity of $n3$ (if it knows about $n3$), and $n1$ can contact $n3$ in a second routing step.

Moreover, this implies that when the node responsible for a data item receives a request, it has to check if it is still responsible for that item, otherwise redirect it to the new responsible node.

Membership

The one-hop DHT used in DataDroplets allows to maintain a dynamic local membership with the ring's composition. The membership service, Figure 3.4, provides methods to: get all nodes between two nodes (`getRangeNodes`), get all nodes in the system (`getMembers`), get the node that is the successor of a given position in the ring (`getSucessor`), get the n successors of a given node (`getSucessors`), and get the predecessor of a given node (`getPredecessor`). Additionally, the membership service also notifies the following events, Figure 3.5, when: a neighbor is added between the current node and its predecessor (`neighborBeforeAdded`), the predecessor of the current node is removed (`predecessorRemoved`), and the predecessor of the current node changed its position in the ring (`moved`).

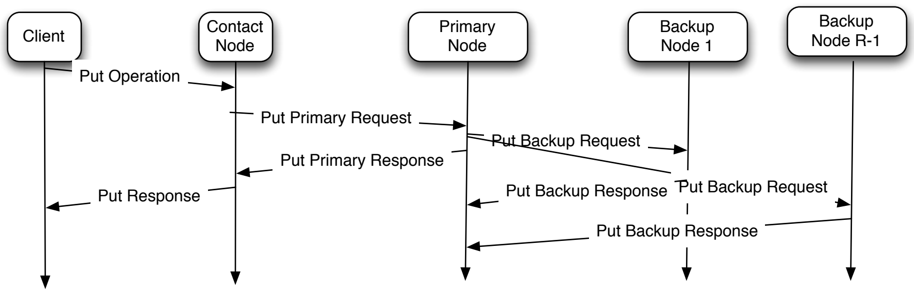
```
Finger* getRangeNodes(Finger min,Finger max)
Finger* getMembers()
Finger getSuccessor(Double position)
Finger* getSuccessors(Finger node,int n)
Finger getPredecessor(Finger node)
```

Figure 3.4: Membership service's API

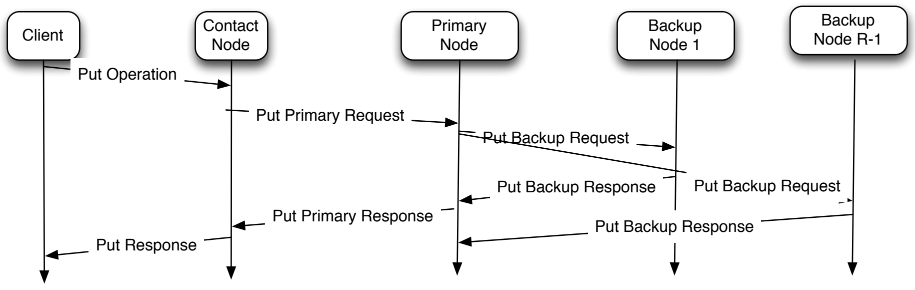
```
neighborBeforeAdded(Finger newNeighbor, Finger oldPredecessor)
predecessorRemoved(Finger oldPredecessor, Finger newPredecessor)
moved(Finger oldSucessor)
```

Figure 3.5: Membership notification events

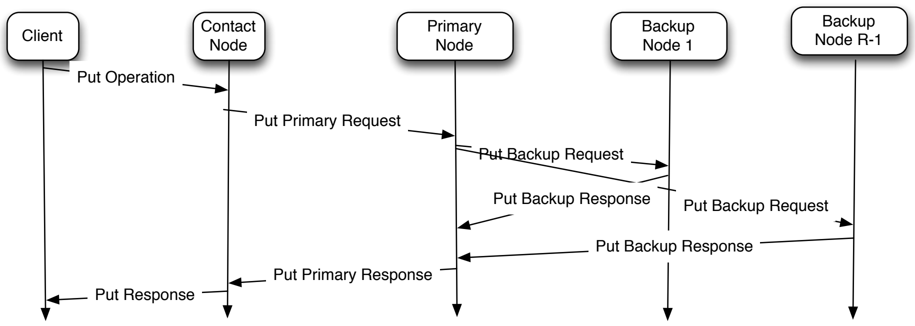
Replication



(a) None



(b) First



(c) All

Figure 3.6: Replication

In the Clouder architecture, replication is optional on the soft-state layer as data is stored persistently and in a dependable manner at the persistent-state layer. However, in DataDroplets, in order to ensure the resilience of each node's metadata, and to help load-balancing read operations, it also replicates its data.

We adopt a simple configurable primary-backup replication protocol that can work from asynchronous mode to synchronous mode (Guerraoui and Schiper 1997). As our goal is to offer stronger consistency, the default mode is synchronous. With R replicas, the primary of the item is the one determined by the single node data placement strategy, and the $R - 1$ backup's nodes are defined according to the replica placement strategy.

When receiving an update request, the contact node sends a request marked as master to the primary node responsible for that item. Then, the primary processes the client's request and forwards the results via state transfer to the other replicas — the backups. Updates can be propagated either in a synchronous, asynchronous, or semi-synchronous manner to the backup replicas. In the first variant 3.6(c), replicas are always kept consistent; thus, read operations can be safely performed at any replica. In the second variant 3.6(a), read operations on backup copies might return stale values. In the latter variant 3.6(a), the master responds to the contact node as soon as it receives the acknowledgment from one of the slaves. Read operations on backup copies might also return stale values.

The system is designed so that every node in the system can determine which nodes are the replicas of any particular tuple. It is worth mentioning that in DataDroplets data is partitioned in several partitions (according to the number of nodes, N), and the primary and, consequently, its backups are chosen on a per partition level. Thus, each partition has a different master node and backup replicas. Therefore, opposite to traditional primary-backup replication schemes, if the primary replica becomes overloaded, more nodes can be added to the system; therefore, new partitions will appear to have a different primary replica.

While a crash of a backup replica does not require specific actions by the replication protocol, a crash of the primary replica requires reconfiguration since a new primary needs to be promoted. In DataDroplets, the new primary node will be the successor of the previous primary node.

By having backups of the primary's copies of data, the replication schema, primarily target at fault tolerance, can also help on load-balancing read operations. While write operations are only sent to the primary replica, and then applied in the backup, read operations can be sent to any available replica. For single item read operations, one replica is chosen randomly to ensure load-balancing of requests, and for multi-item operations, we send the request to the minimal set of nodes that are needed to answer. For example, in the given scenario, Figure 3.7, eight nodes in the system and a query whose keys are in the ring interval [0.1,0.9], the set of nodes that have the needed data is [N2,N3,N4,N5,N5,N7,N8], but if there are three replicas for each item in the system and the two backup nodes are chosen using a successor-list replica placement strategy, then it is sufficient to send the query to the set of nodes [N2,N5,N8].

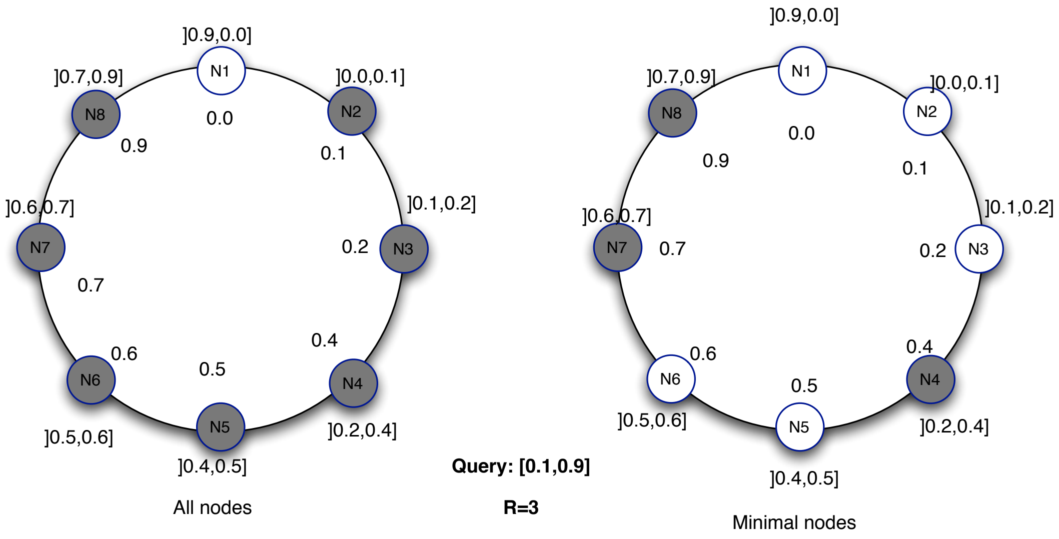


Figure 3.7: Multi-item query

State transfer

In case of failure of a node or a join of a new node, fail-over is done automatic and the state transfer process is started. The state transfer process is triggered in node N when it detects it has a new predecessor or when it detects its old predecessor has failed. The operations that node N performs when any of these situations happens are described in detail below. When the node N detects changes in its predecessor, it:

1. Splits all node N's primary data into backup data, the one between the old predecessor's position and the new predecessor's position, and the new primary data, the one between the new predecessor's position and this node's position.
2. Sends all the data from whose the node is a backup to the new predecessor.

3. Discards data from the most distant backups, backup R.

4. Sends the `ShiftRightMessage` to all backup nodes, whose primary node was N.

Each node S that receives the `ShiftRightMessage` splits the backup data from node N, as if it was being split in two partitions, and if S is the most distant backup of node N, it discards backup data for the new node.

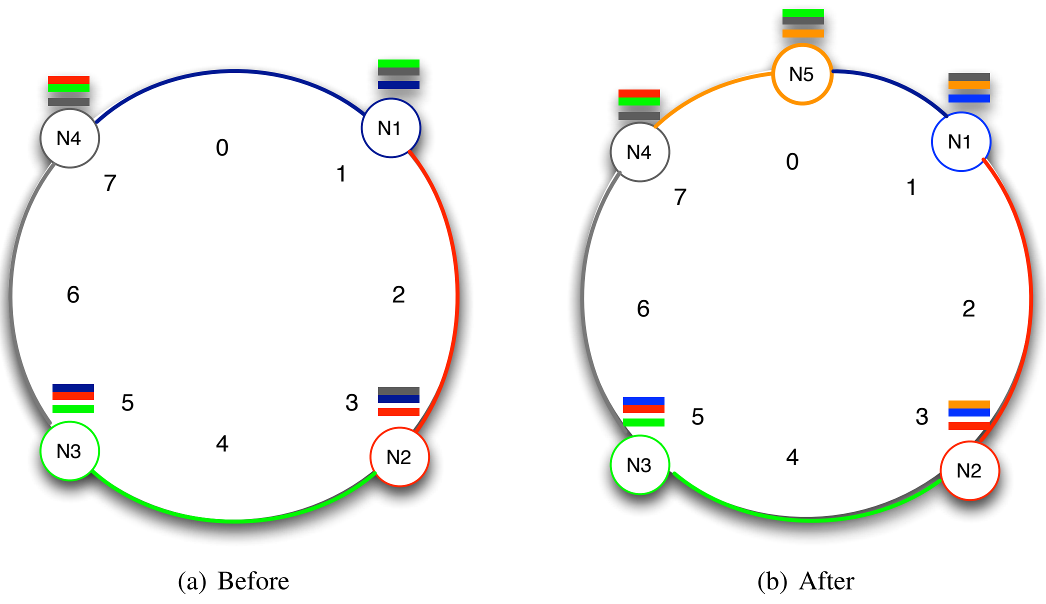


Figure 3.8: Data transfer example

Consider the following example of a ring with four nodes, and where each partition is replicated in two backup nodes, Figure 3.8(a). Each node stores three partitions of data. The first is the partition from which it is the primary, and in the Figure it is shown right at the top of the node. The other two partitions are the ones from which the node is a backup replica. In the Figure, at the top of the primary partition is the backup data from the node nearby, its predecessor. The last partition is the one from the node at a distance of two nodes, predecessor(predecessor).

When node $N5$ joins the system, the membership service will notify node $N1$ that it has a new predecessor; thus, it shall split its data in half (dark blue) with the new node (orange). The partition of data at the end of the transfer is depicted in Figure 3.8(b). Briefly, the changes are:

- Node $N2$ discards the backup data from node $N4$ (gray), and splits the old backup data from node $N1$ (dark blue) in two backup data partitions (light blue and orange).
- Node $N1$ discards backup data from node $N3$, and splits its old primary data (dark blue) into new primary data (light blue) and backup data from the primary data of the new node $N5$ (orange).
- Node $N5$ receives from node $N1$ all of its backups and primary data, where its primary data (orange) is a subset of the old primary data from node $N1$ (dark blue).

3.2.7 Data placement

DataDroplets supports several data placement strategies: random, ordered and a novel *tagged* strategy. The data placement strategy is defined on a collection basis.

The *random placement* is the basic load-balancing strategy present in most DHTs (Stoica et al. 2001; Gupta et al. 2004) and also in most large scale data stores (DeCandia et al. 2007; Cooper et al. 2008; Lakshman and Malik 2010). The *ordered placement* takes into account order relationships among items' primary key favoring the response to *range* oriented reads, and is present in some large scale stores (Cooper et al. 2008; Chang et al. 2006; Lakshman and Malik 2010). This order needs to be disclosed by the application and can be per application, per workload or even per request. We use an order-preserving hash function (Garg and Gotlieb 1986) to generate the identifiers. Compared to a standard hash function for a given ordering relation among the items, an order-preserving hash function *hashorder()* has the extra guarantee that if $o1 < o2$, then $hashorder(o1) < hashorder(o2)$.

The major drawback of the *random placement* is that items which are commonly accessed by the same operation may be distributed across multiple nodes. A single operation may need to retrieve items from many different nodes leading to a performance penalty.

Regarding the *ordered placement*, in order to make the order-preserving hash function uniform as well, we need some knowledge on the distribution of the item's keys (Garg and Gotlieb 1986). For a uniform and efficient distribution we need to know the domain of the item's key, the minimum and maximum values. This yields a trade-off between uniformity and reconfiguration. While a pessimistic prediction of the domain will avoid further reconfiguration, it may break the uniformity. In the current implementation of DataDroplets, the hash function is not made uniform, but, as described in Section 3.2.4, we use a more general approach to balance load.

A key aspect of DataDroplets is the multi-item access that enables the efficient storage and retrieval of large sets of related data at once. Multi-item operations leverage disclosed data relations to manipulate sets of comparable or arbitrarily related elements. The performance of multi-item operations depends heavily on the way correlated data is physically distributed.

The balanced placement of data is particularly challenging in the presence of dynamic and multi-dimensional relations. This aspect is the main contribution of the next section, describing a novel data placement strategy based on multidimensional locality-preserving mappings. Correlation is derived from disclosed tags dynamically attached to items.

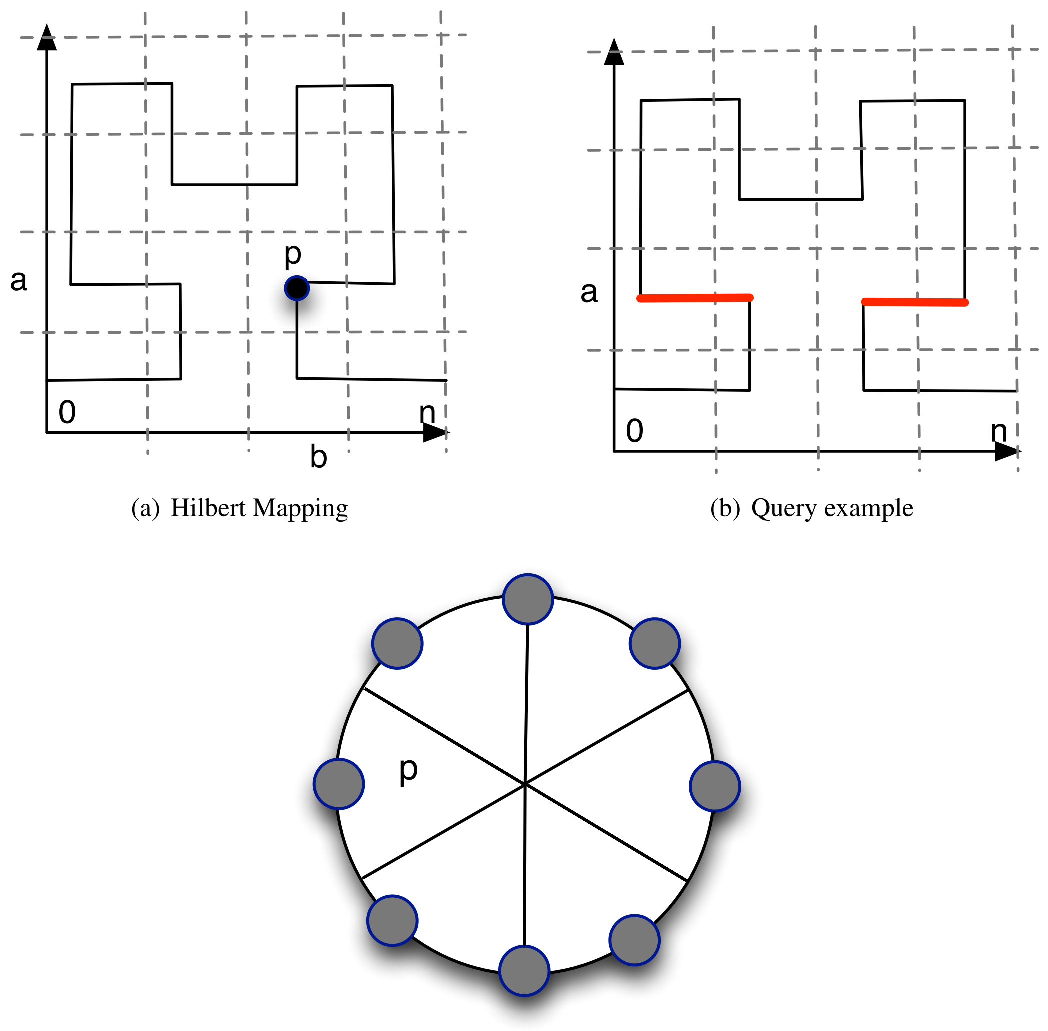
Tagged placement

The placement strategy, called hereafter *tagged*, implements the data distribution according to the set of tags defined per item. A relevant aspect of our approach is that these sets can be dynamic. This allows us to efficiently retrieve correlated items that were previously attached by the application. The strategy uses a dimension reducing and locality-preserving indexing scheme that effectively maps the multidimensional information space to the identifier space, $[0, 1[$.

Tags are free-form strings and form a multidimensional space where tags are the coordinates and the data items are points in the space. Two data items are co-located if they have equal-sized set of tags and tags lexicographically close, or if one set is a sub-set of the other.

This mapping is derived from a locality-preserving mapping called Space Filling Curves (SFCs) (Sagan 1994). A SFC is a continuous mapping from a d-dimensional space to a unidimensional space ($f : N^d \rightarrow N$). The d-dimensional space is viewed as a d-dimensional cube partitioned into sub-cubes, which is mapped onto a line such that the line passes through each point (sub-cube) in the volume of the cube once, entering and exiting the cube only once. Using this mapping, a point in the cube can be described by its spatial coordinates or by the length along the line, measured from one of its ends.

SFCs are used to generate the one-dimensional index space from the multidimen-



(a) Hilbert Mapping (b) Query example

(c) Hybrid-n placement strategy

Figure 3.9: Tagged placement strategy

sional tag space. Applying the Hilbert mapping (Butz 1971) to this multidimensional space, each data element can be mapped to a point on the SFC. Figure 3.9(a) shows the mapping for the set of tags $\{a, b\}$. Any range query or query composed of tags can be mapped into a set of regions in the tag space, and corresponding to line segments in the resulting one-dimensional Hilbert curve. These line segments are then mapped to the proper nodes. An example for querying tag $\{a\}$ is shown in Figure 3.9(b), which is mapped into two line segments.

If an update is made to a previous item without knowing its previous tags, it must find which node has the requested item and then update it. If the update also updates its tags, the item will be moved from the old node, defined by old tags, to the new node, defined by new tags.

As this strategy only takes into account tags, all items with the same set of tags will have the same position in the identifier space; therefore, will be allocated to the same node. To prevent this, we adopt a hybrid-n strategy. Basically, we divide the set of nodes into n partitions, and the item's tags, instead of defining the complete identifier into the identifier space, define only the partition, as shown in Figure 3.9(c). The position inside the partition is defined by a *random* strategy. The *random* strategy was chosen in order to have uniform distribution of nodes within the partition. Therefore, the locality is only preserved inter partition.

Request handling

Most of request processing is tightly dependent on the collection's placement strategy. For the `put` and `multiPut` this is obvious as the target nodes result from the chosen placement strategy.

For operations that explicitly identify the item by key, `get`, `multiGet` and `delete`, the node responsible for the data can be directly identified when the collection is distributed at *random* or *ordered*. Having the data distributed by tags, all nodes need to be searched for the requested key.

For `getByRange` and `getByTags` requests, the right set of nodes can be directly identified if the collection is distributed with the *ordered* and *tagged* strategies, respectively. Otherwise, all nodes need to be contacted and need to process the request.

3.2.8 Replica placement strategy

Further to the novel tagged placement strategy, DataDroplets also supports traditional random and ordered data placement strategies. Each strategy is biased to some kind of operation and none is better for all kind of operations. Briefly, the $random$ strategy is the best for workloads where single random tuple operation, `get`, dominate; the $ordered$ strategy is the best for workloads where set operations to retrieve ranges, `getByRange`, dominate; and the novel $tagged$ strategy is the best for workloads where set operations for equally tagged items, `getByTags`, dominate. Therefore, they are not suited for a multi-tenant environment, where workloads with different dominating operations may coexist.

As DataDroplets replicates data, it must also have a replica placement strategy. Thus, we have defined a new strategy that combines a multi-hashing strategy with successor-list replication. The application defines, on a per collection basis, a list of single node placement strategies with the size being the desired number of replicas, R. If the same strategy is used on more than one replica, the node for the replica with the lowest position in the list is chosen using the defined single node strategy, and other replicas using the same strategy are defined from the successor's list in the order defined in the list. For example, the list $[random, ordered, tagged, tagged]$, defines that items of that collection will have four replicas. The first replica of an item with key K, will be defined using the random placement strategy as *successor(random(k))*, the second replica will be defined using the ordered strategy as *successor(ordered(k))*, the third replica will be defined using the tagged strategy *successor(tagged(k))*, and the last replica will be defined as the successor node of the first tagged replica, *successor(successor(tagged(k)))*.

When processing a request, DataDroplets automatically chooses from the available placement strategy for that collection the most adequate one, trying to minimize the number of requests.

3.3 Prototype

The DataDroplets prototype has been implemented in Java using the ProtoPeer toolkit (Galuba et al. 2008) with about 15k lines of code. ProtoPeer is a rapid distributed systems prototyping toolkit that allows switching between event-driven simulation and live network deployment without changing the application code.

The key architectural features which enable this are the abstract time and networking APIs. The APIs allow only a small number of basic operations: creation of timers for execution scheduling, and creation of network interfaces for sending and receiving messages.

ProtoPeer is event-driven, and the system is composed of a set of peers that communicate with one another by passing messages. Each application defines its set of messages and message handlers. Also, an application typically defines a set of timers and handlers for the timer expiration events. Most of the application logic is called from within the timer and message handlers.

A peer in a distributed system typically implements more than one piece of message passing functionality. In ProtoPeer the message passing logic and state of each of the protocols is encapsulated in components called peerlets. Peers are constructed by putting several peerlets together. The peerlets can also be removed or added at runtime. The peerlets, just as the applets or servlets, have the familiar init-start-stop lifecycle. The peer provides the execution context for all of the peerlet instances it contains. The peerlets can discover one another within that context, and use one another's functionality. Peerlets can export well defined interfaces, for example, a DHT interface, which can have several implementations that can be easily swapped.

Regarding the event-driven simulation from ProtoPeer, we have used the network simulation model and extended it with simulation models for CPU as in Xiongpai et al. (Xiongpai et al. 2009). For the CPU simulation we have used a hybrid simulation approach as described in Sousa et al. (Sousa et al. 2005). Briefly, the execution of an event is timed with a profiling timer, and the result is used to mark the simulated CPU busy during the corresponding period thus, preventing other events to be attributed simultaneously to the same CPU. A simulation event is then scheduled with the execution delay to free the CPU. Further pending events are then considered. Therefore, only the network latency is simulated, and the other execution times are profiled from real execution.

For the live network deployment, ProtoPeer uses a non-blocking I/O library, Apache MINA[1]. We have used Apache Mina 1.1.3.

For the prototype of DataDroplets, we implemented two peerlets: SimpleOneHop and DataDroplets. The former is the implementation of OneHop DHTl (Gupta et al. 2004) used in DataDroplets, and the latter is the DataDroplet's prototype.

3.3.1 SimpleOneHop

This peerlet implements the OneHop protocol for maintaining the set of peer neighbors. On `start()` this peerlet waits for the list of bootstrap peers from the `BootstrapClient` and sends one `ConnectRequest` to each of the bootstrap peers. When it receives a `ConnectRequest`, the peerlet adds the sender to the neighbor set and responds back to the sender with a `ConnectResponse`. When this peerlet receives a `ConnectResponse`, it adds the sender to the neighbor set.

[1]`http://mina.apache.org/`

The peerlet uses a CircularDoubleLinkedList data structure to store the local view membership. The prototype uses the ProtoPeer `Timer` to run the stabilization routine periodically and to detect the time-out.

The peerlet depends on existing peerlets from the ProtoPeer framework, the `BootstrapClient` and `NeighborManager` peerlets. The NeighborManager is a peerlet that manages the set of neighbors of the peer. Neighbors can be added or removed, iterated over or retrieved either by their peer ID or their network address. The BootstrapClient is a peerlet that on `init()`, uses the `idGenerator` supplied in the constructor to initialize the peer's identifier. Then, on `start()`, it contacts the `BootstrapServer` and gets the initial set of neighbors. After that, it calls `bootstrapCompleted` on all of its listeners and becomes idle.

3.3.2 DataDroplets

This is the peerlet that implements DataDroplets. It receives operations from clients and then: discovers to which nodes the request must be sent, sends the requests to the proper nodes, waits for all responses, and notifies the client of the result. There is a single instance of this peerlet per node.

For each client an instance of a `ClientWorker` is created. When the `DataDroplets` peerlet receives a request, it finds the proper `ClientWorker` which will handle it. The `ClientWorker`, depending on the type of the operation: discovers to which nodes the request must be sent, sends the requests to the proper nodes, waits for all responses, and notifies the client of the result.

Data Placement

`KeyGenerator` is the interface to any single node data placement strategy, Figure 3.10. The `ItemBasedKeyGenerator` is the interface to all data placement strategies, taking into account the key of the item. The generator calculates the item's ring position only based on the item itself. The
`double generateKey(Object key)` calculates the item position in the ring within the interval $[0, 1[$ based on the item's key. The
`OrderPreservingKeyGenerator` is the interface to an item-based single node data placement strategy that also preserves order. `TagBasedKeyGenerator` is the interface to single node data placement strategies, based on the item's tags. The gen-

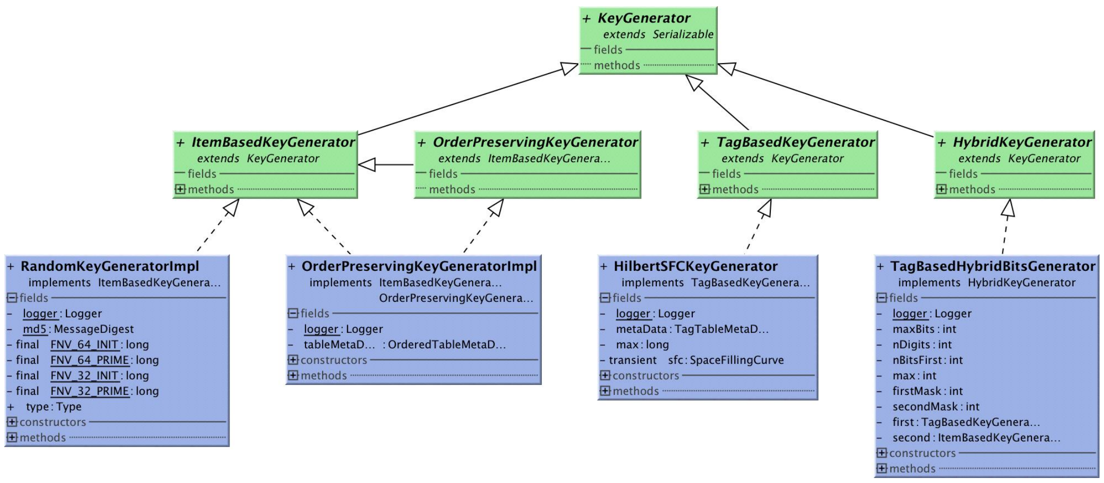


Figure 3.10: Class diagram for single node data placement strategies

erator calculates the item's ring position only based on the defined tags. Tags allow to establish relations between items. The method `double generateKey (Set<String> tags)` calculates the item's position in the ring within the interval $[0, 1[$ based on the given set of tags. `HybridKeyGenerator` is the interface to hybrid-based key generators that combine an item-based generator with a tag-based generator. The generator has the `public double generateKey(Object key, Set<String> tags)` method, and calculates the item's ring position based on both the item's key and tags.

`RandomKeyGeneratorImpl` is an implementation of an item-based key generator using a random hash function. It supports several hashing mechanisms, such as Java default hash code, CRC (Cyclic Redundancy Check) with 32 bits, FNV (Fowler–Noll–Vo) with both 32 and 64 bits[2], and KETAMA (uses MD5)[3]. By default it uses the KETAMA implementation, as from the tests conducted, it presents the most uniform distribution.

`OrderPreservingKeyGeneratorImpl` is the implementation of an item- and order-based key generator. It receives the minimum and maximum expected values for the domain, and distributes them in the interval [0,1[, using an order preserving hash function.

`HilbertSFCKeyGenerator` is the implementation of the novel single node data placement strategy described in Section 3.2.7. It implements the

[2]`http://tools.ietf.org/html/draft-eastlake-fnv-03`
[3]`http://www.audioscrobbler.net/development/ketama/`

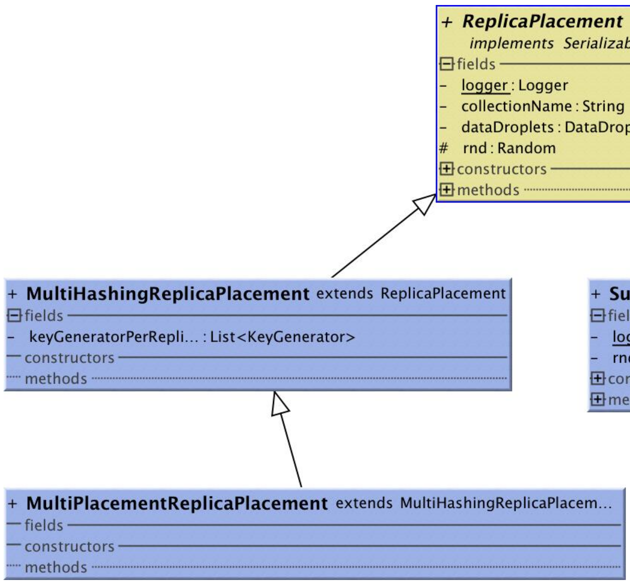

+ ReplicaPlacement
implements Serializable
fields
- logger : Logger
- collectionName : String
- dataDroplets : DataDroplets
# rnd : Random
constructors
methods
+ MultiHashingReplicaPlacement extends ReplicaPlacement
fields
- keyGeneratorPerRepli... : List<KeyGenerator>
constructors
methods
+ SuccessorListReplicaPlacement extends ReplicaPlacement
fields
- logger : Logger
- rnd : Random
constructors
methods
+ MultiPlacementReplicaPlacement extends MultiHashingReplicaPlacem...
fields
constructors
methods

Figure 3.11: Class diagram for replica placement strategies

`TagBasedKeyGenerator` interface, and uses a Hilbert Space Filling Curve. For the implementation of Space Filling Curves, we have used the Uzaygezen library[4] that is a Multi-Dimensional Indexing with Space Filling Curves, with some modifications and optimizations.

`TagBasedHybridBitsGenerator` is the implementation of a hybrid-n strategy that divides the set of nodes into n partitions, and the item's tags define only the partition instead of defining the complete identifier into the identifier space, as previously shown in Figure 3.9(c). As the result of the hash function is an array of bytes, the implementation uses the most significant $nBitsFirst$ from the tag-based key generator, and the other bits from an item-based key generator.

`ReplicaPlacement`, is the abstract class which offers several methods for all replica placement strategies, Figure 3.11. The main methods are:

- $abstract \quad Finger \quad getFingerAtPos(int \quad replica, double \quad ringPosition)$: Returns the node that is the nth $replica$ of the item whose ring identifier is $ringPosition$.

- $abstract \quad Finger \quad getReplica(int \quad replica, Finger \quad finger)$: Returns the node that is the nth backup replica of the primary node $finger$.

[4] `http://code.google.com/p/uzaygezen/`

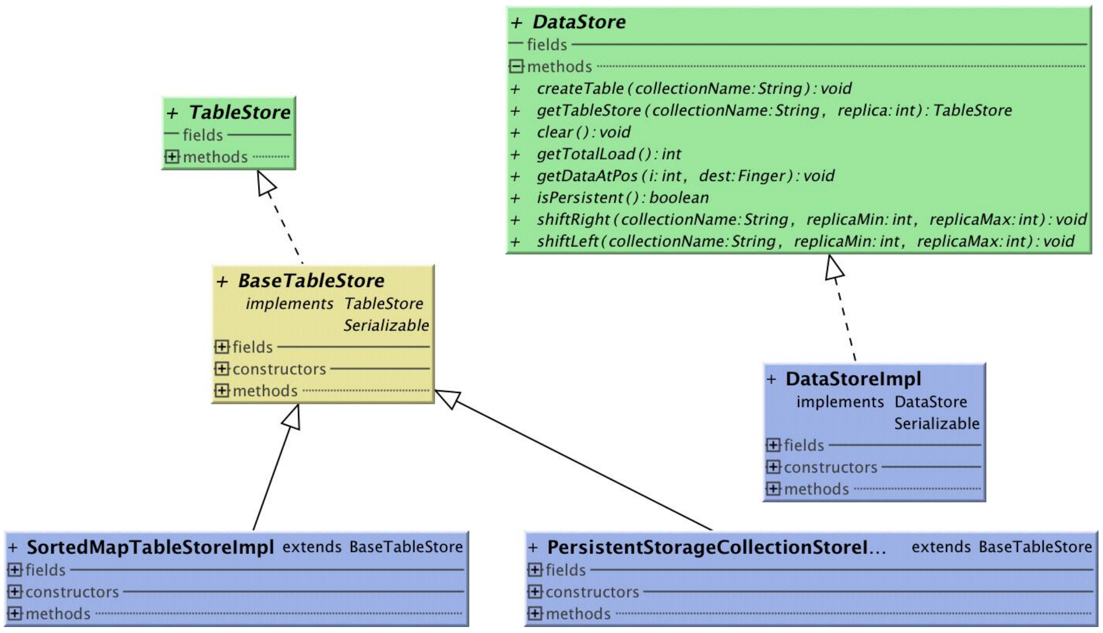

+ DataStore
fields
methods
+ createTable (collectionName: String) : void
+ getTableStore (collectionName: String, replica: int) : TableStore
+ clear () : void
+ getTotalLoad () : int
+ getDataAtPos (i: int, dest: Finger) : void
+ isPersistent () : boolean
+ shiftRight (collectionName: String, replicaMin: int, replicaMax: int) : void
+ shiftLeft (collectionName: String, replicaMin: int, replicaMax: int) : void
+ TableStore
fields
methods
+ BaseTableStore
implements TableStore Serializable
fields
constructors
methods
+ DataStoreImpl
implements DataStore Serializable
fields
constructors
methods
+ SortedMapTableStoreImpl extends BaseTableStore
fields
constructors
methods
+ PersistentStorageCollectionStoreI... extends BaseTableStore
fields
constructors
methods

Figure 3.12: Class diagram for data store classes

- $List < Finger > \quad getReplicas(Comparable \quad key, Set < String > \quad tags)$: Returns all the replicas of the item with the given key and set of tags.

- $List < Finger > \quad getBackupReplicas$: Returns all the backup replicas of the item with the given key and set of tags.

`SuccessorListReplicaPlacement` is a concrete implementation of a replica placement strategy that uses the successor-based replica placement strategy. `MultiHashingReplicaPlacement` is a concrete implementation of a replica placement strategy that uses the successor-based replica placement strategy when multiple replicas of the same type exist. `MultiPlacementReplicaPlacement` is a concrete implementation of the novel replica placement strategy that combines a multi-hashing strategy with successor-list replication, as described in Section 3.2.8.

Data Store

`DataStore`, is the interface to a data store that handles the storage of data in DataDroplets, Figure 3.12. It allows to create table stores and manipulate existing ones, for new collections. `TableStore` is the interface to a table store. All the methods are asynchronous; therefore, they receive a callBack to notify using `handleFinishedOperation` when the operation has finished and the result is available. The main methods are:

- $put(TableStoreOperationsListener \quad listener, Comparable \quad key, Object \quad data, Set < String > \quad tags$: Sets the value associated with the given key to the given data and tags.
- $get(TableStoreOperationsListener \quad listener, Comparable \quad key)$: Gets the value associated with the given key or null if there is no value associated with the key.
- $multiGet(TableStoreOperationsListener \quad listener, Set < Comparable > keys)$: Gets the values associated with the given keys or null if there is no value associated with the key.
- $getByTag(TableStoreOperationsListener \quad listener, Set < String > \quad tags)$: Gets all the values matching the given tags.
- $delete(TableStoreOperationsListener \quad listener, Comparable \quad key)$: If an item exists with the given key, deletes it.
- $getRange(TableStoreOperationsListener \quad listener, Comparable \quad minKey, Comparable \quad maxKey)$: Retrieves the subset of tuples in a given range defined by the key of the minor tuple, $minKey$, and the key of the major tuple, $maxKey$.

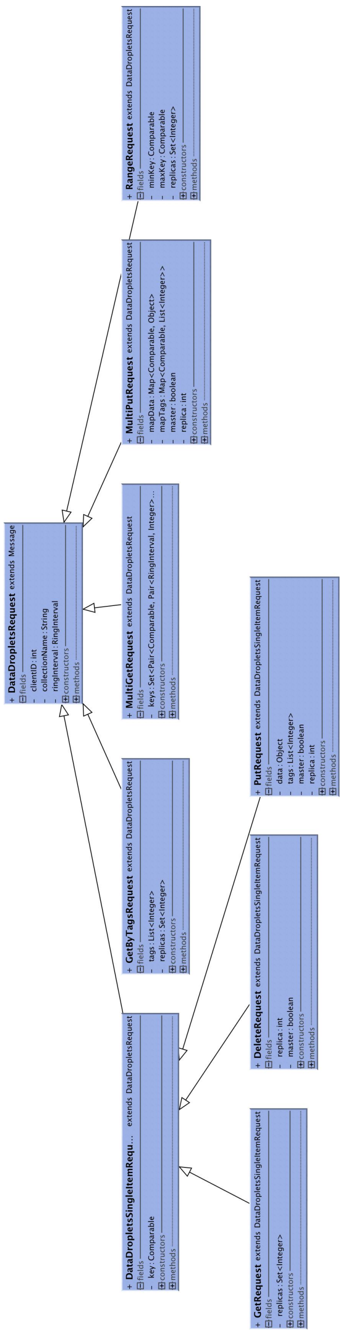
While the soft-state layer of Clouder, DataDroplets, was not designed to persist data, the current prototype of DataDroplets supports both in-memory storage and a persistent storage, as the persistent-state layer of Clouder is not available.

`SortedMapTableStoreImpl` is an implementation of a table store that stores all data in-memory, using a Red-Black tree, Java `TreeMap`. `PersistentStorageCollectionStoreImpl` is an implementation of a table store that stores all data locally in a persistent manner using Berkeley DB Java edition 4.0 [5].

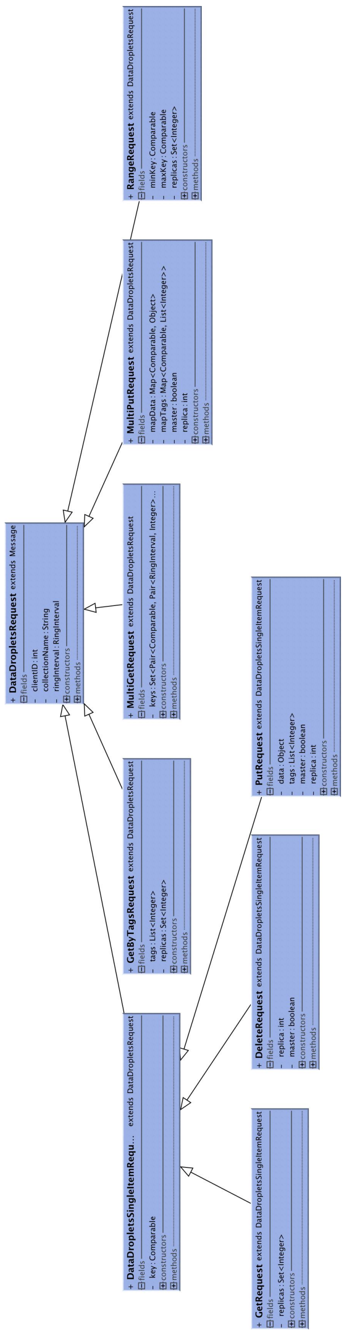
[5] `http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html`

Messages

The main messages used by DataDroplets are depicted into Figure 3.13.


+ RangeRequest extends DataDropletsRequest
+ MultiPutRequest extends DataDropletsRequest
+ DataDropletsRequest extends Message
+ MultiGetRequest extends DataDropletsRequest
+ PutRequest extends DataDropletsSingleItemRequest
+ GetByTagsRequest extends DataDropletsRequest
+ DataDropletsSingleItemRequ... extends DataDropletsRequest
+ DeleteRequest extends DataDropletsSingleItemRequest
+ GetRequest extends DataDropletsSingleItemRequest

(a) Requests


+ RangeRequest extends DataDropletsRequest
+ MultiPutRequest extends DataDropletsRequest
+ DataDropletsRequest extends Message
+ MultiGetRequest extends DataDropletsRequest
+ PutRequest extends DataDropletsSingleItemRequest
+ GetByTagsRequest extends DataDropletsRequest
+ DataDropletsSingleItemRequ... extends DataDropletsRequest
+ DeleteRequest extends DataDropletsSingleItemRequest
+ GetRequest extends DataDropletsSingleItemRequest

(b) Responses

Figure 3.13: DataDroplets' messages

The `CreateTableRequest` and `CreateTableResponse` are used to create a new collection request. The request is first sent from the contact node to one of the nodes, and then to all other nodes. The response is sent from all other nodes to the first node and then it replies to the client with the result. The creation is only successful if it succeeds at all other nodes. The request has the `collectionName` as a String and the set of configurations associated to it, given by `List<CollectionMetaData> metaData`, one per replica. The response has the `collectionName` as a String and the `result` as a boolean.

The `DataDropletsRequest` and the `DataDropletsResponse` are the parent class for all DataDroplets' request and response messages, respectively. Similarly, the `DataDropletsSingleItemRequest` and the `DataDropletsSingleItemResponse` are respectively the parent class of all DataDroplets' single item request messages, and have two attributes: the item's key as a `Comparable`, and the item's collectionName as `String`.

The `LoadInfoMessage` is used during the load balancer process and has the load of a given number (the number of data items it holds) and the node's identifier. The `ShiftRightMessage` and the `MoveDataBetweenPeersMessage` are used during the state transfer process, either triggered by the join/failure of a node or by the load balancer process. The `ShiftRightMessage` has info about how many shifts the node has to perform, given by the distance of the target backup from the primary replica, and the definition of the partition of the new node, given by the ring interval (min and max). `MoveDataBetweenPeersMessage` message is sent from a primary replica, splitting half of its data, and has information about: the node's new position, all collection's metadata, all collection's data as `Map<String,Map<Integer,Map<Comparable,Object>>>` and all collection's tags as `Map<String,Map<Integer,Map<Comparable,Set<String>>>>`.

Finally, the `ProxyCDHTRequest` is used to redirect a `DataDropletsRequest` to another node, when nodes fail.

3.4 Evaluation

In this section we evaluate our implementation of DataDroplets and its data placement strategies. For this, we ran a series of experiments to evaluate the performance of DataDroplets, under a workload representative of applications currently exploiting the scalability of emerging data stores.

As there is neither an available version of most of considered data stores nor enough available machines to run them, in the following, we present performance results for DataDroplets in a simulated setting. We evaluate the enhanced data model and the cost additional consistency guarantees, and its scalability by substantially increasing the number of nodes. Moreover, we also evaluated the suitability of the different data placement strategies in both simulated and real settings.

3.4.1 Test workloads

In this section we give some details about the benchmark's framework implemented in ProtoPeer, and the detailed description of a novel benchmark mimicking the usage of the Twitter social network.

Benchmark's Framework

The `BenchMark` is an abstract class that defines the shared state to all benchmark implementations. It has a `StatsCollector` instance and a list of registered listeners for benchmark events. The main methods are:

- `boolean hasMoreInitOperations():` returns true if the benchmark has more initialization operations.
- `DataDropletsOperation nextInitOperation():` returns the next initialization operation corresponds to a DataDroplets operation.
- `BenchMarkClient createNewClient(Clock clock):` creates a new client for the given benchmark.

`BenchMarkClient` is an interface with the following methods:

- `boolean hasMoreOperations():` returns true if the client has more operations.
- `Pair<DataDropletsOperation,Double> nextOperation():` returns a pair, being the left side the next operation and the right side the think time between the previous operation and this one.
- `int getClientID():` returns the client's unique identifier.

`StatsCollector` is used to store and present statistics for a given benchmark. The main methods are:

- `registerRequest(double startTime,int opID,int nodeID, double latency,boolean write,String requestType)` : registers a new operation with the given parameters. The $requestType$ defines the type of operation.
- `int getTotalRequests(String requestType)` : returns the total number of operations of a given type. If the $requestType$ is null, returns the total for all types of operations.
- `double getRequestsMean(String requestType)` : returns the mean of latency of operations of a given type. If the $requestType$ is null, returns the mean for all types of operations.
- `set/getFinishTime(Time finishTime)` : setter and getter for the finish time of the benchmark.
- `set/getStartTime(Time startTime)` : setter and getter for the start time of the benchmark.

`BenchMarkInit` is a peerlet that waits until the initial ring is complete, and then starts executing benchmark's initialization operations on `DataDroplets`. When all initialization operations have finished, it sends a *BenchInitFinish* message to all nodes.

The `BenchMarkClientExecutor` is a peerlet which executes the specified number of clients for each DataDroplets' node, given by $Map < NetworkAddress, Integer >$. The peerlet waits until it receives the *BenchInitFinish*, and then starts executing client operations on each of the specified DataDroplets' nodes. It finishes when all the client operations finished. The `BenchMarkExecutor` uses a fixed thread pool to run the benchmarks in live experiments.

`MicroBenchmark` is a simple benchmark with a single client that runs a predetermined number of operations for each type of DataDroplets' operation (`Put`, `Get`, `GetByRange`, `GetByTags`, `Delete`), and validates its results.

Twitter workload

As there is no available benchmark for large scale data stores that mimics general purpose multi-item operations, a workload that mimics the usage of the Twitter social network was defined.

Twitter is an on-line social network application which offers a simple micro-blogging service, consisting of small user posts, the *tweets*. A user gets access to other user tweets by explicitly stating a *follow* relationship.

The central feature of Twitter is the user *timeline*. A user's timeline is the stream of tweets from the users one*follows* and from one's own. Tweets are free form strings up to a hundred and forty characters. Tweets may contain two kinds of tags, user mentions formed by a user's ID preceded by @ (for example, @john), and hashtags, arbitrary words preceded by # (for example, #topic), meant to be the target of searches for related tweets.

Our workload definition has been shaped by the results of recent studies on Twitter (Java et al. 2007; Krishnamurthy et al. 2008; Boyd et al. 2010). In particular, we consider just the subset of the seven most used operations from the Twitter's API (Twitter 2010) (Search and REST API as of March 2010):

`List<Tweet>statuses_user_timeline(String userID, int s, int c)` retrieves from userID's tweets, in reverse chronological order, up to c tweets starting from s (read-only operation).

`List<Tweet>statuses_friends_timeline(String userID, int s, int c)` retrieves from userID's timeline, in reverse chronological order, up to c tweets starting from s. This operation allows to obtain the user's timeline incrementally (read-only operation).

`List<Tweet>statuses_mentions(String userID)` retrieves the most recent tweets mentioning userID's, in reverse chronological order (read only operation).

`List<Tweet>search_contains_hashtag(String topic)` searches the system for tweets containing topic as hashtag (read-only operation).

`statuses_update(Tweet tweet)` appends a new tweet to the system (update operation).

`friendships_create(String userID, String toStartUserID)` allows userID to follow toStartUserID (update operation).

`friendships_destroy(String userID, String toStopUserID)` allows userID to unfollow toStopUserID (update operation).

For the implementation of the test workload we consider a simple data model of three collections: `users`, `tweets` and `timelines`. The `users` collection is keyed by userID, and for each user it stores profile data (name, password, and date of creation), the list of the user's followers, a list of users the user follows, and the user's tweetID, an increasing sequence number. The `tweets` collection is keyed by a compound of userID and tweetID. It stores the tweets' text and date, and associated user and topic tags if present. The `timelines` collection stores the timeline for each user. It is keyed by userId, and each entry contains a list of pairs (tweetID, date) in reverse chronological order.

In a nutshell, the operations listed above manipulate these data structures, as follows. The `statuses_update` operation reads and updates the user's current tweet sequence number from `users`, appends the new tweet to `tweets`, and updates the timeline for the user and each of the user's followers in `timelines`. The `friendships_create` and `friendships_destroy` operations update the involved users' records in `users`, and recomputes the followers' `timelines` adding or removing the most recent tweets from the followed, or unfollowed, user. Regarding the read-only operations, `statuses_friends_timeline` simply accesses the specified user timeline record in `timelines`, `statuses_user_timeline` accesses a range of the user's tweets, and `statuses_mentions` and `search_contains_hashtag` the `tweets` collection in general.

Twitter's network belongs to a class of scale-free networks, and exhibit a small world phenomenon (Java et al. 2007). As such, the set of users and their follow relationships are determined by a directed graph, created with the help of a scale-free graph generator (Barabási and Bonabeau 2003).

Workload is firstly initialized with a set of users (that remains unchanged throughout the experiments), a graph of follow relationships (that is updated during the workload run due to the `friendships_create` and `friendships_destroy` operations) and a set of tweets. The initial population of tweets is needed in order to fulfill `statuses_user_timeline`, `statuses_friends_timeline` and `statuses_mentions` requests right from the start of the experiments. The generation of tweets, both for the initialization phase and for the workload, follows a couple of observations over Twitter traces (Krishnamurthy et al. 2008; Boyd et al. 2010). First, the number of tweets per user is proportional to the user's followers (Krishnamurthy et al. 2008). From all tweets, 36% mention some user and 5% refer to a topic (Boyd et al. 2010). Mentions in tweets are created by randomly choosing a user from the set of friends. Topics are chosen using a power-law distribution (Java et al. 2007).

Each run of the workload consists of a specified number of operations. The next operation is randomly chosen, taking into account the probabilities of occurrence de-

Table 3.1: Probability of Operations

Operation	Probability
search_contains_hashtag	15%
statuses_mentions	25%
statuses_user_timeline	5%
statuses_friends_timeline	45%
statuses_update	5%
friendships_create	2.5%
friendships_destroy	2.5%

picted in Table 3.1. To our knowledge, no statistics about the particular occurrences of each of the Twitter operations are publicly available. The figures of Table 3.1 are biased toward a read intensive workload, and based on discussions that took place during Twitter's Chirp conference 2010.[6]

The defined workload may be used with both large scale data stores and traditional relational databases.[7]

3.4.2 Experimental setting

For all experiments presented next, the performance metric has been the average request latency as perceived by the clients. The system was populated with sixty-four topics for tweets and an initial tweet factor of a thousand. An initial tweet factor of n means that a user with f followers will have $n \times f$ initial tweets. For each run, five-hundred thousand operations were executed. Different request loads have been achieved by varying the clients think time between operations. Throughout the experiments no failures were injected.

Simulated setting

The network model of ProtoPeer was configured to simulate a LAN with latency uniformly distributed between 1 ms and 2 ms. Regarding CPU, each node was configured and calibrated to simulate one dual-core AMD Opteron processor running at 2.53GHz. All data has been stored in memory, persistent storage was not considered.

[6]The Twitter official developers conference, `http://pt.justin.tv/twitterchirp/b/262219316`.

[7]The workload is available at `https://github.com/rmpvilaca/UBlog-Benchmark`.

The `DataDropletsSimulatedExperiment` allows to start the simulation of the system with DataDroplets's N nodes, and running the given benchmark. The parameters are: a file path to store statistics; number of DataDroplets nodes; the load balancing interval: the ϵ parameter for load balancing; whether or not the optimization of minimizing replicas is to be used; the primary backup replication strategy (synchronous, asynchronous, or semi-asynchronous); the benchmark to be executed and its parameters. For the Twitter benchmark the parameters are: number of populated users; number of concurrent clients; total number of operations; think time between consecutive operations.

For each node, the experiment adds the following peerlets: NeighborManager, SimpleOneHop, BootstrapClient, and DataDroplets. Additionally, one of the nodes add the following peerlets: BootstrapServer, BenchMarkInit, and BenchMarkClientExecutor.

In detail, the initialization of the peerlets is as follows:

```
Peer newPeer = new Peer(peerIndex);
if (peerIndex == 0) {
    newPeer.addPeerlet(new BootstrapServer());
}
newPeer.addPeerlet(new NeighborManager());
newPeer.addPeerlet(new SimpleOneHop(nPeers 1));
newPeer.addPeerlet(new BootstrapClient(Experiment.getSingleton().getAddressToBindTo(0),
new SimplePeerIdentifierGenerator()));
newPeer.addPeerlet(new DataDroplets(persistent,minimizeReplicas,replicationStrategy,
        cacheSize,true,loadBalancingInterval,deltaLoadBalancing));
if (peerIndex == 0) {
    newPeer.addPeerlet(new BenchMarkInit(benchmark));
    newPeer.addPeerlet(new BenchMarkClientExecutor(benchmark));
}
```

The Twitter workload has been populated with ten-thousand concurrent users, and the same number of concurrent users were simulated (uniformly distributed by the number of configured nodes).

Real setting

We used a machine with 24 AMD Opteron Processor cores running at 2.1GHz, 128GB of RAM and a dedicated SATA hard disk. We ran twenty instances of Java Virtual Machine (1.6.0) running ProtoPeer, and all data has been stored persistently using the `PersistentStorageCollectionStoreImpl` implementation that stores all data locally using Berkeley DB.

We have used two live experiments: `DataDropletsLiveExperiment` to run DataDroplets' nodes, and `BenchmarkLiveExperiment` to run Benchmark nodes. The `BenchmarkLiveExperiment` allows to start a live execution of a given benchmark with the given parameters: a file path to store statistics; number of replicas and their placement strategy for the benchmark's collections; list of DataDroplets nodes node1IP:node1Port,node2IP:node2Port ...; the benchmark to be executed and its parameters. For the Twitter benchmark the parameters are: number of populated users; number of concurrent clients; total number of operations; think time between consecutive operations. `DataDropletsLiveExperiment` allows to start a live peer of DataDroplets with the given parameters: initial peers; the size of the cache used in the data store; the load balancing interval; the ϵ parameter for load balancing; whether or not the optimization of minimizing replicas is to be used; the primary backup replication strategy (synchronous, asynchronous, or semi-asynchronous).

The workload has been populated with two-thousand and five-hundred concurrent users and the same number of concurrent users were run (uniformly distributed by the number of configured instances). During all the experiments IO was not the bottleneck.

3.4.3 Results

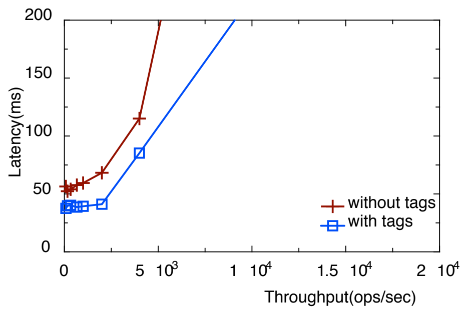
Evaluation of DataDroplets

Figure 3.14(a) depicts the response time for the combined workload. Overall, the use of tags in DataDroplets to establish arbitrary relations among tuples consistently outperforms the system without tags with responses 40% faster.

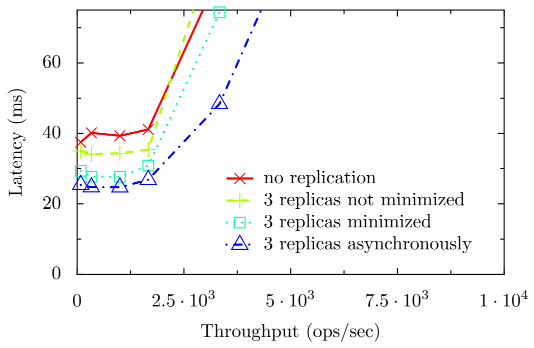
When using tags, DataDroplets may use the data partition strategy that takes into account tuple correlations; therefore, it stores correlated tuples together. As the workload is composed of several operations that access correlated tuples, the access latency when using tags is lower than without tags, as other data partition strategies, which only take into account a single tuple, may be used.

Evaluation of node replication

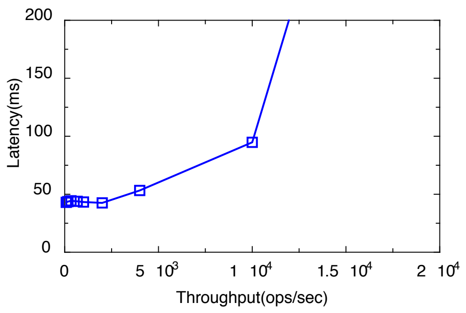
Data replication in DataDroplets is meant to provide fault tolerance to node crashes, and improve read performance through load balancing. Figure 3.14(b) shows the results of the combined workload when data is replicated over three nodes.



(a) 100 nodes configuration without replication



(b) 100 nodes with replication



(c) 200 nodes configuration without replication

Figure 3.14: System's response time

The *minimized* and *not minimized* lines correspond both to a synchronous replication algorithm. As explained in Section 3.2.6, the minimized strategy takes advantage of replication to minimize the number of nodes contacted per operation, having each node responding for all the data it holds, while the other, on the contrary, leverages replication to increase concurrent accesses to different nodes. The overall response time is improved by 22% with the *minimized* strategy.

Despite the impact replication inevitably has on write operations, the overall response time is improved by 27% (due to a high read-only request rate in the workload). Moreover, we can see that the overall gain of asynchronous replication is up to 14%, despite the additional impact synchronous replication inevitably has on these operations, which would not, *per se*, justify the increased complexity of the system. It is actually the dependability aspect that matters most, allowing to provide seamless fail over of crashed nodes.

Evaluation of the system's elasticity

To assess the system's response to a significant scale change, we conducted the previous experiments over the double of the nodes, two-hundred. Figure 3.14(c) depicts the results.

Here, it should be observed that while the system appears to scale up very well providing almost the double of throughput before getting into saturation, for a small workload, up to two-thousand ops/sec with two-hundred nodes, there is a slightly higher latency. This result motivates a judicious elastic management of the system to maximize performance, let alone for economical and environmental reasons.

Evaluation of data placement strategies

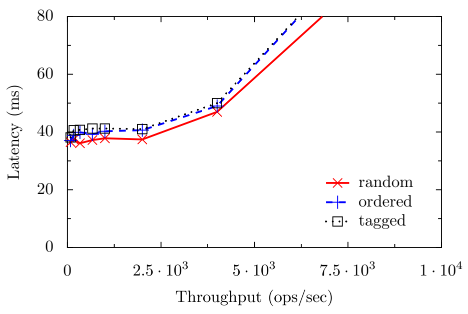
Simulated setting The graphs in Figure 3.15 depict the performance of the system when using the different placement strategies available in the simulated setting. The workload has been firstly configured to only use the random strategy (the most common in existing key-value stores), then configured to use the ordered placement for both the `tweets` and `timelines` collections (for `users` placement has been kept at random), and finally configured to exploit the tagged placement for `tweets` (`timelines` were kept ordered and `users` at random). The random, ordered and tagged lines in Figure 3.15 match these configurations.

We present the measurements for each of the seven workload operations (Figure 3.15(a) through 3.15(g)), and for the overall workload (Figure 3.15(h)). All runs were conducted with a hundred nodes.

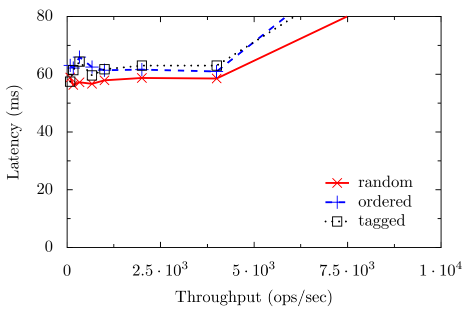
We can start by seeing that for write operations (`statuses_update` and `friendships_destroy`) the system's response time is very similar for all scenarios (Figures 3.15(a)and 3.15(b)). Both operations read *one* user record, and subsequently add or update one of the tables. The costs of these operations are basically the same in all the placement strategies.

The third writing operation, `friendships_create`, has a different impact, though (Figure 3.15(c)). This operation also has a strong read component. When creating a follow relationship, the operation performs a `statuses_user_timeline` which, as can be seen in Figure 3.15(d), is clearly favored when tweets are stored in order.

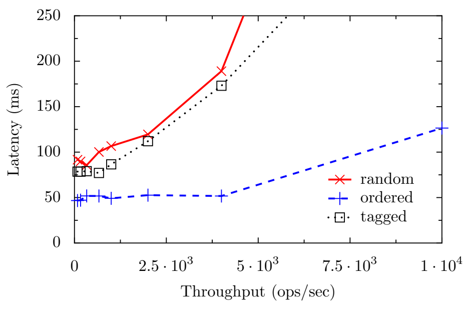
Regarding read-only operations, the adopted data placement strategy may have a high impact on latency, see Figures 3.15(d) through 3.15(g).



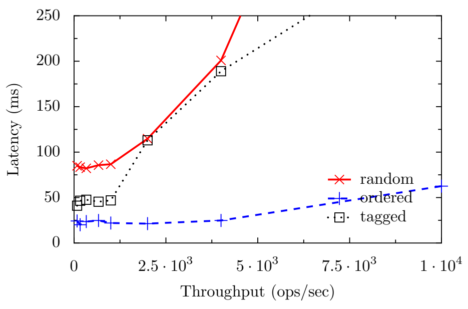
(a) statuses_update op



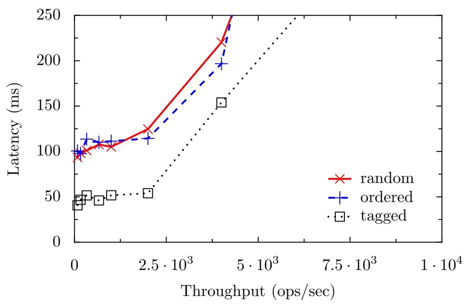
(b) friendships_destroy op



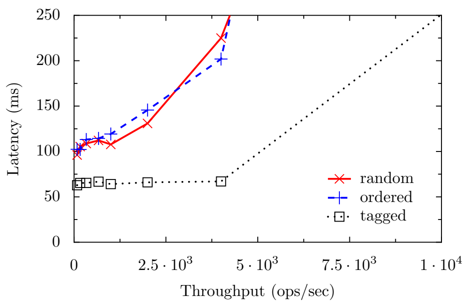
(c) friendships_create op



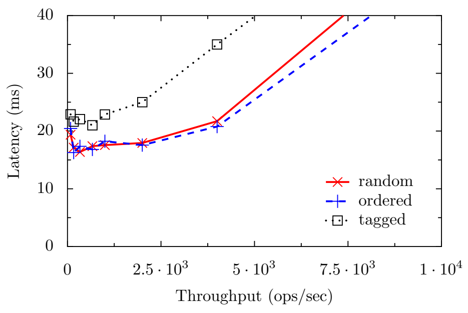
(d) statuses_user_timeline op



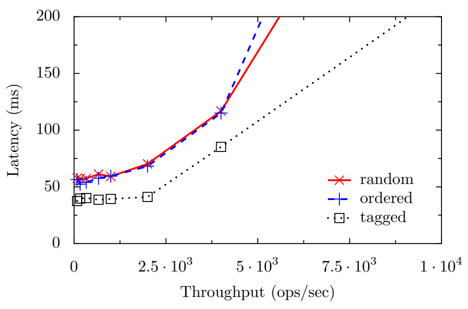
(e) statuses_mentions op



(f) search_contains_hashtag op



(g) statuses_friends_timeline op



(h) Overall workload

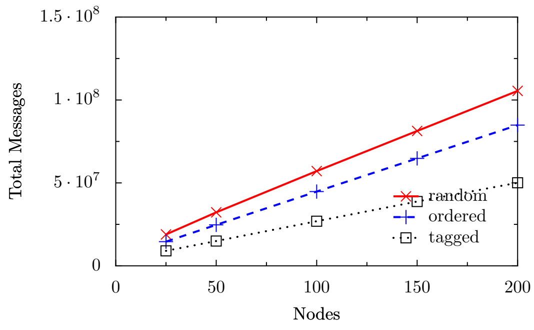
Figure 3.15: System's response time with a hundred simulated nodes

The statuses_user_timeline operation (Figures 3.15(d)) is mainly composed by a range query (which retrieves a set of the most recent tweets of the user); therefore, it is best served when tweets are (chronologically) ordered, minimizing the number of nodes contacted. Taking advantage of SFC's locality preserving property grouping by tags still considerably outperforms the random strategy before saturation.

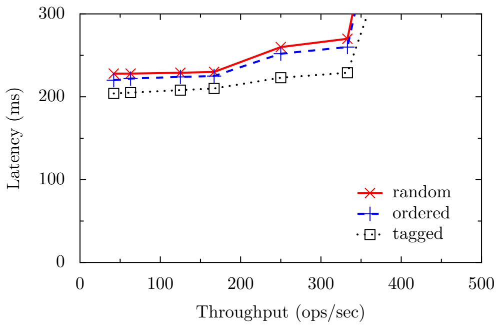
Operations status_mentions and search_contains_hashtag are essentially correlated searches over tweets, by user and by topic, respectively. Therefore, as expected, they perform particularly well when the placement of tweets uses the tagged strategy. For status_mentions, the tagged strategy is twice as fast as the others, and search_contains_hashtag keeps a steady response time up to ten thousand ops/sec, while with the other strategies the systems struggle from the beginning.

Operation statuses_friends_timeline accesses the tweets collection directly by key and sparsely. To construct the user's timeline, the operation gets the user's tweets list entry from timelines, and for each tweetID it reads it from tweets. These end up being direct and ungrouped (that is, single item) requests and, as depicted in Figure 3.15(g), best served by the random and ordered placements.

Figure 3.15(h) depicts the response time for the combined workload. Overall, the new SFC-based data placement strategy consistently outperforms the others with responses 40% faster.



(a) Total number of messages exchanged with system size



(b) System's response time

Figure 3.16: Additional evaluation results

Finally, it is worth noticing the substantial reduction of the number of exchanged messages attained by using the tagged strategy. Figure 3.16(a) compares the total number of messages exchanged when using the random and tagged strategies. This reduction is due to the restricted number of contacted nodes by the tagged strategy in multi-item operations.

Real setting Figure 3.16(b) depicts the response time for the combined workload in the real setting. The results in the real setting confirm the previous results from the simulated setting. Overall, the new SFC-based data placement strategy consistently outperforms the others.

The additional response time in the real setting, compared with the simulated setting, is due to the use of a persistent storage.

Evaluation of replica placement strategy

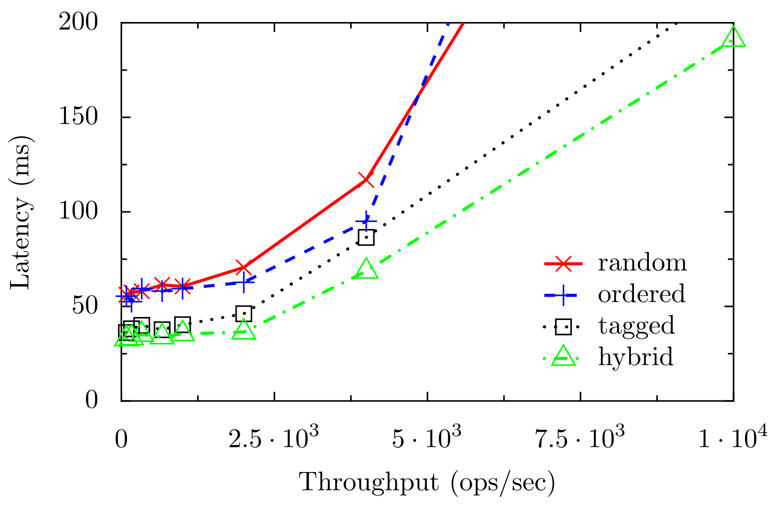


Figure 3.17: Replica placement results

In all the experiments, three replicas were used. The *random*, *ordered*, and *tagged* lines use three replicas using the same data placement strategy (the primary is determined by the data placement strategy, while the backups are determined by the successor-list replica strategy). In the *hybrid* line, the novel replica placement strategy is being used with a different data placement strategy per replica, the list of replicas is $[random, ordered, tagged]$.

Figure 3.4.3 depicts the response time for the new replica placement strategy. As the replica placement strategy chooses the best strategy for each type of operation, the overall performance is increased regarding the best standalone strategy, tagged.

3.4.4 Summary of results

DataDroplets aims at shifting the trade-offs established by current data stores toward the needs of common business users. It provides additional consistency guarantees and higher-level data processing primitives, smoothing the migration path for existing applications. Specifically, DataDroplets adjusts to the access patterns required by most current applications, which arbitrarily relate and search data by means of free-form tags.

The results show the benefit, in request latency, of DataDroplets' enhanced data model and API, the minimal cost of synchronous replication, and attest the scalability of DataDroplets.

The novel data placement strategy, based on multidimensional locality preserving mappings, adjusts to access patterns found in many current applications, which arbitrarily relate and search data by means of free-form tags, and provides a substantial improvement in overall query performance. Additionally, we show the usefulness of having multiple simultaneous placement strategies in a multi-tenant system by supporting also ordered placement, for range queries, and the usual random placement. Moreover, the new replica placement strategy that combines different replica placement strategies presents the low latency.

3.5 Discussion

Current large scale data stores focus on a specific narrow trade-off regarding consistency, availability and migration cost that fits tightly their very large internal application scenarios. While current approaches use either a hierarchical or decentralized architecture, Clouder uses a hybrid architecture with two collaborating layers that ease the offering of additional consistency guarantees and higher-level data processing without hindering scalability.

For some applications single tuple and range operations are not enough. These applications have multi-tuple operations that access correlated tuples. However, current data store APIs only provide single tuple operations or at most range operations over tuples of a particular collection. Therefore, DataDroplets extends the data model of current data stores with tags, allowing to establish arbitrary relations between tuples, which allows to efficiently retrieve them through a tag-based data partition strategy.

Moreover, while availability is commonly assumed, data consistency and freshness is usually severely hindered. Current data stores offer varying levels of tuple consistency, but only PNUTS and Bigtable can offer tuple atomicity. However, in both the burden is left to the application that must deal with multiple tuple versions. In DataDroplets, if an application needs atomic guarantees per tuple, it simply uses the default synchronous replication mode, and it will obtain it transparently without having to maintain and deal with multiple tuple versions. Additionally, the replication at the soft-state layer will be complemented with replication at the persistent-state layer, where items are massively replicated through gossiping.

While existing data stores do data partitioning, taking into account only a single tuple, randomly or in an ordered manner, DataDroplets also supports a data partition strategy that takes into account tuple correlations. Currently, it supports three data partition strategies: random placement, ordered placement, and tagged placement that handles dynamic multi-dimensional relationships of arbitrarily tagged tuples. The novel *tagged* data placement strategy, based on multidimensional locality preserving mappings, adjusts to access patterns found in many current applications, which arbitrarily relate and search data by means of free-form tags.

Moreover, the optimal solution to the data placement problem is dependent on the workload. However, all existing strategies are biased to some kind of operation and are not suited for a multi-tenant environment, where workloads with different requirements may coexist. Therefore, we define a new strategy that combines successor replication with a multi-hashing replica placement strategy. This allows the system to automatically adapt to different workloads with different types of operations.

Chapter 4

SQL on large scale data stores

Most Web-scale applications, such as Facebook, MySpace, and Twitter, remain SQL-based for their core data management (Rys 2011). Particularly, one of the most requested additions to the Google App Engine platform has been a SQL database (Google 2012).

Given the prevalence of SQL as a query language for databases, Web-scale applications can highly benefit from an efficient SQL query engine running on large scale data stores, which is able to scale with the database. Without full SQL support, it is hard to provide a smooth migration, and this is a hurdle to the adoption of large scale data stores by a wider potential market. Moreover, a high number of tools coupled to SQL have been developed over the years. Consequently, having full SQL support means that all these tools become immediately available to developers of Web-scale applications.

Support for efficient SQL on an elastic data store poses several architectural challenges in the query engine, in fact, to the same extent as any scalable query engine (Stonebraker et al. 2007; Stonebraker and Cattell 2011). In addition, it requires solving the mismatch between the relational model and the data model of the target large scale data store.

In this chapter, we present a proposal for easing migration of existing SQL application code by providing a complete implementation of SQL with a standard JDBC client interface that can be plugged in existing applications and middleware (for example, object-relation mappers).

First, Section 4.1 presents the assumptions we make on the distributed query engine. Section 4.2 describes the challenges for the efficient support of SQL on a large scale data store. Section 4.3 introduces the proposed architecture. Section 4.4 describes how it is implemented using Derby components and HBase. Finally, Section 4.5 presents the experimental evaluation of the overhead and scalability of our approach.

4.1 Assumptions

The query engine can be used from a JPA engine or directly by applications based on SQL interfaces using JDBC and ODBC drivers. The transaction management is performed by an external layer providing snapshot isolation (Berenson et al. 1995), and the query engine must intercept transactional SQL queries, and explicitly provide the start and commit timestamps for a transaction. Thus, it is assumed that the underlying large scale data store provides the abstraction of a transactional tuple store. It should expose primitive tuple set, get and scan operations that enforce snapshot isolation semantics.

We assume that multiple instances of the query engine are used not only for high availability, ensuring that the query engine is not a single point of failure, but also for scalability to cope with a workload composed by a large number of concurrent transactions. Moreover, for elasticity, instances of the query engine must be spawned and terminated dynamically while the system is running.

We target potentially complex queries involving multiple relational joins; but assuming that they are highly selective and output (both as interim and final results) a small to moderate number of rows when executed optimally. These queries are issued, for instance, when navigating mapped object relations with Java Persistence Query Language (JPQL). Applications using these queries belong to the constant and bounded query scaling classes (Armbrust et al. 2011), and include On-Line Transaction Processing (OLTP), and also interactive web applications, such as social networking applications.

4.2 Challenges

Efficient and scalable support of SQL on an elastic large scale data store poses the following main challenges:

- Traditional RDBMS architectures include several legacy components that are not suited to modern hardware, and impose an important overhead to transaction processing (Harizopoulos et al. 2008) limiting both performance and scalability.
- The relational model and the data model of the data store present impedance mismatches that have to be addressed (Meijer and Bierman 2011).
- To have acceptable performance, several challenges appear when processing data stored in a large scale data store.

4.2.1 Scalable query processing

Traditional Relational Database Management Systems (RDBMS) are based on highly centralized, rigid architectures that fail to cope with the increasing demand for scalability as well as dependability, and are not cost-effective. High performance RDBMS invariably rely on mainframe architectures or clustering based on a centralized shared storage infrastructure (Lahiri et al. 2001). Although easy to setup and deploy, these often require large investments upfront and have limited scalability (Stonebraker et al. 2007). Even distributed and parallel databases, which have been around for decades (Özsu and Valduriez 1999), build on the same architecture; thus, they have the same scalability limitations.

One common solution for traditional database scalability is sharding, where data is partitioned across multiple databases (NetLog). Briefly, tables are broken horizontally and distributed across multiple independent RDBMS servers. This is done using fixed boundaries on data, and re-partitioning is necessary for load-balancing. However, this is operationally complex, and the process of re-partitioning imposes a major load on I/O resources. Thus, most of the times sharding nullifies the key benefits of the relational model while increasing total system cost and complexity.

One of the major reasons for this resides in traditional RDBMS architectures that include several components, such as on-disk data structures, log-based recovery, and buffer management, which were developed years ago, but are not suited to modern hardware. Those components impose a huge overhead to transaction processing (Harizopoulos et al. 2008) limiting both performance and scalability, and they should be removed from a scalable query processing system.

A DataBase Management System (DBMS) application that requires scalable performance must offer high-level languages without jeopardizing performance, provide

High Availability (HA) and automatic recovery, and allow to do almost all operations on line and have administrative simplicity (Stonebraker and Cattell 2011).

If the query engine is embedded within the client application, there is no communication overhead between the application and the query engine, and a high-level language can be offered to applications without jeopardizing performance. Moreover, if the query engine component is mostly stateless it can easily scale horizontally.

Using a query engine component, which is stateless regarding data and can be executed without coordination among different client application instances, allows a query engine to start without losing any data when an instance fails. Moreover, as the query engine and the large scale data store scale independently the query engine takes advantage of the elastic and high available properties of the used large scale data store to achieve high availability with automatic recovery.

In order to minimize down-time, the database should be designed in a way that most operations do not require to take the database off line. This is a major problem in current RDBMS. By taking advantage of the underlying large scale data store, we support flexible schema to add or remove attributes to an existing database without interruption of the service. Moreover, as indexes are stored in the large scale data store and as it provides flexible schema, they can be added or dropped. Regarding provisioning, for the query engine layer new instances can be easily added or removed from the system, while other instances remain on line. As the large scale data store is fully elastic, it also allows to increase or decrease the number of nodes without hindering availability.

4.2.2 Data model mismatch

Large scale data stores use a simple key-value store or at most variants of the Entity-Attribute-Value (EAV) model (Nadkarni and Brandt 1998). This data model allows to dynamically add new attributes that only apply to certain tuples. This flexibility of the EAV data model is helpful in domains where the problem is itself amenable to expansion or change over time. Another benefit of the EAV model that may help in the conceptual data design is the multi-value attributes in which each attribute can have more than one value.

One of the challenges faced is the mapping of the relational model to large scale data store's data models, while supporting SQL queries. Moreover, it implies mapping relational tables and indexes to the data store tables in such a way that the processing capabilities of the data store are exploited at its best.

In a relational table, each column is typed (for example, char, date, integer, decimal, varchar), and in indexes data is ordered according to the natural order of the column data type. However, row keys in most large scale data store are plain byte arrays. Therefore, in order to build and store indexes in the large scale data store while preserving the data type's order, we need to map row keys into plain bytes in such a way that when the large scale data store compares them, the order of the relational data type is preserved. Another mismatch lies in relational databases both primary and secondary indexes can be composite, defined from multiple columns. Thus, we also need to define a way to encode multiple indexed columns in the same large scale data store row's key.

4.2.3 Performance

The performance challenges in the proposed architecture arise from large scale data stores offer a simple tuple store interface, allowing applications to insert, query, and remove individual tuples or at most range queries based on the primary key of the tuple. The range queries allow for fast iteration over ranges of rows, and allow to limit the number and decide which column are to be returned. However, they don't support partial key scans, but index scans in RDBMS must perform equals and range queries on all or a subset of the fields in the index.

Moreover, query processing relies on data statistics and the cost of each operator for proper selection of the best query plan. However, this is done taking into account a cost model defined for the current architecture of RDBMS. Thus, a major challenge is how to adapt this cost model when using a query engine on a scalable large scale data store.

4.3 Architecture

The proposed architecture is shown in Figure 4.2, in the context of a large scale data store and a traditional RDBMS. A major motivation for a large scale data store is scalability. As depicted in Figure 4.1, a typical large scale data store builds on a distributed setting with multiple nodes of commodity hardware. Just by adding more nodes into the system (that is, scaling out), one can not only increase the overall performance and capacity of the system, but also its resilience, and thus availability by means of data

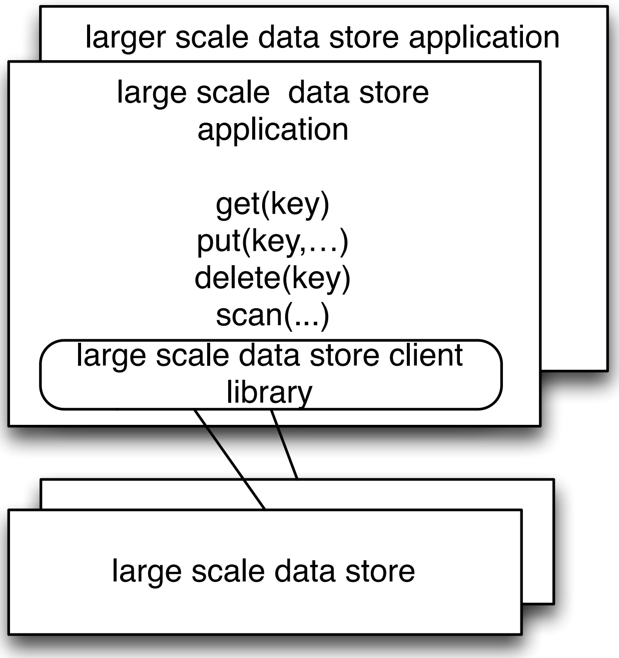

larger scale data store application
large scale data store application
get(key)
put(key,…)
delete(key)
scan(...)
large scale data store client library
large scale data store

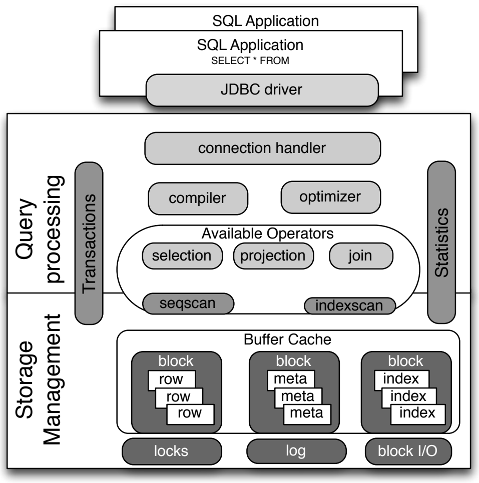
Figure 4.1: Large scale data stores architecture.

replication. By allowing clients to directly contact multiple fragments and replicas, the system can also scale in terms of clients connected. To make this possible, they provide a simple data model as well as primitive querying, and searching capabilities that allow applications to insert, query, and remove individual items or at most range queries based on the primary key of the item (Vilaça et al. 2010).

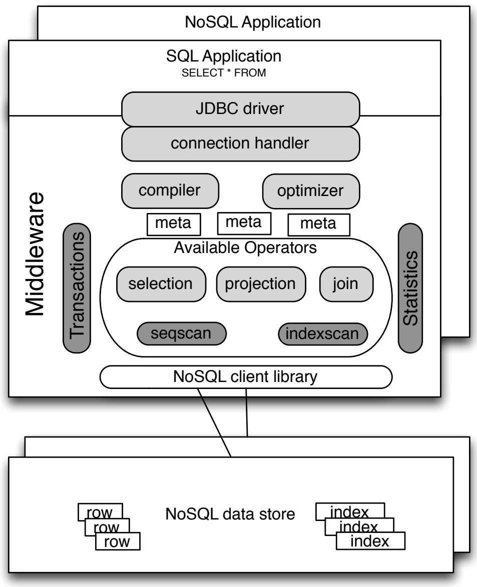
In sharp contrast, a RDBMS is organized as tables called relations, and developers are not concerned with the storage structure but instead express queries in a high-level language, SQL. SQL allows applications to conduct complex filtering and processing capabilities, such as filtering, joining, grouping, ordering and counting.

Our proposal builds on a rewrite of the internal architecture of RDBMS by reusing some existing components, by adding new components on large scale data stores as well as removing several components that are not needed on modern hardware, which would limit scalability. Toward understanding how components can be reused in our proposal, we examine the internals of traditional RDBMS architecture, dividing it roughly in a *query processor* and a *storage manager* functions, Figure 4.2(a).

The query processor is responsible for offering a relational SQL-based API for applications, and to translate the application queries, comprising two main stages: compilation and execution of the query. The compilation stage includes: (i) Starting from an initial SQL query, the query is parsed and a parse tree for the query is generated; (ii) the parse tree is converted to an initial implementation plan, represented by an algebraic expression, which is then optimized using algebraic laws – the logical query plan; (iii) the physical query plan is then constructed from the logical query plan, which implies the choice of the operators to be used and the order in which they must be executed, based on the statistics provided by the storage layer in order to select an efficient execution plan. In the execution stage, the physical plan with the expected lowest cost is executed calling into the storage layer through scan operators and performing actual data manipulation. The storage manager is responsible for actually storing and retrieving rows. Very briefly, this means mapping logical data items to physical blocks, providing different indexing data structures, and performing locking and logging for isolation and recovery.



(a) Centralized SQL RDBMS.



(b) Distributed query engine (DQE).

Figure 4.2: Data management architectures.

The architecture proposed (Figure 4.2(b)) reuses a number of components from the SQL query processor (shown in light gray). In detail, these are: the JDBC driver and client connection handler, the compiler and the optimizer, and a set of generic relational operator implementations. These components can be shielded from changes as they depend only on components that are re-implemented (shown in medium gray), providing the same interfaces as those that in the RDBMS embody the centralized storage functionality (shown in dark gray) and that are removed from our architecture. In detail, the components that have to be re-implemented are:

- A mapping from the relational model to the data model of a large scale data store. This includes: atomic data types and their representation; representation of rows and tables; representation of indexes.
- A mapping from the relational schema to the datastore that allows data to be interpreted as relational tables.
- Implementation of sequential and index scan operators. This includes: matching the interface and data representation of the datastore; taking advantage of indexing and filtering capabilities in the datastore to minimize data network traffic.
- Computation and storage of statistics within the datastore, conveying statistics to the optimizer to enable correct planning decisions.
- A stub of the transaction management.

The proposed architecture has the key advantage of being stateless regarding data. In fact, Data Manipulation Language (DML) statements can be executed without coordination among different client application instances. Therefore, the proposed architecture should retain the seamless scale-out of the large scale data store and application.

4.4 Prototype

The prototype, DQE, has been developed in the context of the CumuloNimbo [1] European Union Seventh Framework Programme (FP7), project under grant agreement n. 257993.

In the context of the project, the HBase has been chosen as the large scale data store as it is the best choice in terms of elasticity (Konstantinou et al. 2012), and is one of the most successful, widely and even commercially used due to its robustness and already considerable maturity.

The approach described in the previous Section is general and could be applied to any large scale data store, particularly DataDroplets, taking advantage of its enriched data model and processing primitives. However, as explained in Section 4.1, we assume that transaction management is performed by an external layer providing snapshot isolation, and the underlying large scale data store provides the abstraction of a transactional tuple store. The transactional tuple store should expose primitive tuple set, get and scan operations that enforce snapshot isolation semantics. In the context of the CumuloNimbo project, two alternative implementations (as further explained) where developed on HBase. Implementing this for DataDroplets would require additional research outside the focus of this thesis. Thus, the presented prototype is built on Apache Derby[2], and uses HBase[3] as the large scale data store.

This section starts with an overview of HBase, Derby and CumuloNimbo's architecture. Then, the architecture of the prototype is presented, and a description of the main design decisions involved in mapping the relational data model and Derby's data structures, to the HBase. It also describes how performance is improved by reducing data transfer over the network, and how it fulfills both CumuloNimbo's stacks.

4.4.1 HBase overview

HBase is a key-value based distributed data storage system based on Bigtable (Chang et al. 2006). Thus, HBase's data model is similar to Bigtable's, Equation 2.2.

In HBase, data is stored in the form of HBase tables (HTable) that are multi-dimensional sorted maps. The index of the map is the row's key, column's name, and a timestamp. Columns are grouped into column families. Column families must

[1]`http://www.cumulonimbo.eu/`
[2]`http://db.apache.org/derby/`
[3]`http://hbase.apache.org`

be created before data can be stored under any column key in that family. Data is maintained in lexicographic order by row key. Finally, each column can have multiple versions of the same data indexed by their timestamp.

A read or write operation is performed on a row using the row-key and one or more column-keys. Update operations on a single row are atomic, that is, concurrent writes on a single row are serialized. Any update performed is immediately visible to any subsequent reads. HBase exports a non-blocking key-value interface on the data: put, get, delete, and scan operations.

HBase closely matches the scale-out properties assumed, as HTables are horizontally partitioned in regions. In turn, regions are assigned to RegionServers, and each region is stored as an appendable file in the distributed file system, Hadoop File System (HDFS) (Shvachko et al. 2010) based on GFS (Ghemawat et al. 2003).

4.4.2 Derby overview

Apache Derby is an open source relational database implemented entirely in Java and available under the Apache License, Version 2.0. Besides providing a complete implementation of SQL and JDBC, Derby has the advantage of already providing an embedded mode.

The storage management layer of Derby is split into two main layers, access and raw. The access layer presents a conglomerate (table or index)/row-based interface to the SQL layer. It handles table scans, index scans, index lookups, indexing, sorting, locking policies, transactions, isolation levels. The access layer sits on the raw store, which provides the raw storage of rows in pages in files, transaction logging, transaction management.

Following the architecture proposed in the previous chapter, the raw store was removed in our prototype and some components of the access layer were replaced.

4.4.3 CumuloNimbo's architecture

CumuloNimbo targets to obtain a highly scalable transactional platform as a service (PaaS). One of its innovations will be attaining scalability without trading off consistency as it is the norm in today's PaaS.

CumuloNimbo's approach lies in deconstructing transactional processing at fine granularity components, and scaling each of these components in an independent but

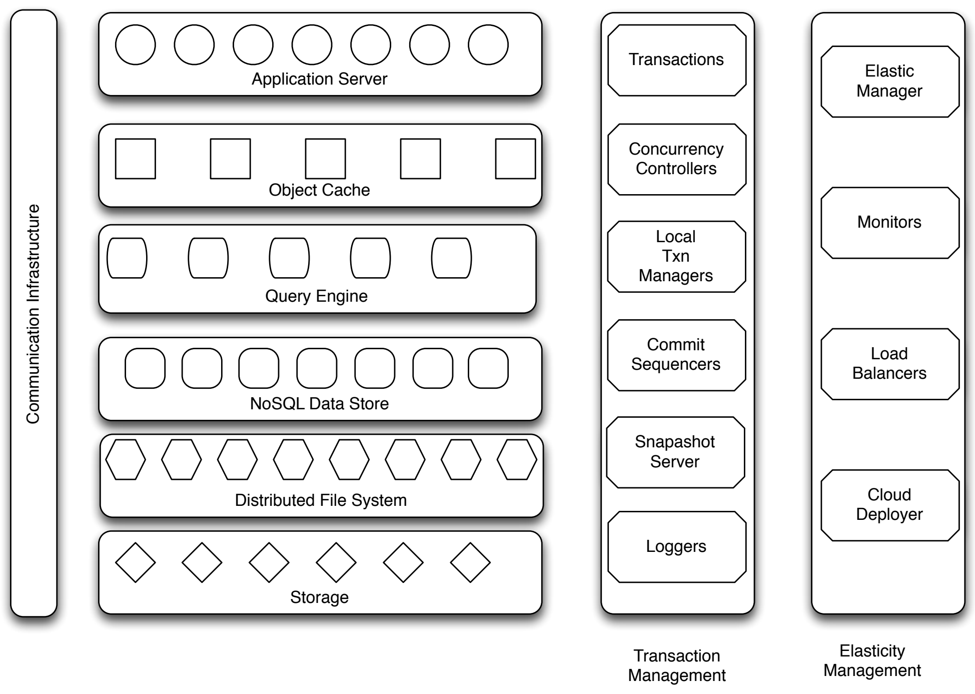


Figure 4.3: Scalable PaaS Architecture

composable manner. The architecture consists of several tiers:

- Application server: This tier contains both a component container and a persistent manager, such a Java EE container and a JPA manager, respectively.
- Object cache: This is a distributed object cache to improve the performance of the application server.
- Query engine: This tier consists of a number of query engines able to process SQL.
- Large scale data store: This tier caches file blocks to improve the performance of the query engine.
- Distributed file system: This is the file system where tables are stored.
- Storage: The storage subsystem.
- Communication: The communication subsystem.

- Transaction management: Based on an innovative transactional processing protocol, orchestrates transactional management at holistic level in a highly scalable way.
- Elasticity management: Monitors each server of each tier, and balances the load across server within the same tier, and reconfigures the number of servers in the tiers to minimize the resources used to process the incoming load.

The CumuloNimbo has two alternative implementations of the architecture, as follows:

Holistic stack

In the first stack, CumuloNimbo holistically manages transactions across its tiers by employing a cross-cutting implementation of snapshot isolation (Jimenez-Peris and Patiño-Martinez Filled 2011). That is, the concurrency controller avoids concurrent updates on the same objects, the distributed commit sequencer provides a global ordering for update transactions, and the distributed logger supports atomicity in a scalable way by exploiting the high throughput storage infrastructure. Thus, it is sufficient for the SI Light implementation to only provide a partial set of transactional guarantees to the upper layers. The HBase Client is extended to have a limited form of snapshot isolation by exploiting the multi-versioning support in HBase, without concerning with conflict detection, logging, and failure recovery. Timestamp generation, both for start and commit timestamps, is done at the commit sequencer, and atomicity is guaranteed by the distributed logger and holistic recovery mechanism.

HBase transaction manager stack

The second stack fully implements the transactional logic at the large scale data store layer, ReTSO (Junqueira et al. 2011). It implements comprehensive transaction management that is capable of operating in conjunction with the holistic stack, as well as operating independently below the distributed query engine when the holistic components are not employed. ReTSO is able to guarantee full atomicity and durability for committed transactions. An open source implementation of this transactional support for HBase, which enforces Snapshot Isolation semantics, is available an open source project[4].

[4]Omid - `https://github.com/yahoo/omid`

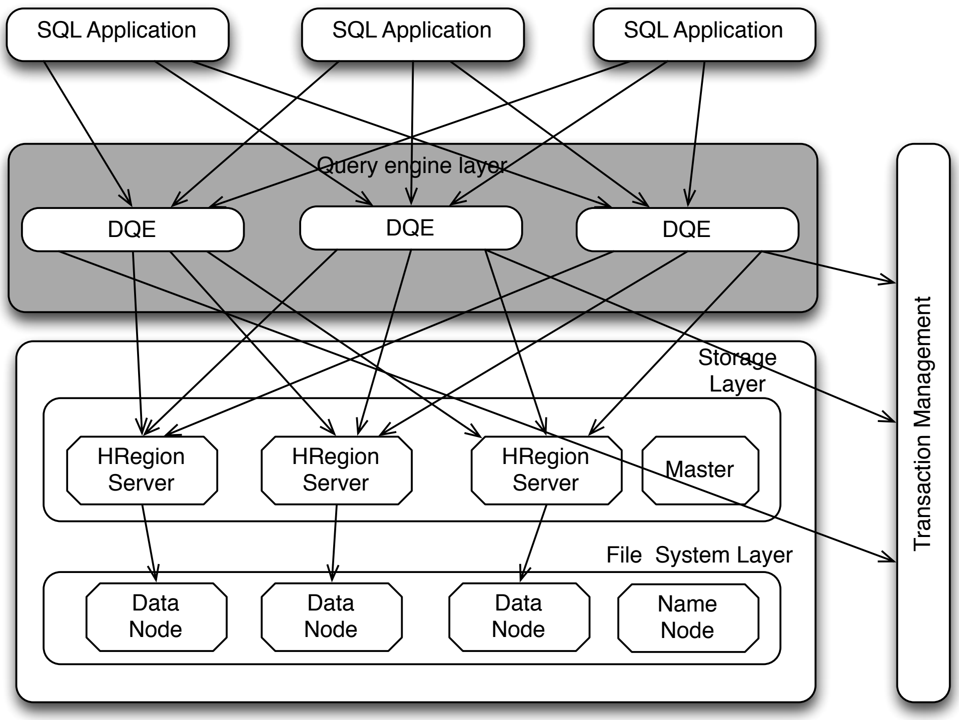


Figure 4.4: Architecture prototype

4.4.4 Prototype architecture

We now detail the query engine's architecture. The query engine's architecture builds on traditional relational database management. However, instead of providing an embedded storage engine, it interfaces with HBase for durability and basic indexing, and with transaction management for snapshot isolation. A key feature of this architecture is that no additional coordination is required for data manipulation statements; thus, allowing the query engine to scale out in a distributed fashion.

The system is composed of the following layers: (i) query engine, (ii) storage, and (iii) file system. Clients issue SQL transactional requests to any of the instances of the query engine's layer. Those instances can either be embedded in the application or be dedicated server nodes. A query engine node communicates both with storage nodes and the transaction management, executing queries and delivering results to applications. The storage layer supports multiple versions of data and exports a transactional non-blocking key-value interface on the data: put, get, delete, and scan operations that enforce Snapshot Isolation's semantics.

We mostly reuse the query processing sub-system of Derby. However, the storage management sub-system is replaced to be able to operate on HBase. For query processing, Derby's compiler and optimizer were reused. Two new operators for index and sequential data scans have been added to the set of Apache Derby's generic relational operators. These operators leverage HBase's indexing and filtering capabilities to minimize the amount of data that needs to be fetched. The query engine translates the user queries into some appropriate put, get, delete, and scan operations to be invoked on HBase. The SQL advanced operators, such as joins and aggregations, are not supported by HBase and are implemented at the query engine.

When a query engine receives an SQL *begin transaction*, it sends a request to the transaction management to receive the transaction start timestamp, which will be included in all future requests sent from this transaction to the storage layer. When the query engine receives the SQL *commit/rollback* request, it sends a commit/abort request to the transaction manager.

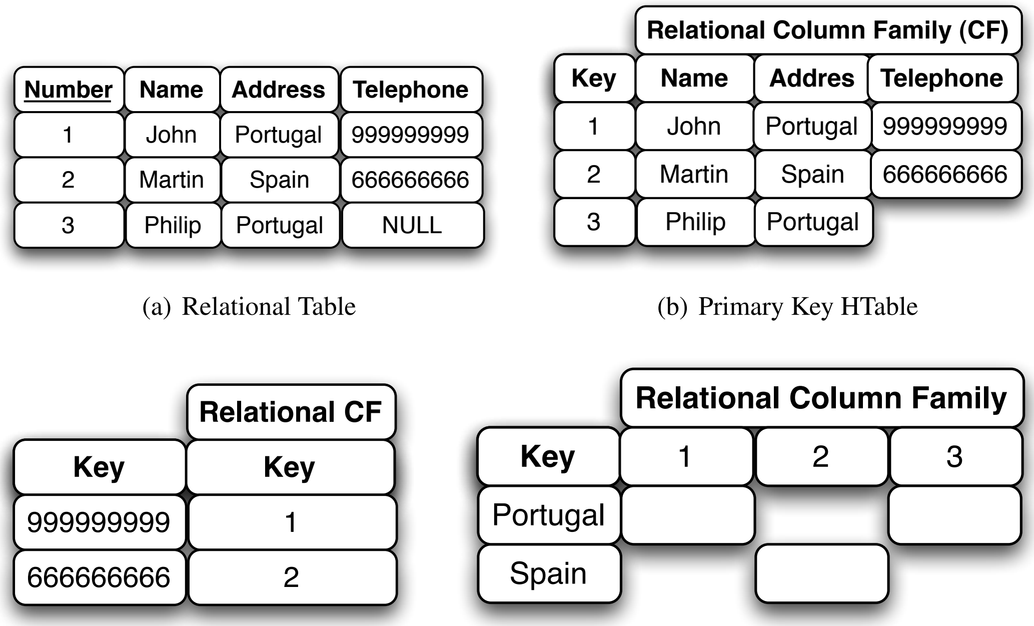
The query engine processes ANSI SQL, and is fully compatible with JDBC and ODBC client interfaces. Therefore, it supports the development of applications both on application servers and applications built on SQL databases by directly offering an SQL interface.

4.4.5 Relational-tuple store mapping

Relational tables and secondary indexes are mapped to the column-oriented data model of HBase, the tuple store being used. Briefly, the data model of HBase can be described as follows: Each HTable in HBase is a multi-dimensional sorted map.

We adopted a simple mapping from a relational table to an HTable. There is a one-to-one mapping where the HBase row's key is the relational primary key (simple or compound), and all relational columns are mapped into a single column family. Since relational columns are not multi-valued, each relational column is mapped to a HTable column. The schema of relational tables is rigid, that is, every row in the same table must have the same set of columns. However, the value for some relational columns can be NULL; thus, a HTable column for a given row only exists if its original relational column for that row is not NULL.

Furthermore, each secondary index (in the relational model) is mapped into an additional HTable. The additional table is necessary so that data is ordered by the indexed attributes. For each indexed attribute, an HTable row is added and its row's key is the indexed attribute. For unique indexes, the row has a single column with



<u>Number</u>	Name	Address	Telephone
1	John	Portugal	999999999
2	Martin	Spain	666666666
3	Philip	Portugal	NULL

(a) Relational Table

	Relational Column Family (CF)		
Key	Name	Addres	Telephone
1	John	Portugal	999999999
2	Martin	Spain	666666666
3	Philip	Portugal	

(b) Primary Key HTable

	Relational CF
Key	Key
999999999	1
666666666	2

(c) Unique Index HTable

	Relational Column Family		
Key	1	2	3
Portugal	□		□
Spain		□	

(d) Non-unique Index HTable

Figure 4.5: Data model mapping

its value being the key of the matching indexed row in the primary key table. For non-unique indexes, there is one column per matching indexed row with the name of the column being the matching row's key. Figure 4.5(a) depicts an example relational table. The column Number is the primary key, and the table has two additional indexes: one unique index on column Telephone, and a non-unique index on column Address. Therefore, the mapping will have three HTables: base data, Figure 4.5(b), unique index on column Telephone, Figure 4.5(c), and non-unique index on column Address, Figure 4.5(d).

4.4.6 Optimizing data transfer

To reduce network traffic between the query engine and HBase, the implementation of sequential and index scan operators takes advantage of the indexing and filtering capabilities in the HBase data store.

For index scans, we need to maintain data ordered by one or more columns. This allows restricting the desired rows for a given scan by optionally specifying the start and the stop keys. In a relational table each column is typed and data is ordered according to the natural order of the indexed column data type. However, row keys in HBase are plain byte arrays and neither Derby or HBase byte encoding preserve the data type's natural order. In order to build and store indexes in HBase maintaining the data type's order we had implemented proper encoding for integer, decimal, char, varchar and date types. As indexes may be composite, besides each specific data type encoding, we also need to define a way to encode multiple indexed columns in the same byte array. We do so by simply concatenating them from left to right, according to the order they are defined in the index using a pre-defined separator.

In HBase, the start and stop keys of a scan must always refer to all the columns defined in the index. However, when using compound indexes, the Query Engine may generate scans using subsets of the index columns. Indeed, an index scan can use equal conditions on any prefix of the indexed columns (from left to right) and at most one range condition on the rightmost queried column. In order to map these partial scans, the default start and stop keys in HBase are not used, but instead the scan expression is run through HBase's BinaryPrefixComparator filter.

The above mechanisms reduce the traffic between the query engine and HBase by only bringing the rows that match the range of the index scan. However, the scan can also select non-indexed columns. A naïve implementation of this selection would fetch all rows from the index scan and test the relevant columns row by row.

In detail, doing so on HBase would require a full table scan, which means fetching all the table rows from the different regions and possible different RegionServers. Therefore, the full table would be brought to the query engine instance, and only then discard those rows not selected by the filter. To mitigate this performance overhead, particularly for low selective queries that this approach may incur, the whole selection is pushed down into HBase. This is done by using the SingleColumnValueFilter filter to test a single column, and to combine them respecting the conjunctive normal form, using the FilterList filter. The latter represents an ordered list of filters that are evaluated with a specified boolean operator FilterList.Operator.MUST PASS ALL (AND) or FilterList.Operator.MUST PASS ONE (OR).

4.4.7 Scan costs

Regarding the cost model for selection of proper scan operators, we have defined a model that extends the existing model taking into account the execution time of the involved HBase operations for each type of scan operator. Briefly, table scans and

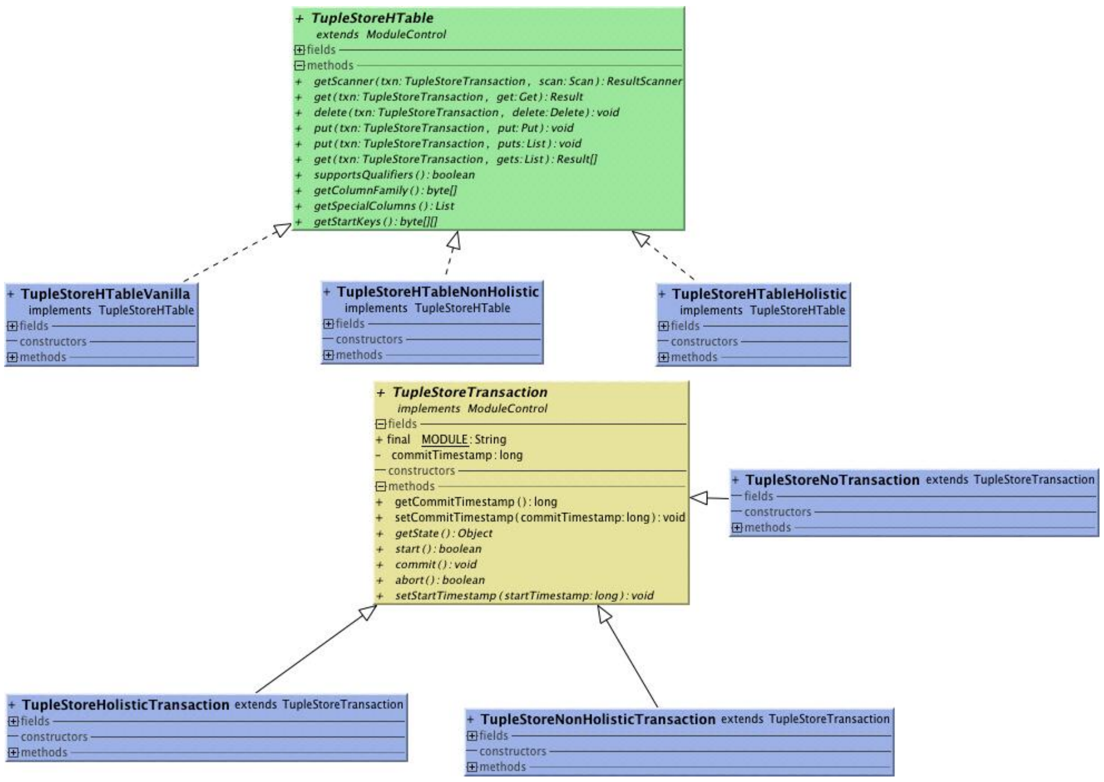

+ TupleStoreHTable
extends ModuleControl
fields
methods
+ getScanner(txn: TupleStoreTransaction, scan: Scan): ResultScanner
+ get(txn: TupleStoreTransaction, get: Get): Result
+ delete(txn: TupleStoreTransaction, delete: Delete): void
+ put(txn: TupleStoreTransaction, put: Put): void
+ put(txn: TupleStoreTransaction, puts: List): void
+ get(txn: TupleStoreTransaction, gets: List): Result[]
+ supportsQualifiers(): boolean
+ getColumnFamily(): byte[]
+ getSpecialColumns(): List
+ getStartKeys(): byte[][]
+ TupleStoreHTableVanilla
implements TupleStoreHTable
fields
constructors
methods
+ TupleStoreHTableNonHolistic
implements TupleStoreHTable
fields
constructors
methods
+ TupleStoreHTableHolistic
implements TupleStoreHTable
fields
constructors
methods
+ TupleStoreTransaction
implements ModuleControl
fields
+ final MODULE: String
- commitTimestamp: long
constructors
methods
+ getCommitTimestamp(): long
+ setCommitTimestamp(commitTimestamp: long): void
+ getState(): Object
+ start(): boolean
+ commit(): void
+ abort(): boolean
+ setStartTimestamp(startTimestamp: long): void
+ TupleStoreNoTransaction extends TupleStoreTransaction
fields
constructors
methods
+ TupleStoreHolisticTransaction extends TupleStoreTransaction
fields
constructors
methods
+ TupleStoreNonHolisticTransaction extends TupleStoreTransaction
fields
constructors
methods

Figure 4.6: Multi-architecture interface

primary key index scans in DQE involve a HBase Scan operation; secondary indexes involve a HBase Scan on the index HTable, and possibly a further Get (done in batchs) per index row to retrieve additional data from the base table. Moreover, we have measured the impact of the scan size and number of selected, and projected columns in the Scan execution time. These costs have been added to the model using a calibration database (Gardarin et al. 1996).

4.4.8 Support for multiple architectures

The prototype can be configured to run with both CumuloNimbo's stacks, and additionally using standard HBase for debug and performance comparison. As a result, generic interfaces to the transactional HBase (`TupleStoreHTable`) and the transaction manager (`TupleStoreTransaction`) were defined, as shown in Figure 4.6.

Each query engine instance preserves transactional contexts and consistency through the interface to the CumuloNimbo's transactional framework. The implementation of the interface depends on the considered stack, it can be a thin adapter to HBase

(Holistic), a full-fledged transactional logic component implemented in HBase (HBase Transaction Manager), or vanilla HBase.

When running the HBase standard stack, the `TupleStoreNoTransaction` is used and as standard HBase has no transactional context it does nothing, and the `TupleStoreHTableVanilla` ignores the transaction context.

Regarding the CumuloNimbo's stacks, the behavior of each stack is as follows. For the Holistic stack the class `TupleStoreHolisticTransaction` behaves as follows:

- On starting a transaction, builds a local transaction context by receiving the transaction start timestamp from the distributed application server layer, through the STARTTS(?) procedure;
- This context is provided to `TupleStoreHTableHolistic`, whenever it is used in the context of the same transaction;
- Before transaction termination, the distributed application server layer sets the commit timestamp using the COMMITTS(?);
- Upon transaction termination, it notifies the HBase thin client setting the commit timestamp.

On the other hand, for the HBase Transaction Manager stack, the class `TupleStoreNonHolisticTransaction` behaves as follows:

- On starting a transaction, the query engine receives from the HBase Transaction Manager the transaction state (start timestamp);
- This state is used whenever data is fetched from or written back to HBase, using the `TupleStoreHTableNonHolistic` class by pointing to the relevant versions of each row, using the transaction's start timestamp to read from the right snapshot;
- Upon transaction termination, a commit request with the transaction context is sent to the HBase Transaction Manager.

4.5 Evaluation

We did the evaluation of the prototype both with vanilla HBase and with one of the stacks of the CumuloNimbo with transactional guarantees. The former allows us to stress the distributed query engine layer, showing its overhead and scale-out properties on HBase, preserving its isolation semantics, row level atomicity. The latter, allows us to see the scale-out of the full stack for a highly scalable transactional platform.

4.5.1 Test workloads

Yahoo! Cloud Serving Benchmark, YCSB (Cooper et al. 2010), was designed to benchmark the performance of large scale data stores under different workloads. It has client implementations for several large scale data stores and a JDBC client for RDBMs. We have used the HBase and JDBC clients without further modifications.

Additionally, we run an industry standard on-line transaction processing SQL benchmark, TPC-C. It mimics a whole-sale supplier with a number of geographically distributed sales districts and associated warehouses. The warehouses are hotspots of the system, and the benchmark defines ten clients per warehouse.

TPC-C specifies five transactions: NewOrder with 44% of the occurrences; Payment with 44%; OrderStatus with 4%; Delivery with 4%; and StockLevel with 4%. The NewOrder, Payment and Delivery are update transactions, while the others are read-only. The traffic is a mixture of 8% read-only and 92% update transactions; therefore, it is a write intensive benchmark.

We have used two implementations for TPC-C: a SQL-based implementation and an implementation optimized for HBase.

The former is an existing SQL implementation [5], without modifications. The latter is an existing TPC-C implementation optimized for HBase[6]. Briefly, in the HBase implementation TPC-C columns are grouped into column families, named differently for optimization, and data storage layout has been optimized.

4.5.2 HBase

We measured the overhead of our proposal and its scale-out properties running on HBase.

[5]BenchmarkSQL - `http://sourceforge.net/projects/benchmarksql/`

[6]PyTPCC - `https://github.com/apavlo/py-tpcc/wiki/HBase-Driver`

Overhead

We measured the overhead of our proposal in terms of latency, compared to both standard HBase client and a traditional RDBMS, Derby, using a typical benchmark of large scale data stores, YCSB. Moreover, we evaluated the throughput of our proposal under the load of a traditional database workload, TPC-C, compared to traditional RDBMs in a single machine.

Experimental setting

The machines used for those experiments have 2.4 GHz Dual-Core AMD Opteron(tm) Processor, with 4GB memory and a local SATA disk.

For these experiments, two machines were used for the evaluation of our proposal: one to run the workload generator, either YCSB or TPC-C, using an embedded connection to the DQE; and another one running the database depending on the configuration, either HBase or standard Derby server. HBase was run in standalone mode, meaning that the HBase master and HBase RegionServer were collocated in the same machine using the local filesystem. Regarding standard Derby, we used the weaker isolation level, TRANSACTION_READ_UNCOMMITTED (ANSI level 0).

The YCSB database was populated with a hundred thousand rows (185MB) and the workload consists of one million operations. The proportion for the different types of operations was read=0.6, update=0.2 and scan=0.2. The operations are distributed uniformly over database rows. The size of scan operator was also a uniform random number between one and ten. Each client has one or fifty threads and a target throughput of a hundred operations per second.

The TPC-C database was populated with ten warehouses resulting in a database with 5GB, and the number of client threads varied from one to a hundred.

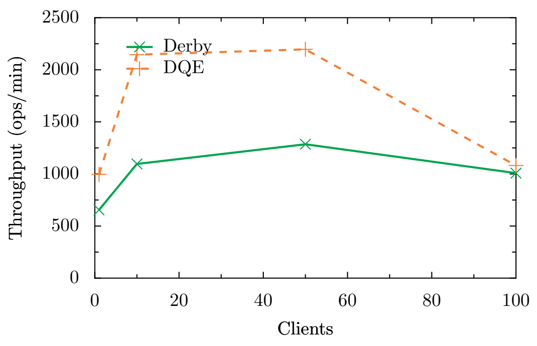
Results

The overhead in terms of average latency (in milliseconds) for the YCSB workload is shown in Table 4.1. We compare the overhead of our proposal, DQE, against the standard HBase client, and also with a traditional centralized SQL RDBMS, Derby.

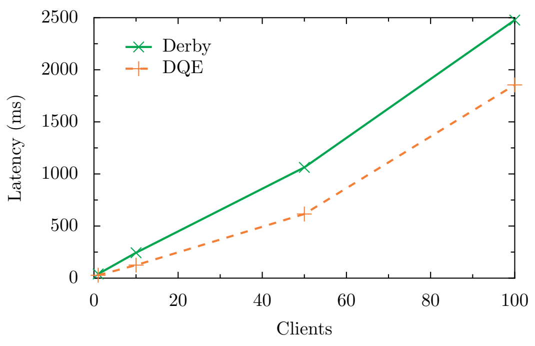
The results show that for all types of operations the query engine has a minimal overhead compared to the direct usage of HBase. The additional overhead is due to SQL processing and additional marshaling/unmarshaling. The results also show that the DQE has similar latencies of a traditional centralized RDBMS.

Table 4.1: Overhead results (ms)

Workload	1 c/ 100 tps			50 c/ 100 tps		
Operation	Derby	HBase	DQE	Derby	HBase	DQE
Insert	1.45	0.58	0.93	23	1.04	1.98
Update	0.92	0.51	1.3	1.39	2.66	3.1
Read	0.53	0.53	0.79	0.97	1.63	1.7
Scan	3.3	1.43	2.9	4.48	4.64	6.1



(a) Throughput



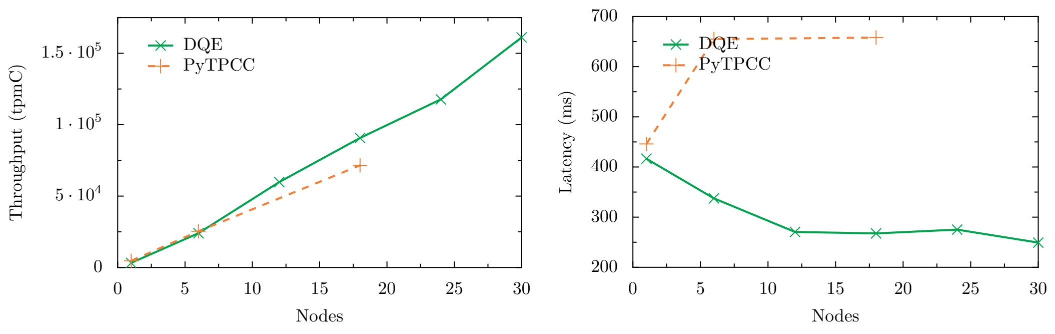
(b) Latency

Figure 4.7: TPC-C single machine results

We measured the overhead of our proposal with a typical benchmark of large scale data stores and restricted to simple operations, and with a traditional benchmark of relational databases (TPC-C) with complex operations. The throughput and latency for DQE, and standard Derby for the TPC-C benchmark are depicted in Figure 4.7.

The throughput results, Figure 4.7(a), show that our proposal has higher throughput than a traditional RDBMS even with a single client. Moreover, the peak throughput with a single database instance is also higher with our proposal. However, while the traditional RDBMS is offering transactional guarantees, in this setting DQE is using default HBase, preserving the isolation semantics offered by HBase. Oppositely to traditional RDBMS, our solution scaled out by allowing to add both additional storage nodes and query engine instances, as further shown.

The latency results, Figure 4.7(b), corroborate the throughput results, DQE has lower latencies than Derby.



(a) Throughput (b) Latency

Figure 4.8: TPC-C scaling out results

Scale out

We evaluated the scalability of our proposal in terms of achieving increased throughput by scaling out the system from a cluster with a single RegionServer to thirty.

Experimental setting

We ran the experiments on a cluster of forty-two machines with 3.10GHz GHz Quad-Core i3-2100 CPU, with 4GB memory and a local SATA disk.

The TPC-C workload has run from a varying number of machines. For our proposal, we varied the number of client machines from one to ten, each running a hundred and fifty client threads. Each client machine also ran an DQE instance in embedded mode.

One machine was used to run the HDFS namenode, HBase Master and Zookeper (Hunt et al. 2010). The remaining machines are RegionServers, each configured with a heap of 3GB, and also running a HDFS datanode instance.

The TPC-C database was populated according to the number of RegionServers, ranging from five warehouses for a single RegionServer to a hundred and fifty warehouses for thirty RegionServers. All TPC-C tables were partitioned and distributed, so there were five warehouses per RegionServer each handling a total of fifty clients.

Results

We evaluated the scalability in terms of achieving increased throughput by scaling out the system from a cluster with a single RegionServer to thirty RegionServers.

The throughput under the scaling-out of the system with one, six, twelve, eighteen, twenty-four, and thirty RegionServers is depicted in Figure 4.8(a). The results show that our proposal presents linear scalability. This is mainly due to the scale independence of the query processing layer and the large scale data store layer. Moreover, as previously shown, the query processing layer has a low overhead and has the advantage of being stateless.

Furthermore, while our proposal, using an existing SQL implementation, has a slightly lower throughput for one and six RegionServers than the implementation specifically developed and optimized for HBase, it scales better.

This can be mainly attributed to the optimizations achieved by the distributed query engine that take advantage of relational operators, filtering and secondary indexes, while manual optimizations and de-normalization still incur on greater complexity resulting in a greater overhead, and affecting the desired scalability.

In this specific case the distributed query engine can greatly restrict the amount of data retrieved from HBase by also taking advantage of HBase Filters to select non-indexed columns, as previously explained. As a matter of fact, the network traffic when using our proposal is much lower than using the TPC-C implementation for HBase. This prevents us from getting results for this implementation with more than eighteen RegionServers, because network was saturated.

The result for latency, Figure 4.8(b), confirms these statements.

4.5.3 HBase transaction manager

We also measured the scalability of our proposal with transactional guarantees using the CumuloNimbo HBase Transaction Manager's stack compared to traditional RDBMs, MySQL. We evaluated our prototype in two settings. The first measured the overhead of our prototype in terms of latency, and the second measured its scalability with the performance metric being the throughput.

For these tests we used a modified version of YCSB. The vanilla implementation operates on single rows. To benchmark the performance of multirow transactions, we modified YCSB to add a multirow operation.

Experimental setting

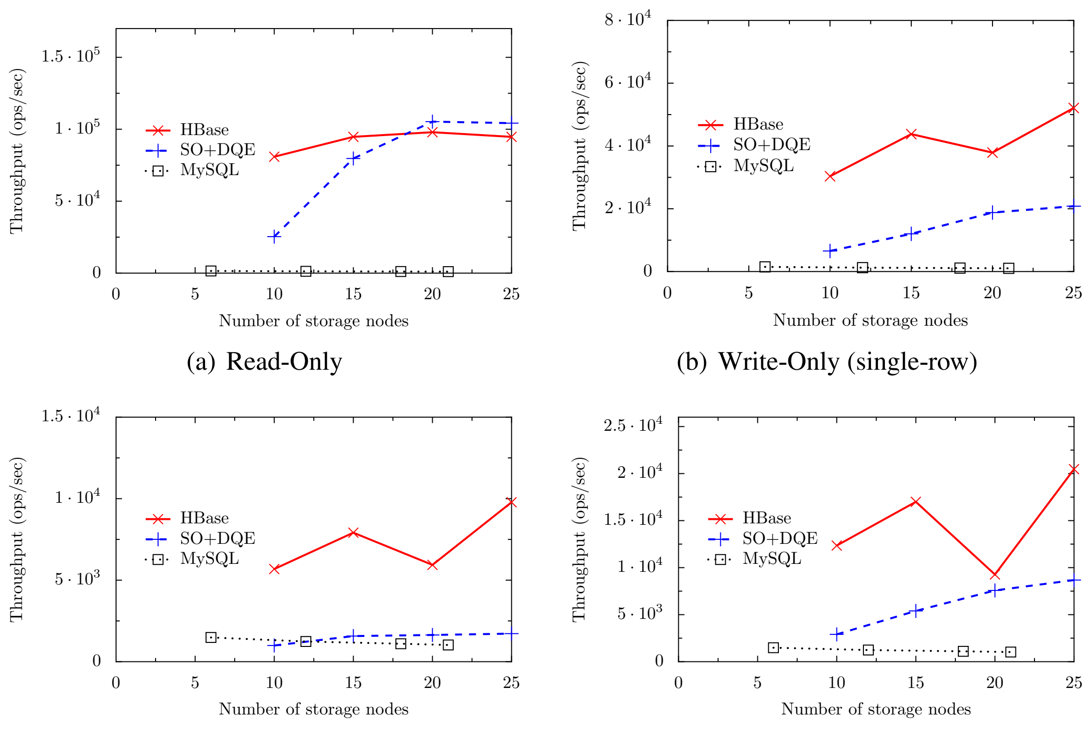
We ran the experiments on a cluster of thirty-six machines, six of which are used to run the HDFS namenode, HBase Master, the SO server, a Zookeeper ensemble and a Bookkeeper ensemble, which implements a WAL replicated on multiple remote servers (Zookeeper (Hunt et al. 2010) and Bookkeeper ensembles are collocated on the same three machines). Bookkeeper [7] is used for durability of transaction statuses, it maintains a WAL in multiple remote nodes, for recovery. The YCSB workload is run from five machines, each running a hundred client threads. We use the remaining machines as RegionServers, each configured with a heap of 16GB. Each machine has two Xeon Quad-Core 2.40Ghz processors, 24GB of memory, gigabit Ethernet and four SATA hard disks. MySQL is configured with a single management node and a single MySQL daemon. The number of threads per client is set to ten, as higher would trigger the daemon to stop accepting new connections. We use NDBCluster storage back-end over a multiples of three machines. Multiples of 3 have to be used as we specify a replication level of three to match the level of redundancy offered by HDFS. On each NDBNode, Data and Index memory were both set to 4GB. We set the number of log segments to two-hundred so that the high rate of requests would not overwhelm the write-ahead log.

Results

Here, we evaluate how our proposal scales with the number of RegionServers in a database of twenty million rows. We tested four workloads: read-only, single-row write, multi-row write, and mixed, which consisted of 30% read, 40% multi-row write, and 30% scan. Rows were selected at random using a uniform distribution. The size of both scan and multi-row write is a uniform random number between one and ten.

The results are depicted in Figure 4.9. Because of larger delays for reads in our system, with low number of machines, our performance is lower than that of HBase (Figure 4.9(a)). With the increase in number of RegionServers, the extra load distributes among them and our system's performance approaches that of HBase. After a certain point, the network or the HDFS saturates and the performance levels off. The performance with write-only workload is in general less than with read-only workload, since its frequent changes into the regions leads to more frequent calls for *compaction*, which is an expensive operation in HBase. RegionServers batch the writes from all the threads into one request to the WAL; therefore, impose much less overhead on the network and the WAL. Since writing into HDFS, which here plays the role of a WAL for RegionServers, is not the bottleneck, the major overhead in write-only traffic is the

[7] `http://zookeeper.apache.org/bookkeeper`



(a) Read-Only (b) Write-Only (single-row)

(c) Write-Only (multi-row) (d) Mixed

Figure 4.9: Machine scaling results

message processing cost at RegionServers. With more RegionServers, this overhead is split among more nodes; therefore, each RegionServer could service more write requests. The write operations in HBase (and accordingly to our system) still scale well with the number of RegionServers. The system is, therefore, not saturated even with twenty-five RegionServers. The same feature also alleviates the overhead of the additional write into the *PendingWrite* column. The throughput of our system is still less than HBase, since we have to write into HBase twice: once with the start timestamp, and once with the commit timestamp obtained after the commit. The same behavior of write operations in the mixed workload makes it scale better. The Omid+DQE, with full SQL transactional support, scales as well as HBase, while maintaining good throughput.

In all the experiments, it outperforms MySQL. This difference becomes more significant when the proportion of reads in the workload is bigger.

4.5.4 Summary of results

The proposed approach provides a SQL query engine for a large scale data store, including the standard JDBC client interface. The result is a query engine that can be either embedded in the application as a middleware layer or run as dedicated server instances. In both cases, there is no need for central components or distributed coordination; thus, it is able to scale out.

The feasibility of the approach is demonstrated by the performance results obtained with YCSB and TPC-C. Moreover, the comparison with a TPC-C implementation optimized for HBase shows that by simply using the distributed query engine, SQL applications can be easily run and even achieve better results than with manual optimizations.

Furthermore, the evaluation of our proposal, in the context of the CumuloNimbo's architecture with a transaction manager enforcing Snapshot Isolation, shows the system using DQE for query processing scales as well as HBase, and outperforms MySQL.

4.6 Discussion

Recently, several proposals for middleware that expose higher-level query interfaces on the barebones key-value primitives have appear. Many of these aim at approximating the traditional SQL abstraction, ranging from shallow SQL-like syntax for simple key-value queries to translation of analytical queries into map-reduce jobs. However, they only offer a subset of the processing facilities offered by SQL, and define new query languages similar to SQL. This implies that applications must be written specifically for them using their data model, and some queries must be written in application code. This is due to to the lack of support for complex processing as general joins are not offered by those systems.

In this chapter, we propose a solution for the migration of existing SQL application code by providing a complete implementation of SQL on large scale data stores with a standard JDBC client interface, which can be plugged in existing applications and middleware (for example, object-relation mappers). The solution builds on query engine's components from traditional RDBMs.

Chapter 5

Conclusions

With cloud-based large scale data stores in Platform-as-a-Service offerings, such as Google Data Store API in Google App Engine and Amazon's DynamoDB, and open source packages, such as HBase and Cassandra, large scale data stores become attractive for a wider spectrum of applications. However, they are in strong contrast with traditional relational databases, presenting very simple data models and APIs. They lack most of the established relational data management operations, and relax consistency guarantees, providing eventual consistency.

Given the prevalence of traditional relational databases, Web-scale applications can highly benefit from some of their features in large scale data stores without jeopardizing scalability. Without them it is hard to provide a smooth migration, and this is a hurdle to the adoption of large scale data stores by a wider potential market.

Our proposal resides in exploring two complementary approaches to provide additional consistency guarantees and higher-level data processing primitives in large scale data stores: extending data stores with additional operations, such as general multi-item operations, and coupling data stores with existent processing facilities.

One one hand, we start by proposing a new architecture for large scale data stores that allows to find the right trade-offs among flexible usage, efficiency, fault tolerance and quality of service, through a clear separation of concerns between different functional aspects of the system. Then, taking into account that existing large scale data stores do not consider general multi-item operations, we devised a new multi-tuple data placement strategy, which allows to efficiently store and retrieve large sets of related data at once. Multi-tuple operations leverage disclosed data relations to manipulate sets of comparable or arbitrarily related elements. Evaluation results show the benefits in: request latency of DataDroplets' enhanced data model and API; the minimal cost of synchronous replication; attesting the scalability of DataDroplets; and a substantial improvement in overall query performance when using the novel data placement strategy. Additionally, we show the usefulness of having multiple simultaneous placement strategies in a multi-tenant system by also supporting ordered placement for range queries, and the usual random placement. Moreover, the new replica placement strategy that combines different replica placement strategies presents the lowest latency.

On the other hand, we provide a SQL query engine for a large scale data store, including the standard JDBC client interface. The result is a query engine that can be embedded in the application as a middleware layer, without the need for central components or distributed coordination; thus, it does not impact the ability to scale out. Evaluation results show the minimal overhead of the query engine and the ability to scale out SQL based applications.

All the contributions have been implemented and evaluated with realistic benchmarks.

5.1 Future work

In extension to the work presented in this thesis, we believe that several points are worth further research.

Clouder persistent-state layer This thesis's focus was on the major problems of the soft-state layer of Clouder. The work on the underlying persistent-state layer is future work. As previously said, the main design ideas for the persistent-state layer can be found in a position paper (Matos et al. 2011).

The first issue to be researched is how to rely on an epidemic dissemination protocol to spread data and operations to relevant nodes, taking advantage of the inherent scalability and ability to mask transient node and link failures.

Multi-master replication in DataDroplets Currently, DataDroplets has a simple configurable primary-backup replication protocol that can work from asynchronous mode to synchronous mode. Recent work has also proposed mechanisms to provide strong consistency in key-value stores (Glendenning et al. 2011; Lloyd et al. 2011).

It may be interesting to explore multi-master replication used in traditional RDBMS, in the context of large scale data stores that may be adaptive, with the number of replicas defined in a dynamic fashion, according to the current load.

Formal analysis A recent article presents a mathematical data model for the most common large scale data stores, and shows that their data model is the mathematical dual of SQL's relational data model of foreign-/primary-key relationships (Meijer and Bierman 2011). It may be interesting to exploit this relation between the SQL and large scale data stores for a formal analysis of DQE, the couple of a SQL processor on a large scale data store.

Moreover, we think that it may be interesting to explore systems similar to the one described in Zellag and Kemme (Zellag and Kemme 2011). These systems quantify and classify consistency anomalies in multi-tier architectures to exploit the overall consistency given by the CumuloNimbo's architecture, Figure 4.3, which deconstructs transactional processing at fine granularity components.

Aggregates and pre-computed views For queries with complex joins and aggregates, the current version of DQE requires the bulk of data to be fetched from the large scale data store to the query engine, and most of it to be later discarded after the processing.

It may be interesting to explore how to further extend large scale data stores by pushing some traditional relational operators and optimizations into it. One approach can be through the maintenance of materialized views in large scale data stores, when requested by the query engine.

This can be particularly useful for applications that depend on the continuous processing of dynamic data for the prompt answer to user queries, such as the computation of network trends in social networks.

Snapshot Isolation versioning DQE assumes that the underlying large scale data store provides the abstraction of a transactional tuple store with primitive tuple set, get and scan operations that enforce Snapshot Isolation semantics. The evaluation presented in this thesis uses an implementation on HBase that takes advantage of its multiple versions. However, not all large scale data stores support multiple versions. It may be interesting to explore how to offer a lightweight transactional layer on any large scale data stores.

Large scale data stores' elasticity Large scale data stores were designed to take advantage of large resource pools, and provide high availability and high performance levels. Moreover, they were designed to be able to cope with resource availability changes. It is possible, for instance, to add or remove database nodes from the system, and the database will handle such change. However, they are not elastic even though they can handle elasticity: an external entity is required to control when and how to add and remove nodes. Some recent research work (Lim et al. 2010; Trushkowsky et al. 2011; Konstantinou et al. 2011) is striving for elasticity in large scale data stores. The approach taken is to gather system information and add or remove nodes from the system in order to adjust it to the demands.

Nevertheless, simply adding and removing nodes is insufficient as different workloads have different access patterns that may change over time.

Bibliography

Michael Armbrust, Nick Lanham, Stephen Tu, Armando Fox, Michael J. Franklin, and David A. Patterson. The case for PIQL: a performance insightful query language. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 131–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807149. URL `http://doi.acm.org/10.1145/1807128.1807149`. - **Cited** on page 38.

Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. PIQL: success-tolerant query processing in the cloud. *Proc. VLDB Endow.*, 5(3):181–192, November 2011. - **Cited** on page 88.

Jason Baker, Chris Bondç, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean M. L´eon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011. - **Cited** on page 38.

A.-L. Barabási and E. Bonabeau. Scale-free networks. *Scientific American*, 288:60–69, 2003. - **Cited** on page 76.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995. ISSN 0163-5808. doi: 10.1145/568271.223785. URL `http://doi.acm.org/10.1145/568271.223785`. - **Cited** on page 88.

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5. - **Cited** on page 3.

Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Computer Communication*

Review, 34(4):353–366, 2004. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/1030194.1015507. - **Cited** on pages 19 and 21.

Kenneth Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems.*, 17(2):41–88, 1999. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/312203.312207. - **Cited** on page 46.

Danah Boyd, Scott Golder, and Gilad Lotan. Tweet tweet retweet: Conversational aspects of retweeting on twitter. In IEEE Computer Society, editor, *Proceedings of HICSS-43*, January 2010. - **Cited** on pages 75 and 76.

Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: http://doi.acm.org/10.1145/343477.343502. - **Cited** on page 3.

A. R. Butz. Alternative algorithm for hilbert's space-filling curve. *IEEE Trans. Comput.*, 20(4):424–426, 1971. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/T-C.1971.223258. - **Cited** on page 62.

J.C.S. Cardoso, C. Baquero, and P.S. Almeida. Probabilistic estimation of network size and diameter. In *Dependable Computing, 2009. LADC '09. Fourth Latin-American Symposium on*, pages 33 –40, 2009. doi: 10.1109/LADC.2009.19. - **Cited** on page 46.

Nuno Carvalho, Jose Pereira, Rui Oliveira, and Luis Rodrigues. Emergent structure in unstructured epidemic multicast. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 481–490, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2855-4. doi: http://dx.doi.org/10.1109/DSN.2007.40. - **Cited** on page 43.

Miguel Castro, Manuel Costa, and Antony Rowstron. Should we build gnutella on a structured overlay? *SIGCOMM Comput. Commun. Rev.*, 34:131–136, January 2004. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/972374.972397. URL `http://doi.acm.org/10.1145/972374.972397`. - **Cited** on page 10.

R. Chand and P. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par'05)*, August 2005. - **Cited** on page 47.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI'06*, 2006. - **Cited** on pages 2, 11, 25, 60 and 95.

Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered DHT applications. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 97–108, New York, NY, USA, 2005. ACM. ISBN 1-59593-009-4. doi: http://doi.acm.org/10.1145/1080091.1080104. - **Cited** on pages 19 and 20.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362384.362685. URL `http://doi.acm.org/10.1145/362384.362685`. - **Cited** on page 35.

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. *Proc. VLDB Endow.*, 2008. - **Cited** on pages 2, 11, 24 and 60.

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 '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL `http://doi.acm.org/10.1145/1807128.1807152`. - **Cited** on page 105.

Nuno Cruces, Rodrigo Rodrigues, and Paulo Ferreira. Pastel: Bridging the Gap between Structured and Large-State Overlays. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, pages 49–57, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3156-4. doi: 10.1109/CCGRID.2008.72. URL `http://dx.doi.org/10.1109/CCGRID.2008.72`. - **Cited** on page 17.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `http://doi.acm.org/10.1145/1327452.1327492`. - **Cited** on pages 3, 43 and 48.

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 twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294281. URL `http://doi.acm.org/10.1145/1294261.1294281`. - **Cited** on pages 2, 11, 17, 24, 42 and 60.

Patrick Eugster, Rachid Guerraoui, Sidath Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, 21(4):341–374, 2003. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/945506.945507. - **Cited** on page 46.

W. Galuba, K. Aberer, Z. Despotovic, and W. Kellerer. Protopeer: From simulation to live deployment in one step. In *Peer-to-Peer Computing , 2008. P2P '08. Eighth International Conference on*, pages 191–192, Sept. 2008. doi: 10.1109/P2P.2008.13. - **Cited** on page 64.

Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 19–24, New York, NY, USA, 2004. ACM. doi: http://doi.acm.org/10.1145/1017074.1017081. - **Cited** on pages 19 and 20.

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book.* 2008. - **Cited** on page 1.

Georges Gardarin, Fei Sha, and Zhao-Hui Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *Proceedings* *of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 378–389, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1-55860-382-4. URL `http://dl.acm.org/citation.cfm?id=645922.673485`. - **Cited** on page 103.

Anil K. Garg and C. C. Gotlieb. Order-preserving key transformations. *ACM Trans. Database Syst.*, 11(2):213–234, 1986. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/5922.5923. - **Cited** on pages 19 and 60.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Operating Systems Review*, 37(5):29–43, 2003. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1165389.945450. - **Cited** on pages 31 and 96.

Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, DBISP2P'05/06, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71660-0. URL `http://dl.acm.org/citation.cfm?id=1783738.1783748`. - **Cited** on page 23.

C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks: algorithms and evaluation. *Performance Evaluation In P2P Computing Systems*, 63 (3):241–263, 2006. ISSN 0166-5316. - **Cited** on page 47.

Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043559. URL `http://doi.acm.org/10.1145/2043556.2043559`. - **Cited** on page 114.

Google. Cloud SQL: pick the plan that fits your app. http://googleappengine.blogspot.pt/2012/05/cloud-sql-pick-plan-that-fits-your-app.html, May 2012. - **Cited** on page 87.

Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, April 1997. ISSN 0018-9162. doi: 10.1109/2.585156. URL `http://dx.doi.org/10.1109/2.585156`. - **Cited** on page 57.

Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. In *First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004. - **Cited** on pages 15, 53, 54, 60 and 65.

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: http://doi.acm.org/10.1145/1376616.1376713. URL `http://doi.acm.org/10.1145/1376616.1376713`. - **Cited** on pages 39 and 89.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/78969.78972. - **Cited** on page 3.

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855840.1855851`. - **Cited** on pages 54, 108 and 110.

Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: understanding microblogging usage and communities. In *WebKDD/SNA-KDD '07: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 56–65, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-848-0. doi: http://doi.acm.org/10.1145/1348549.1348556. - **Cited** on pages 75 and 76.

Márk Jelasity and Ozalp Babaoglu. T-man: Gossip-based overlay topology management. In *In 3rd Int. Workshop on Engineering Self-Organising Applications (ESOA'05*, pages 1–15. Springer-Verlag, 2005. - **Cited** on page 43.

R. Jimenez-Peris and M. Patiño-Martinez. System and method for highly scalable decentralized and low contention transactional processing.

http://lsd.ls.fi.upm.es/lsd/papers/2011/System-and-method-for-highly-scalable-decentralized-and-low-contention-transactional-processing-descriptionPatent-61-561 - **Cited** on page 98.

F. Junqueira, B. Reed, and M. Yabandeh. Lock-free transactional support for large-scale storage systems. In *7th Workshop on Hot Topics in System Dependability (HotDep'11)*, 2011. - **Cited** on page 98.

David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM. ISBN 0-89791-888-6. doi: http://doi.acm.org/10.1145/258533.258660. - **Cited** on page 11.

David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, New York, NY, USA, 2004. ACM. ISBN 1-58113-840-7. doi: http://doi.acm.org/10.1145/1007912.1007919. - **Cited** on page 54.

Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. On the elasticity of NoSQL databases over cloud management platforms. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 2385–2388, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0717-8. doi: 10.1145/2063576.2063973. URL `http://doi.acm.org/10.1145/2063576.2063973`. - **Cited** on page 116.

Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, Christina Boumpouka, Nectarios Koziris, and Spyros Sioutas. TIRAMOLA: elastic NoSQL provisioning through a cloud management platform. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 725–728, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213943. URL `http://doi.acm.org/10.1145/2213836.2213943`. - **Cited** on page 95.

Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32:422–469, December 2000. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/371578.371598. URL `http://doi.acm.org/10.1145/371578.371598`. - **Cited** on page 37.

Balachander Krishnamurthy, Phillipa Gill, and Martin Arlitt. A few chirps about twitter. In *WOSP '08: Proceedings of the first workshop on Online social networks*, pages 19–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-182-8. doi: http://doi.acm.org/10.1145/1397735.1397741. - **Cited** on pages 75 and 76.

Salma Ktari, Mathieu Zoubert, Artur Hecker, and Houda Labiod. Performance evaluation of replication strategies in DHTs under churn. In *MUM '07: Proceedings of the 6th international conference on Mobile and ubiquitous multimedia*, pages 90–97, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-916-6. doi: http://doi.acm.org/10.1145/1329469.1329481. - **Cited** on pages 21, 24 and 40.

Tirthankar Lahiri, Vinay Srihari, Wilson Chan, N. MacNaughton, and Sashikanth Chandrasekaran. Cache fusion: Extending shared-disk clusters with shared caches. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 683–686, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4. URL `http://dl.acm.org/citation.cfm?id=645927.672377`. - **Cited** on page 89.

Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. ISSN 0163-5980. doi: 10.1145/1773912.1773922. URL `http://doi.acm.org/10.1145/1773912.1773922`. - **Cited** on pages 2, 11, 17, 25 and 60.

Matthew Leslie, Jim Davies, and Todd Huffman. Replication strategies for reliable decentralised storage. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*, pages 740–747, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2567-9. doi: http://dx.doi.org/10.1109/ARES.2006.108. - **Cited** on pages 21, 24 and 40.

H.C. Lim, S Babu, and J.S. Chase. Automated control for elastic storage. *Proceeding of the 7th international conference on Autonomic computing*, pages 1–10, 2010. URL `http://dl.acm.org/citation.cfm?id=1809051`. - **Cited** on page 116.

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 Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043593. URL `http://doi.acm.org/10.1145/2043556.2043593`. - **Cited** on page 114.

Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, 7(2):72–93, 2005. ISSN 1553-877X. - **Cited** on page 10.

L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 123–132, New York, NY, USA, 2006. ACM. ISBN 1-59593-384-0. - **Cited** on page 47.

Miguel Matos, Ricardo Vilaca, José Pereira, and Rui Oliveira. An epidemic approach to dependable key-value substrates. In *International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV 2011)*, June 2011. - **Cited** on pages 45 and 114.

Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag. ISBN 3-540-44179-4. - **Cited** on page 21.

Erik Meijer. The world according to LINQ. *Commun. ACM*, 54(10):45–51, October 2011. ISSN 0001-0782. doi: 10.1145/2001269.2001285. URL `http://doi.acm.org/10.1145/2001269.2001285`. - **Cited** on pages 3 and 48.

Erik Meijer and Gavin Bierman. A co-relational model of data for large shared data banks. *ACM Queue*, 9(3):30:30–30:48, March 2011. ISSN 1542-7730. doi: 10.1145/1952746.1961297. URL `http://doi.acm.org/10.1145/1952746.1961297`. - **Cited** on pages 89 and 115.

P Nadkarni and C. Brandt. Data Extraction and Ad Hoc Query of an Entity-Attribute-

Value Database. *Journal of the American Medical Informatics Association*, 5(6): 511–527, 1998. - **Cited** on pages 25, 48 and 90.

Jurriaan Persyn (NetLog). Database sharding at netlog, with mysql and php. http://www.jurriaanpersyn.com/archives/2009/02/12/ database-sharding-at-netlog-with-mysql-and-php/., 2012. - **Cited** on page 89.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD'08*, 2008. - **Cited** on pages 3 and 48.

M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999. ISBN 0-13-659707-6. - **Cited** on pages 17, 37 and 89.

J. Pereira, L. Rodrigues, M.J. Monteiro, R. Oliveira, and A.-M. Kermarrec. Neem: network-friendly epidemic multicast. *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 15–24, Oct. 2003. ISSN 1060-9857. - **Cited** on pages 43 and 46.

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 2005. - **Cited** on pages 3 and 48.

C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA, 1997. ACM. ISBN 0-89791-890-8. doi: http://doi.acm.org/10.1145/258492.258523. - **Cited** on pages 13 and 16.

Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association. - **Cited** on page 15.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer* *communications*, pages 161–172, New York, NY, USA, 2001. ACM. ISBN 1-58113-411-8. doi: http://doi.acm.org/10.1145/383059.383072. - **Cited** on pages 20, 23 and 43.

S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the Conference on File and Storage Technologies*. USENIX, 2003. - **Cited** on page 13.

John Risson, Aaron Harwood, and Tim Moors. Stable high-capacity one-hop distributed hash tables. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 687–694, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2588-1. doi: http://dx.doi.org/10.1109/ISCC.2006.152. - **Cited** on page 53.

Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag. ISBN 3-540-42800-3. - **Cited** on pages 14, 20, 21, 43 and 46.

Michael Rys. Scalable SQL. *ACM Queue: Tomorrow's Computing Today*, 9(4):30, April 2011. ISSN 1542-7730 (print), 1542-7749 (electronic). doi: http://dx.doi.org/10.1145/1966989.1971597. - **Cited** on page 87.

Hans Sagan. *Space-Filling Curves*. Springer-Verlag, New York, 1994. - **Cited** on pages 19 and 61.

Cristina Schmidt and Manish Parashar. Flexible information discovery in decentralized distributed systems. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, page 226, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1965-2. - **Cited** on pages 19 and 20.

Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, Berkeley, CA, USA, 2007. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267903.1267904`. - **Cited** on page 17.

Thorsten Schutt, Florian Schintke, and Alexander Reinefeld. Structured overlay without consistent hashing: Empirical results. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, page 8, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2585-7. - **Cited** on pages 19 and 21.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972. URL `http://dx.doi.org/10.1109/MSST.2010.5496972`. - **Cited** on page 96.

A. Sousa, J. Pereira, L. Soares, A. Correia Jr., L. Rocha, R. Oliveira, and F. Moura. Testing the Dependability and Performance of Group Communication Based Database Replication Protocols. In *International Conference on Dependable Systems and Networks (DSN'05)*, june 2005. - **Cited** on page 65.

Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001. - **Cited** on pages 12, 20, 21, 29, 43, 44, 46, 53 and 60.

Michael Stonebraker and Rick Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54(6):72–80, June 2011. ISSN 0001-0782. doi: 10.1145/1953122.1953144. URL `http://doi.acm.org/10.1145/1953122.1953144`. - **Cited** on pages 87 and 90.

Michael Stonebraker and Joseph M. Hellerstein. *Readings in database systems (3rd ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-523-1. - **Cited** on page 35.

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL `http://dl.acm.org/citation.cfm?id=1325851.1325981`. - **Cited** on pages 87 and 89.

A. Thusoo, J.S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996 –1005, march 2010. doi: 10.1109/ICDE.2010.5447738. - **Cited** on page 38.

D. Tran and Cuong Pham. PUB-2-SUB: A Content-Based Publish/Subscribe Framework for Cooperative P2P Networks. *NETWORKING 2009*, 5550:770–781, May 2009. doi: 10.1007/978-3-642-01399-7. URL `http://www.springerlink.com/index/1705131P5411181W.pdf`. - **Cited** on page 47.

B Trushkowsky, P. Bodík, A Fox, M.J. Franklin, M.I. Jordan, and D.A. Patterson. The SCADS director: Scaling a distributed storage system under stringent performance requirements. *Proc. of FAST*, 2011. URL `http://www.usenix.org/event/fast11/tech/full_papers/Trushkowsky.pdf`. - **Cited** on page 116.

Twitter. Twitter api documentation. http://apiwiki.twitter.com/Twitter-API-Documentation, 2010. - **Cited** on page 75.

Ricardo Vilaça, Francisco Cruz, and Rui Oliveira. On the expressiveness and trade-offs of large scale tuple stores. In Robert Meersman, Tharam Dillon, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems, OTM 2010*, volume 6427 of *Lecture Notes in Computer Science*, pages 727–744. Springer Berlin / Heidelberg, 2010. URL `http://dx.doi.org/10.1007/978-3-642-16949-6_5`. - **Cited** on page 92.

Werner Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, 2009. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1435417.1435432. URL `http://doi.acm.org/10.1145/1435417.1435432`. - **Cited** on page 2.

Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Consistent join queries in cloud data stores. Technical Report IR-CS-068, 2011. - **Cited** on page 38.

Qin Xiongpai, Cao Wei, and Wang Shan. Simulation of main memory database parallel recovery. In *SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference*, pages 1–8, San Diego, CA, USA, 2009. Society for Computer Simulation International. - **Cited** on page 65.

Haifeng Yu, Phillip B. Gibbons, and Suman Nath. Availability of multi-object operations. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on*

Networked Systems Design & Implementation, pages 16–16, Berkeley, CA, USA, 2006. USENIX Association. - **Cited** on pages 20, 21, 24 and 40.

K. Zellag and B. Kemme. Real-time quantification and classification of consistency anomalies in multi-tier architectures. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 613 –624, april 2011. doi: 10.1109/ICDE.2011.5767927. - **Cited** on page 115.

B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001. - **Cited** on pages 13, 21 and 23.

Ming Zhong, Kai Shen, and Joel Seiferas. Correlation-aware object placement for multi-object operations. In *ICDCS '08: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, pages 512–521, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3172-4. doi: http://dx.doi.org/10.1109/ICDCS.2008.60. - **Cited** on pages 20 and 48.