Cloudy: A Modular Cloud Storage System

Donald Kossmann Tim Kraska* Simon Loesing
Stephan Merkli Raman Mittal Flavio Pfaffhauser

Systems Group, ETH Zurich
{firstname.lastname}@inf.ethz.ch

* *University of California, Berkeley*
kraska@cs.berkeley.edu

ABSTRACT

This demonstration presents Cloudy, a modular cloud storage system. Cloudy provides a highly flexible architecture for distributed data storage and is designed to operate with multiple workloads. Based on a generic data model, Cloudy can be customized to meet application requirements. The goal of this demonstration is to show the ability of Cloudy to efficiently process different query languages, and to automatically adapt to varying load scenarios.

1. INTRODUCTION

Cloud storage systems are increasingly gaining popularity for all kinds of deployments. They promise to simplify administration, be fault-tolerant and able to scale on commodity hardware. This makes them so attractive, that more and more developers prefer cloud storage solutions even for smaller self-hosted environments over traditional database systems.

By today, a huge variety of cloud storage systems is available, all with different functionality, optimizations and guarantees. That is, solutions focus on a particular scenario and the provided services are tailored accordingly. As a result, cloud storage systems vary in the data format (e.g., Key/Value vs. row store), access-path optimization (e.g., read vs. write, one-dimensional vs. multi-dimensional access), distribution (e.g., single vs. multi-data center distribution), query language, transaction support, availability etc. For example, Cassandra [6] uses the fully distributed ring architecture of Dynamo [2] and, thus, can be available even in the presence of network partitioning. In contrast, HBase[7] values consistency over availability and does not tolerate network partitioning. HBase offers versioning capabilities and is particularly suited for read workloads, whereas Cassandra only uses a simple version number for conflict resolution and is better suited for write workloads. Both systems offer a multi-dimensional access path but no complex filtering, joins etc. Redis [5], another open-source system, only supports a one-dimensional access path, is master-slave based with reduced availability in the presence of network partitioning, but offers an interesting data model, which allows to store queues and stacks in it. Next to Redis, many other systems like Voldemort, MongoDB, Scalaris, etc. exist all positioning themselves somewhere in the design space of availability, consistency, data model, and workload optimizations.

Although the variety is huge, it is still likely that the user needs are not perfectly covered by one of the systems. For example, if one would like the data model of Redis but the availability of Cassandra, a completely new system needs to be built. Today's systems are over-customized, and force people to either adapt their applications to the storage system or develop their own system, which also explains the huge variety of currently available systems. Furthermore, by deciding for a particular setup, users narrow the application focus (e.g., either analytical or transactional) and change the way the application can evolve over time, often resulting in misuses of services.

To overcome this system jungle we started to build Cloudy, a modular cloud storage system. The goal of Cloudy is to provide a configurable cloud storage system, which can be tailored and modified by the user to meet his requirements. Cloudy allows customizing the availability and consistency guarantees to the user's needs by adjusting the consistency protocol, the routing (e.g., master-slave vs. DHT) and the load balancing (e.g., load calculator and strategy) components. Furthermore, Cloudy allows for using the storage engine which best suits the application (e.g., in-memory vs. disk). In order to enable all components to communicate seamlessly with each other, Cloudy introduces one generalized data model called *DPI*. The DPI model not only allows to map richer models (e.g., relational, RDF) to it, but also simplifies the processing throughout the system as it enables representing query and data in the same format. Finally, Cloudy does not restrict itself to a Key/Value interface, instead it allows for different higher interfaces (e.g. SQL, XQuery) and - as data and queries are the same - Cloudy naturally supports continuous queries and streaming scenarios.

We currently have a working prototype of Cloudy with several key features that we would like to demonstrate. The implementation includes a Key/Value and a SQL interface as well as streaming capabilities. As all configurations naturally inherit the common parts of the systems, we can further show the fault tolerance as well as the automatic scale up and scale down depending on the load for all use cases. In the remainder of this demonstration proposal, we first

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 2
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

present an architectural overview of Cloudy, highlighting the modular architecture to be presented during the demonstration. Then we summarize the presented features, namely the query interfaces (Key/Value, SQL, XQuery Streaming) & load balancing and scale up, with details on how they will be used to showcase the major technical features of Cloudy.

2. SYSTEM OVERVIEW

In this section, we outline the main properties and features of Cloudy. This includes a description of the modular architecture and the data model, as well as a brief presentation of all components. This section ends with a demonstration of a typical data workflow inside Cloudy.

2.1 Big Picture

From the beginning, Cloudy has been designed in a modular way with the goal to operate with easily exchangeable components. In this flexible architecture, each component has a specific task, implements a defined set of functionalities and is connected to other components via clear interfaces. By using different implementations of one component, specific properties of the system can be modified (e.g., changing the protocol component can change the way data is replicated) and the system can be tuned to fulfill specific requirements or execute particular workloads.

At the core of Cloudy is the modular distribution architecture. Each node in the system is an autonomously running process which executes the entire Cloudy component-stack. This implies that every node can receive requests, process these requests and at the same time physically store parts of the data. In Cloudy, every node covers all functionalities of the storage system and renders a centralized control unnecessary. To avoid an undefined behavior, each node in a running Cloudy system shares the same configuration and components. However, depending on the used components, different nodes might take different roles in the system (e.g., by master election one node can become the manager of the complete system). Cloudy even allows to change the topology from fully distributed to centralized control depending on the user requirements.

Cloudy supports several replication and partitioning schemes for distributing the load across nodes. Having several replicas of the data increases the overall failure resilience of the system. Cloudy distinguishes between the partitioning (how data is grouped together), the routing (to determine where the data is stored) and the consistency protocol (how different replicas are kept in sync). It even allows combining different partitioning schemes with different consistency and routing models. One of the currently existing implementations is the quorum protocol combined with a distributed hash table for routing and order-preserving hashing similar to the techniques used in Dynamo [2]. The quorum protocol provides high scalability as well as availability, but reduced consistency. An alternative implementation which we use for the streaming workload, consists of a query-based partitioning, with DHT routing and a strong consistency protocol for keeping replicas in sync (at the cost of reduced availability). This, together with the fact that Cloudy distinguishes between temporary and permanent failures, facilitates the trade-off between consistency, availability and tolerance against network partitioning as defined by the CAP-theorem individually for every expected use case.

In a distributed storage system load, is not always evenly

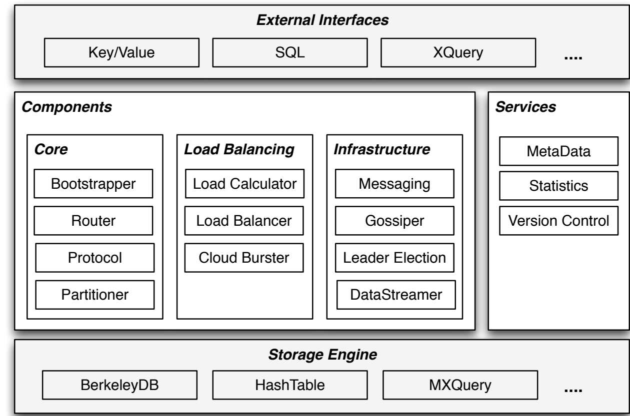

Figure 1: Cloudy Modular Architecture

distributed and "hot spots" (nodes with very high load) show up. That is why Cloudy continuously keeps track of the load situation on each node. As soon as a node suffers from high load, a part of its data is shifted to another node and consequently the load is reduced. In case load balancing among existing nodes is no longer possible (e.g., all nodes are overloaded), the system will *cloudburst*. Cloudbursting refers to the ability to dynamically add or remove nodes from the system depending on the load. This ability to flexibly adapt to changing load situations is one of the main advantages of the cloud computing paradigm and is a key to high scalability as well as optimized resource utilization. Again, Cloudy offers several alternative strategies for performing load balancing and cloudbursting operations.

Cloudy supports a variety of interfaces to access the data. The most simple interface is Key/Value which supports *put* and *get* requests. More advanced interfaces include SQL and XQuery. All query languages and data models are automatically translated by the interface to the internal DPI model, which is explained in Section 2.3. Finally, Cloudy supports different storage engines, ranging from in-memory hash map to relational stores (e.g., Berkeley DB).

This freedom of choice of data model, query language, storage format, consistency and distribution architecture allows customizations way beyond the capabilities of existing cloud storage systems.

2.2 Components

Figure 1 gives an overview of the component-architecture of Cloudy. In the following we give a brief description of the most important components and how they are related to each other:

- **External Interfaces**: The external interfaces represent the access points to the system. Cloudy currently provides a Key/Value, SQL and continuous XQuery interface. The SQL interface is offered through MySQL, for which we implemented a storage engine translating the SQL requests to Cloudy's internal operations and data model. The XQuery streaming interface offers registering continuous XQuery and push events to the system. For all interfaces, we offer client-side routing. This means that the location of data is cached and new requests are always directly sent to the responsible node.
- **Router**: The router's task is to assign data to nodes

and to keep track where data is located. The protocol uses the router to identify all endpoints (nodes) which are responsible for a specific data item. Currently, the router is implemented as a distributed hash table (DHT), however, static routing tables or trees are alternatives.

- **Partitioner**: The task of the partitioner is to determine data groups which should be placed together on one node. At the moment, Cloudy supports domain (i.e., database) partitioning, table or column-based grouping.
- **Protocol**: The protocol coordinates data access and is the layer between the external interfaces and the persistent store. The protocol guarantees a specific level of replication and data consistency. One implementation of this component is the quorum protocol as described in the previous section. However, also more advanced consistency implementations are possible including consistency rationing [3], which automatically adapts the level of consistency at run-time.
- **Store**: The store is responsible for storing data persistently. Current implementations of the store either use an in-memory hash map or the Berkeley DB (BDB) for data storage or MXQuery [4] for continuous queries.
- **Gossiper**: The gossiper's role is to determine the liveness of nodes and to propagate metadata throughout the system. The latter allows to synchronize the DHT among the nodes and minimizes the number of misrouted messages in the system.
- **Load Balancer** and **Load Calculator**. In regular intervals, the load balancer calls the load calculator and then decides whether to change the load distribution and repartition the data.
- **Cloudburster**: The cloudburster checks in regular intervals if new nodes have to be added or removed from the system. As this typically requires a centralized decision, the default implementation depends on leader election to avoid over-adding machines.

Some components, shown in Figure 1 as services, are not exchangeable. These services are deeply integrated into the system core and modularizing them would significantly increase complexity while adding little value. However, future versions of Cloudy might modularize them if use cases make it reasonable.

Because of space constraints, we are not able to describe all components of Figure 1 in detail. In particular, we do not describe the mentioned services and infrastructure components. Those components/services provide low-level functionalities for all other components (e.g., the messaging services or the statistics module) and are not expected to be frequently individually configured by the user.

2.3 Data Model

The data model of Cloudy is based on a data structure we call the DPI. The DPI is a set of fields which can be used to identify, transport and query data items. A DPI contains a key (unique identifier of a data item), information about the data structure (e.g., to which domain the item belongs), the data itself and additional metadata. In combination with three basic data operations to read, write and delete data, the DPI is a powerful data model, generic enough to be used to implement a multitude of more complex data models (e.g., a relational model). The following three basic data operations are provided by Cloudy:

- `Set<DPI> get(DPI dpi)`
- `void put(DPI dpi)`
- `void delete(DPI dpi)`

Each request to the system must first be transformed into a DPI object from the external interface before being processed by the system. While the *put* and *delete* operations do not return any value, the *get* operation returns a set of zero, one or more DPI objects. A DPI object is composed of the following fields:

- **Key**: The key is the unique identifier of a DPI. Different DPIs with the same key point to the same requested data.
- **Type**: The type is used as an additional identifier. It allows defining the data as part of a domain or a hierarchical data model (e.g., define database, table and attribute of the relational model). Depending on the implementation, the pair (key, type) uniquely determines a DPI.
- **Value**: The value field contains the data as a binary object.
- **Lifetime**: The lifetime defines how long data is valid. Essentially, with this field we introduce data with a limited durability for streaming. Thus, in the streaming use case, data (i.e., events) is streamed into the system and not stored unless an continuous query extends the lifetime.
- **Version**: The version is an internally maintained field that contains version information. It can be used by the protocol to provide specific consistency guarantees.
- ***Hint (optional)***: With the hint field, additional information can be passed between components (e.g., from the front-end to the store).

It is important to note, that the DPI serves as the data model and query language by using wildcards and simple variables. Depending on the data operation and how the fields are set, the DPI can have a different meaning. For a *put* operation at least the *key*, *type* and the *value* fields have to be set. This causes Cloudy to create a unique identifier based on the type and key. Queries on the other hand, can have wildcards for the different DPI fields. For example, a wildcard for the key and value, but not the type, would return a list of all DPIs matching the type. Only the lifetime field cannot be used for matching the data, as a value directly defines the lifetime of the query (e.g., 0 for one-time queries and ∞ for continuous queries).

The following data operations exemplify the query mechanism of Cloudy using DPI objects
(syntax: `DPI<Key,Type,Value,LifeTime>`):

- `put DPI<Mike, db1:customer, "MyStree;MyCity", ∞>` is a put request that permanently stores a new DPI with the key *Mike* and type *db1:customer*.
- `get DPI<Mike, db1:customer, *, 0>` returns the previously inserted DPI.
- `get DPI<[a,M],db1:*, *, 0>` is a range query which returns all DPIs matching the type db1:* with a key in the range between a and M.

To illustrate how Cloudy processes data, we will explain a sample workflow of a *get* operation in the next section.

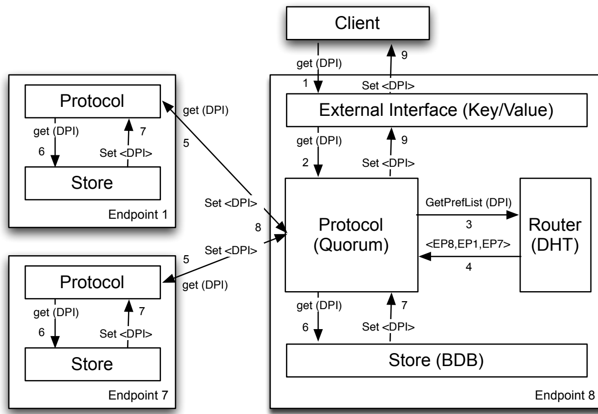

Figure 2: Sample Data Workflow in Cloudy

2.4 Sample Data Workflow

Figure 2 presents a data workflow of a *get* request through Cloudy. In this example we use the Key/Value interface, the DHT implementation, the quorum protocol (with replication factor of three and read quorum of two) and the BDB store. Furthermore, we assume that no failures occur during execution.

1. The client issues a *get* request to Cloudy. Because of client side routing the request is directly sent to the responsible endpoint 8.
2. The external interface forwards the get request to the quorum protocol.
3. The quorum protocol checks whether this endpoint is responsible to handle the request and asks the router component (here, a DHT) to provide it with the preference list (list of all replicas) for the given DPI.
4. The router checks in its routing table which endpoint is responsible for the DPI (endpoint 8) and which endpoints are the two replicas (endpoint 1, endpoint 7).
5. The quorum protocol sends out the get request to all the endpoints in the preference list using the messaging component.
6. The quorum protocol on each endpoint receives the get request message and checks again the responsibility as data might have been moved. If the node is responsible, it accesses the local store or otherwise forwards the request.
7. The store sends back a set of DPIs to the quorum protocol.
8. The retrieved set of DPIs is sent back to the requester protocol. Because the read quorum is set to two the responsible endpoint will only wait for two responses to arrive.
9. Finally the result set is sent back to the Client.

3. DEMONSTRATION DETAILS

In order to demonstrate the key features of Cloudy, we will show how the different query languages and data models are automatically mapped to Cloudy's internal data model and operations. Furthermore, we will demonstrate how the system adapts to changing workloads by balancing a set of nodes and automatically adding and removing nodes in the system. For the whole demonstration Cloudy will be executed on virtual machines provided by the Amazon Elastic Compute Cloud (EC2) service.

3.1 Query Processing Demo

Cloudy's data model is designed to support a variety of different query languages and data models. In this first part of the demonstration, we will show how Cloudy's internal data model and the supported operations are used for a Key/Value store, the relational model with SQL and streaming with XQuery. To demonstrate the capabilities, we will use different queries ranging from key and key-range requests to the full implementation of the LinearRoad streaming benchmark [1]. In addition, we will allow the user to issue custom queries against pre-loaded data and visualize how those query translate to the underlying data model and operations. Finally, we will outline how the system can be configured by choosing a partitioning, routing and storage engine.

3.2 Load Balancing and Scale Up Demo

Cloudy permits exchanging single parts of the system to customize it for different workloads without requiring to change other components in the system. Thus, all configurations can naturally profit from already existing features such as failure detection and load balancing and it is not required to re-invent the wheel. In this second part of the demo we are going to show, how the system automatically adapts itself to failures and changing workloads.

For the Key/Value store, we are going to create a synthetic workload of read and write requests to the system. The workload is changing over time (e.g., the key-request distribution changes) and by means of a GUI we will demonstrate how the system adapts itself by re-partitioning the data and adding and removing nodes to the system.

For the streaming use case, we are going to use the LinearRoad benchmark in order to show the streaming as a service capabilities of Cloudy. However, the design objective of Cloudy is not to minimize the latency of single queries (Cloudy so far has no automatic query splitting capabilities). Cloudy focuses on supporting many users with several queries simultaneously. Hence, instead of issuing one LinearRoad benchmark and varying the load of it, we issue several LinearRoad benchmarks simultaneously and vary the number of benchmarks over time. Again, by means of the same GUI we are going to present how the system adapts itself to the changing workload.

4. REFERENCES

[1] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *Proc. of VLDB*, pages 480–491, 2004.

[2] G. DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of SOSP*, pages 205–220, 2007.

[3] T. Kraska et al. Consistency Rationing in the Cloud: Pay only when it matters. In *Proc. of VLDB*, 2009.

[4] MXQuery. MXQuery: A lightweight, full-featured XQuery Engine. http://mxquery.org/, March 2010.

[5] Redis. Redis. http://code.google.com/p/redis/, March 2010.

[6] The Apache Software Foundation. Cassandra. http://cassandra.apache.org/, March 2010.

[7] The Apache Software Foundation. HBase. http://hadoop.apache.org/hbase/, March 2010.