Query Processing in the Cloud

Master Thesis
Raman Mittal
mittalr@student.ethz.ch

Supervised by:
Dr. Tim Kraska
Prof. Dr. Donald Kossmann

Systems Group, http://www.systems.ethz.ch/
Department of Computer Science
ETH Zurich

October, 2009 - March, 2010
Abstract

Companies with huge amounts of data have already gone past what could reasonably fit on a single machine. Relational databases (RDBMS) seem to have a limitation on handling such big data volumes and modern workloads. This has led to the introduction of various cloud storage systems like Amazon Dynamo, Yahoo PNUTS, Google Big Table, Cassandra, Microsoft Azure etc. All of these systems store data in various formats ranging from a simple key-value pair to a specific data model. Similarly, they provide different query interfaces to access the stored data.

However, SQL is still the standard of data processing and none of the modern cloud storage service interface is rich enough to provide the users with SQL like query capabilities of aggregation and joins. This thesis introduces a cloud storage system, Cloudy2 which apart from offering a highly scalable and available storage system, offers a SQL query interface for data access. In addition to this, the thesis advertises the highly modular architecture of Cloudy2 which allows a user to tailor it as per his requirements.
Acknowledgments

I would like to express thanks towards my supervisors Dr. Tim Kraska and Simon Loesing for their constant feedback, guidance and support during the course of this thesis. Next, I would like to thank the other Cloudy2 development team members, Stephan Merkli and Flavio Pfaffhauser for realizing a great team work. I would also like to thank Prof. Donald Kossmann for his valuable discussions and encouragement during our meetings. I express my gratitude towards the Systems Group for the very friendly and helpful atmosphere. Last but not least, many thanks to all my friends who supported me during this thesis, as well as my lovely family.
Contents

1 Introduction 1
   1.1 Motivation ........................................ 1
   1.2 Problem Statement ................................. 2
   1.3 Contributions ..................................... 2
   1.4 Structure of the Thesis ......................... 2

2 Cloudy2 5
   2.1 Overview ........................................ 5
   2.2 Architecture ..................................... 5
      2.2.1 Cloudy2 Data Model - DPI ....................... 6
      2.2.2 External Client Interfaces ..................... 7
      2.2.3 Components ................................ 9
      2.2.4 Services .................................. 14
      2.2.5 Cloudy2 Configuration File .................. 15
   2.3 Sample Workflow ................................ 16
      2.3.1 Put Request .................................. 16
      2.3.2 Get Request .................................. 18
      2.3.3 Delete Request ................................ 18

3 MySQL 19
   3.1 Overview ........................................ 19
   3.2 Architecture ..................................... 20
      3.2.1 Storage Engine API ......................... 21
      3.2.2 SQL Query Execution ....................... 23

4 MySQL On Cloudy2 25
   4.1 Overview ........................................ 25
   4.2 Architecture ..................................... 26
      4.2.1 Cloudy2 Storage Engine .................... 26
      4.2.2 Cloudy2 Storage Layer ....................... 29
      4.2.3 Data Partitioning ............................ 29
   4.3 Query Execution Workflows ...................... 30
      4.3.1 Table Creation ............................... 30
      4.3.2 Insert Query ................................ 31
      4.3.3 Update Query ............................... 31
      4.3.4 Delete Query ............................... 31

5 Indexes and Optimizations 33
   5.1 Index Maintenance ............................... 33
5.1.1 Index Query Execution ........................................ 34
5.2 Optimizations ..................................................... 35
  5.2.1 Batching ..................................................... 35
  5.2.2 Prefetching .................................................. 36
  5.2.3 Condition Pushdown ....................................... 36
  5.2.4 Digest Query ................................................ 37
  5.2.5 Partial Fetching ......................................... 39
  5.2.6 Serialization ............................................. 39
  5.2.7 Configuration ............................................. 39

6 Experiments and Benchmarks .................................... 41
  6.1 Microbenchmarks ............................................. 41
    6.1.1 InnoDB vs Cloudy2 BDB Store .......................... 43
    6.1.2 Database Level Partitioning ............................ 44
    6.1.3 Key Level Partitioning .................................. 48
  6.2 TPC-W Benchmark ............................................ 52
    6.2.1 System Behavior under Load ............................ 54

7 Conclusions ..................................................... 57
  7.1 Future Work .................................................. 57
    7.1.1 Transaction Support .................................... 58
    7.1.2 Cloudy2 Storage Engine ................................. 58
    7.1.3 Cloudy2 Storage Layer .................................. 58
    7.1.4 Full TPC-W Benchmark ................................... 58
    7.1.5 Failure Scenarios ....................................... 59
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Cloudy2 Components, Services and Interfaces</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Three way gossip message exchange between EP1 and EP2</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Sample request workflow in Cloudy2</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>MySQL architecture with pluggable storage engines</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>MySQL instances running on Cloudy2 cluster nodes</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Thrift interface communication</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Berkeley DB secondary indexes</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Cloudy2 batching process</td>
<td>36</td>
</tr>
<tr>
<td>6.1</td>
<td>InnoDB vs Cloudy2 BDB store</td>
<td>43</td>
</tr>
<tr>
<td>6.2</td>
<td>InnoDB vs Cloudy2 BDB store</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Response time comparison for insert and index query</td>
<td>45</td>
</tr>
<tr>
<td>6.4</td>
<td>Response time comparison for range scan and join query</td>
<td>45</td>
</tr>
<tr>
<td>6.5</td>
<td>Response time comparison for insert and index query</td>
<td>46</td>
</tr>
<tr>
<td>6.6</td>
<td>Response time comparison for range scan and join query</td>
<td>46</td>
</tr>
<tr>
<td>6.7</td>
<td>Response time comparison for insert and index query (cache = 3.0 GB)</td>
<td>47</td>
</tr>
<tr>
<td>6.8</td>
<td>Response time comparison for index read (cache = 6.0 GB)</td>
<td>47</td>
</tr>
<tr>
<td>6.9</td>
<td>Response time comparison for range scan (cache = 6.0 GB)</td>
<td>48</td>
</tr>
<tr>
<td>6.10</td>
<td>Response time comparison for insert and index query</td>
<td>49</td>
</tr>
<tr>
<td>6.11</td>
<td>Response time comparison for range scan and join query</td>
<td>49</td>
</tr>
<tr>
<td>6.12</td>
<td>Response time comparison for insert and index query</td>
<td>50</td>
</tr>
<tr>
<td>6.13</td>
<td>Response time comparison for range scan and join query</td>
<td>50</td>
</tr>
<tr>
<td>6.14</td>
<td>Response time comparison for insert and index query</td>
<td>51</td>
</tr>
<tr>
<td>6.15</td>
<td>Response time comparison for range scan and join query</td>
<td>51</td>
</tr>
<tr>
<td>6.16</td>
<td>Response time comparison for insert and index query</td>
<td>52</td>
</tr>
<tr>
<td>6.17</td>
<td>Cloudy2 behavior under load</td>
<td>54</td>
</tr>
</tbody>
</table>
List of Tables

3.1 MySQL storage engines ........................................... 20
4.1 Customer table ..................................................... 28
6.1 Fixed configuration for all experiments. ....................... 41
6.2 Experiment datasets. .............................................. 42
6.3 tpcw database size ............................................... 53
Chapter 1

Introduction

This chapter describes the motivation behind the idea of Cloudy2 and this particular thesis.

1.1 Motivation

Modern business software and services demand high scalability and availability. This has resulted in an increasing demand for large scale infrastructures which generally turns out to be highly complicated and expensive for the organization needing them. These higher resource and maintenance costs have given rise to a paradigm shift towards cloud computing wherein services are offered and maintained by various providers over the Internet. These service offerings range from software applications to virtualized platforms and infrastructures. Some of these major cloud service providers are Amazon [1], Google [2], Microsoft [3] and Salesforce [4]. Using such cloud services not only prevents the users from purchasing and maintaining hardware but also cuts down the operational costs as they pay only for the resources they use. It is the responsibility of service providers to maintain the cloud infrastructure required to host such services and support multiple clients. Some of these cloud storage systems are Dynamo [5], PNUTS [6], BigTable [7], Voldemort [8], Apache Cassandra [9]. However, these systems are developed with a focus towards the particular service they aim to support. Modifications to such systems are not trivial and incur high costs. So it is quite possible that users are forced to implement a new cloud storage system if their demands are not completely fulfilled by an existing system. This motivated the idea behind Cloudy2, which aims to develop a cloud storage system built upon a generic architecture and independent of any particular use case scenario. The modularity of Cloudy2 allows users to use it as a base implementation of a cloud storage system and easily extend it as per their requirements.

Apart from this, cloud computing has also changed the view on data management by focusing primarily on cost, flexibility and availability instead of ACID guarantees of traditional RDBMS. Generally, these storage systems trade-off the data consistency [10] to provide the user with a more fault tolerant and scalable service. Some of these cloud storage systems do provide a rich query interface for data access but still lack many useful functionalities of query languages like SQL. Since SQL is still widely used for data management, it would be quite useful for cloud services to enable users to access data over an interface which provides
Introduction

similar functionalities like SQL. The primary motivation behind this thesis is to add query processing support on Cloudy2 which allows a user to access data over a standard SQL interface. This Cloudy2 SQL interface would not only offer the currently missing SQL functionalities in cloud storage services but would also enable an easy porting of existing SQL based applications to cloud environments.

1.2 Problem Statement

Over-customization of cloud storage systems has resulted in a large variety of complex systems but none with the desired level of flexibility and extensibility. With Cloudy2, we propose that a generic cloud storage system architecture can prevent this jungle of cloud systems to grow further. Modular architecture is one of the major design goals for Cloudy2. It is desired that its various modules are loosely coupled and interact with each other over well defined service interfaces. The modularity feature of Cloudy2 allows its modules to be exchanged in accordance to a specific use case requirements. This thesis primarily focuses on development of a particular use case which aims to support a SQL query interface for Cloudy2. This would allow storage and retrieval of data from Cloudy2 using the standard SQL query language which is missing in modern cloud storage services.

1.3 Contributions

The initial contribution of this thesis is development of the Cloudy2 system as a highly available and scalable key-value service. Cloudy2 was developed with a highly modular architecture which could be used to easily implement new use cases by extending or exchanging Cloudy2 components. The major contribution of this thesis is the development of a specific use case which provides SQL query interface for Cloudy2. To accomplish this, a new storage engine was plugged into MySQL which allows MySQL to use Cloudy2 for its data storage. This Cloudy2 storage engine acts as an external Cloudy2 client and communicates with it for data access whenever SQL queries are executed on MySQL. To enable this communication between the storage engine and Cloudy2, a thrift interface was developed over Cloudy2. The thesis also introduces various optimizations implemented to enhance the response time for the SQL queries e.g. prefetching, indexing, batching and serialization of data stored in the Cloudy2 system. In the end, the SQL interface for Cloudy2 was tested and benchmarked with different configurations of Cloudy2.

1.4 Structure of the Thesis

This thesis is structured as follows:

- **Chapter 1**: Describes the motivation, problem statement and the respective contributions made by this thesis.

- **Chapter 2**: Provides an overview of various Cloudy2 architectural parts like its data model, client interfaces, modules and configuration settings. The chapter also includes sample control workflows through the Cloudy2 system.
1.4 Structure of the Thesis

- **Chapter 3**: Describes architecture of MySQL and its storage layer. Later it explains the steps involved during a SQL query execution on MySQL.

- **Chapter 4**: Describes the components involved in supporting SQL interface over Cloudy2. The later part of the chapter walks through some SQL query execution scenarios on MySQL with Cloudy2 as its storage layer.

- **Chapter 5**: Describes the index maintenance mechanism for MySQL data in Cloudy2. It also describes various performance optimizations for data access.

- **Chapter 6**: Presents the benchmarking and evaluation results of various SQL query executions over Cloudy2.

- **Chapter 7**: Concludes the thesis and presents the future work for the SQL interface and Cloudy2.
Chapter 2

Cloudy2

2.1 Overview

This chapter first presents the general system requirements for Cloudy2 and then provides an overview of its architecture.

The development on Cloudy2 started with the following system requirements:

- **Scalability**: should scale autonomously in all dimensions e.g. storage capacity, computing power, number of users and number of requests.
- **Availability**: should have no single point of failure and offer application desired replication guarantees.
- **Adaptability**: should have a modular architecture which is independent of any particular use case.
- **Decentralization**: should be based on peer-to-peer techniques. Each machine in the cluster should be symmetrical.
- **Flexible data model**: should be able to support different use cases, such as key-value store, relational store, stream processing, application hosting etc.

2.2 Architecture

The figure 2.1 shows the various architectural blocks of Cloudy2:

- **External Interfaces**: These are the interfaces or services provided by Cloudy2 for external client interaction.
- **Components**: Components are those modules which primarily comprise the core infrastructure for Cloudy2. They interact with each other over well defined interfaces. This loose coupling between various components allows them to be modified or exchanged without affecting other components.
- **Services**: Services comprise of those functionalities in Cloudy2 which are cohesive with the underlying system implementation and cannot be exchanged or defined in a generic manner. These services are generally used by Cloudy2 components to accomplish their tasks.
The next section first presents the Cloudy2 data model which is a generic model for data exchange between various parts of Cloudy2. Following section would then briefly discuss the various parts of Cloudy2.

2.2.1 Cloudy2 Data Model - DPI

The Cloudy2 data model or DPI encodes complete information associated with a data access query or its response in a generic manner. This generality in DPI structure allows it to represent such information for any use case scenario which Cloudy2 aims to support. Whenever a Cloudy2 client needs to access data from Cloudy2, it creates a DPI request with relevant information and sends it to a Cloudy2 server. Cloudy2 then uses that DPI to interpret the request, route the request and then store or return results to the client.

**DPI Structure**

- **Type**: The type is a string that describes the domain associated with a DPI. A domain could be a generic class or a namespace to which the DPI belongs. For example with MySQL use case, it could either be a database name, table name or a combination of both.

- **Key/Key-Range**: A key or a key range can be set in the DPI. The type and key/key-range fields together represent a unique key identifier for that DPI in Cloudy2. If the type field is not set in the DPI, the key itself acts as the unique key identifier. This unique key identifier is used to store or fetch the DPI from Cloudy2. Both the key and key-range are of string data type. A key-range contains a range of keys encoded in form of a regular expression. For example, \([\text{rangeStartKey}, \text{rangeEndKey}]\) represents a range taken lexicographically from \(\text{rangeStartKey}\) to \(\text{rangeEndKey}\) both inclusive. The user can also specify exclusivity with both range limits.
2.2 Architecture

- **Value**: The value field is a byte array which contains the data associated with the unique key identifier of a DPI.
- **Version**: This field is used internally by Cloudy2 to maintain the version information of a stored DPI. This is necessary to provide the required level of data consistency.
- **Hint**: Hints field is a byte array which can be used by Cloudy2 or its client to provide more information on how to process a DPI query. For example, hints could include indexing information, data filters, result data limit, query identifier etc.

Sample DPIs

Following are queries encoded as a DPI in a triplet of <key, type, value>.

- DPI <keyx, , valx>: A DPI containing the key as keyx, no type and value valx. This represents a normal put (keyx, valx) query with unique identifier as keyx.
- DPI <keyx, , >: A DPI containing just the key as keyx with no type or value. It is a normal get (keyx) query with unique identifier as keyx.
- DPI <{ keyx, keyy }, , >: A DPI containing a range from keyx (exclusive) to keyy (inclusive). It is a normal get ([keyx, keyy]) range query.
- DPI <keyx, typex , >: A DPI containing the key as keyx, the type as typex and no value. A normal get (typex:keyx) query with unique identifier as typex:keyx.
- DPI <{ keyx, keyy }, typex , >: A DPI containing a range from keyx (exclusive) to keyy (inclusive), with typex as type and no value. This is a normal get ([typex:keyx, typex:keyy]) range query.

Throughout this thesis the term key when associated with a DPI refers to the unique identifier for that DPI i.e. key of the DPI if type is not defined or a combination of key and type, key:type, if type is defined. Also, in this thesis, the term DPI and <key, type, value, hints> means the same, as the later is just a structural representation of the former.

2.2.2 External Client Interfaces

The Cloudy2 storage system is currently accessible to the clients over the following external interfaces:

Java

This interface allows Java clients to interact with the Cloudy2 Java server using the following methods:

- **Set<DPI> get (DPI)**
  This service method retrieves results in form of DPIs for a given DPI query from Cloudy2 storage.

---

1The hint field is not shown in the example, as it contains use case specific data for each DPI e.g. index information or query filters for SQL use case.
• **void put (DPI)**
  This service method stores a given DPI in Cloudy2 storage.

• **void delete (DPI)**
  This service method deletes a given DPI from Cloudy2 storage. Any further
  *get (DPI)* for the deleted DPI would return empty results.

All above methods throw appropriate exceptions if due to some reason,
the request could not be executed successfully by Cloudy2.

**HTTP**
This is a REST interface for the HTTP clients and provides the same service
methods as done by the Java interface.

**SQL**
The SQL interface enable clients to use the standard MySQL shell to access data
from Cloudy2 using SQL queries.

**Thrift**
The thrift interface was primarily introduced due to the need of a cross-language
communication support while introducing the SQL interface. This communication gap arose because MySQL is written in C/C++ and Cloudy2 is developed
completely in Java. Apache Thrift [11] library was used to close down this communication gap. Apart from this, the thrift interface can support clients written
in languages like C/C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#,
Cocoa, Smalltalk, and OCaml to communicate with Cloudy2 system through a
thrift server running on each Cloudy2 node. The thrift interface provides the
following service methods :

• **Set<thriftDPI> get (thriftDPI)**
  This service method retrieves results in form of thriftDPIs for the given
  thriftDPI query.

• **void put (thriftDPI)**
  This service method stores a given thriftDPI in Cloudy2 system.

• **void delete (thriftDPI)**
  This service method deletes a given thriftDPI from the Cloudy2 system.

• **insertTableMetadata (tableDesc td)**
  This service method is used by MySQL to notify Cloudy2 about creation of
  a new table. The Cloudy2 node stores the new table description as metadata
  in the Cloudy2 cluster. This metadata is later on used by Cloudy2 to process
data access and index maintenance requests for that table.

The thriftDPI structure can be considered as a one to one mapping to the Cloudy2
DPI structure. It contains exactly the same fields as of a Cloudy2 DPI and was
introduced to communicate with Cloudy2 over the thrift interface. It is used by
a thrift client to send DPI request data to the Cloudy2 thrift server. The Cloudy2
thrift server then transforms it into a Cloudy2 DPI so that it can be interpreted
by various Cloudy2 components.
2.2 Architecture

JMX

All parts of the Cloudy2 system can register their implementations with JMX, allowing them to be viewable and modifiable at runtime. This feature proved to be very helpful during development and testing of new settings and prevented the need of repeatedly restarting the Cloudy2 cluster.

2.2.3 Components

This section explains the various components of the Cloudy2 system.

Messaging System

The messaging system supports the exchange of control and update messages between two nodes or endpoints. An endpoint registers a handler with the messaging system for each type of message it expects to handle. On reception of a message by the messaging system it invokes the handler registered with that particular message type. The system supports both TCP or UDP style communication.

The current implementation is based upon the latest Apache Cassandra [9] messaging system available with version 0.5.0.

Leader Election

The leader election component can be used to elect a leader among a group of endpoints. The leader is elected on the basis of grant of a distributed lock on a unique string identifier. The endpoint which first requests the lock from the lock server, gets it and all other endpoints making the same request have to wait. In the current implementation the lock service is implemented using the Apache Zookeeper [12] centralized service. The Zookeeper instances are run replicated on seeds of the Cloudy2 cluster. Seeds are those special Cloudy2 nodes which are contacted by any new bootstrapping node wishing to join the Cloudy2 cluster.

Gossiper and Failure Detector

The gossiper is responsible to make sure that every endpoint in the Cloudy2 system eventually learns about the new state information for every other endpoint upon a state change event. The Cloudy2 gossiper implementation is primarily based on Apache Cassandra project. Gossiper runs as a timer task on every Cloudy2 node and gossips the state information to other Cloudy2 nodes using an epidemic protocol. In each gossiping round, it exchanges the endpoint state information of its known endpoints with a randomly selected live endpoint, a seed and an unreachable endpoint (with a hope to contact that node in this round).

Each endpoint state consists of the endpoint’s heartbeat state and the application state. The heartbeat state can be considered as a pulse of an endpoint which is incremented for each round of gossip initiated from that endpoint. The application state comprise of the state and its version associated with various active components on that endpoint. The gossiper running on a Cloudy2 node maintains the state information as mappings of endpoint to endpoint state for all endpoints it knows.
As shown in the figure 2.2, each gossiping round consists of three way message exchange between two end points.

- First the endpoint (EP1) which starts the gossiping sends a GossipSynMessage. This message contains a list of GossipDigests which is created for each end point in its endpoint state map. Each GossipDigest contains the endpoint information and the maximum version of its endpoint state known to this endpoint. Both the heartbeat state and the application state are considered while finding the maximum version of state. In the figure 2.2, EP1 would send a GossipSynMessage to EP2 containing GossipDigest \( gd1 \) as \((\text{EP1:5}), (\text{EP2:4}), (\text{EP3:9}), (\text{EP4:3})\)

- Once the other endpoint (EP2) receives the GossipSynMessage, it checks from the GossipDigest \( gd1 \), if the gossiping endpoint has more recent (higher version) state information for any endpoint it knows or an information about a new end point. In the figure 2.2, EP2 would find that it has outdated versions for EP1 and EP3. Apart from this, if this endpoint finds that it has higher version state information for an endpoint (by comparing the max version of the endpoint in its endpoint state map and its version in gossipdigests from the gossiping node), it adds that state difference to a GossipDigestAck message which is then sent to the gossiping endpoint. This message also contains the information about the endpoints for which the updated state information needs to be fetched from the gossiping endpoint. In the figure, EP2 would request the updated state for EP1 and EP3 in \( gd2 \) and it would add the most recent state \( eps2 \) for EP2 and EP4 for which EP1 lags behind.

- The gossiping end point (EP1) on receiving the GossipDigestAck message applies the state information updates to the endpoints for which it was lagging behind and sends a GossipDigestAck2 message to the other endpoint with the requested state information. Here EP1 would update the states of EP2 and EP4 and sends the requested state information by EP2 for the endpoint EP1 and EP3 \( eps1 \).
2.2 Architecture

- The other endpoint (EP2) on receipt of GossipDigestAck2 message, updates its endpoint state information for EP1 and EP3 using eps1 and finishes the gossiping round.

Apart from exchanging the state information associated with each endpoint, the gossiper also takes care of the group membership view of the cluster and notifies the registered listeners whenever there is an associated event. To accomplish this, the gossiper subscribes to the failure detector module which notifies it on membership change events. On reception of any higher version state information for a endpoint, the gossiper updates the failure detector about that endpoint by sending a notification. The failure detector maintains a heartbeat window for each endpoint, which contains the samples of last received information for that endpoint. If the failure detector finds that the information is not received for an extended period of time, then that endpoint is added to the list of suspects. This leads to notifications being sent to the failure detector subscribers. The information and its interpretation in failure detection is based on the Phi Accrual failure detector [13].

The gossiper also maintains a live endpoints, unreachable endpoints and dead endpoints list. On receiving a notification from the failure detector for a suspect endpoint, gossiper removes that endpoint from the list of live endpoints and adds it to the list of unreachable endpoints. Live endpoint are the ones for which this endpoint is actively getting higher versions of their state information. An unreachable endpoint which does not respond or sends its state information for more than certain Cloudy2 threshold, is declared as dead.

Router

The router is used to provide routing information for a given DPI. The routing information is provided in form of a preference list which contains the endpoints responsible for storing that DPI. The first endpoint in the preference list is the master and the rest are the replicas. The master can be considered as the endpoint which in a concurrent environment, forces an ordering on the DPI requests. This ordering is implemented by setting a version for the DPI.

The current router implementation is based on the consistent hashing [14] technique. Each endpoint in the cluster holds a token or a set of tokens (called virtual nodes) which signifies its position or positions on a ring of endpoints. When the router receives a DPI, it creates a token for that DPI by hashing DPI’s key, based upon a hashing strategy. The hashing strategy basically maps a DPI key to a token which represents a position on the ring. Each endpoint is responsible for all DPIs which hash to the range between the endpoint tokens and their predecessors.

In current implementation, an endpoint stores the token range information in a routing map implemented over a skip list data structure.

Protocol

The protocol holds the responsibility to service client requests by handling routing, storage and retrieval of users data. Apart from this, the protocol also carries out the necessary tasks to ensure replication and consistency of data in the system. After receiving a client request, the external interface passes on the client DPI to the protocol for processing. The current protocol implementations supported by
Cloudy2 are:

- **Simple Protocol**: This protocol implementation does not support any replication or replication factor is 1. The DPI is routed to the first endpoint or master in the preference list obtained from the router. This endpoint is responsible for storage and retrieval of data. In case of a DPI get request, the master directly retrieves the DPI from the store and returns the results. When the master receives a DPI put request, before storing the DPI it also sets the version (timestamp) of the DPI. This is used to force an ordering on the DPIs when Cloudy2 receives concurrent put requests for the same DPI. In case of concurrent put requests with same DPI, the DPI with higher version (timestamp) gets persisted to the store. Since there is only a single endpoint which stores the data, this protocol implementation supports strong consistency but is not fault tolerant. So, if an endpoint goes down, it is not possible for Cloudy2 clients to retrieve data that was stored on the failed endpoint.

- **Quorum Protocol**: This implementation uses a quorum of endpoints to service the client DPI. The protocol allows a configurable replication factor in addition to configurable read and write quorum size through protocol settings. For a given DPI, the protocol fetches the preference list from the router which contains the master and replica endpoints for that DPI. The protocol sends the user request to the master and all replicas and wait until a quorum of responses is received. This quorum would be a read quorum in case of a get request and write quorum in case of a put or delete request. A put request returns as soon as the write quorum is reached. In case of a get request, once the read quorum is reached, the obtained DPI result set goes through an additional resolution process. In resolution process, result DPIs sets from all responses are compared based on the versions of DPIs. This process creates the final DPI result set consisting of most recent DPIs obtained from the master and its replicas. This DPI result set is then sent back to the external interface which returns it to the Cloudy2 client.

**Protocol consistency levels**:

- **Strong consistency**: When the quorum settings create a valid quorum, this protocol would provide strong consistency. However, the consistency levels might drop to eventual consistency if the master endpoint fails and puts are still taking place for DPIs which belongs to its responsible range.

- **Eventual consistency**: When the quorum settings do not form a quorum or during master failure scenarios only eventual consistency [15] can be guaranteed.

**Store**

The store component acts a storage layer for Cloudy2 data. Cloudy2 stores the user data in form of key-value pairs where key is a DPI’s unique identifier and value is the DPI itself. Following are the store implementations currently supported by Cloudy2:

- **In Memory Store**: This is a simple non-persistent map which stores the key-value pairs. It does not provide any support to maintain indexes.
• **Berkeley DB Store**: This acts as a persistent storage implementation for Cloudy2 data. Berkeley DB [16] (BDB) is an open source, embeddable, key-value pair database written entirely in Java. Despite having a simple architecture, BDB supports many advanced database features such as ACID transactions, fine-grained locking, hot backups and replication.

The current implementation of Cloudy2 BDB storage layer offers the following functionalities:

- Provides a storage (get, put and delete) interface for key-value pairs.
- Internally maintains indexes for stored data and provides the flexibility to choose index data structure type.
- Maintains a caching layer to enhance the read performance.
- Implements write operations like put and delete as transactions.
- The implementation also supports an iterator access which can be used for a range scans.
- The store offers a possibility of partial retrieval of value associated with a key. This can substantially enhance performance when only a few bytes are required to be retrieved from a DPI with a huge value data. For example, to count the number of DPIs in the store, retrieving a single byte for each key, should be enough.

**Bootstrapper**

This component is used to bootstrap a new Cloudy2 node. Whenever a new node wishes to join the Cloudy2 cluster, its bootstrapper invokes a bootstrapping method. This method contacts the Cloudy2 seeds to receive the bootstrap information. Bootstrap information is generally the information about the range of DPI for which the new node would be responsible in the Cloudy2 cluster. The bootstrap information depends upon the Cloudy2 router implementation. For example, in the DHT implementation of router, the bootstrap information would be the token or position of the new node on the ring. This information can also contain multiple tokens, in case Cloudy2 supports virtual nodes. If the new node is unable to contact any of the Cloudy2 seeds or receives an invalid bootstrap information and it is not a Cloudy2 seed itself, it fails to bootstrap and shuts down. If in case, this node is one of the Cloudy2 seeds, it contacts its own router for bootstrap information (randomly generated) and creates a new Cloudy2 cluster.

**Load Calculator**

This component calculates the load value on an endpoint based on different factors like the CPU usage, memory consumption, disk usage, number of DPIs or a combination of various factors. This load value is used during the load balancing step to take decisions on how to balance the load, if required.

**Load Balancer**

This component executes as a timer which is scheduled at regular intervals. It takes load balancing decision by examining the load value associated with an endpoint. Following are the two implementations currently supported by Cloudy2:
• **Simple Load balancer**: It tries to balance the load only with its neighbors.

• **Maha Load balancer**: It extends the simple load balancer and makes use of virtual nodes to balance load with any light nodes in the cluster. This implementation borrows its name from authors of "Scalable Range Query Processing for Large-Scale Distributed Database Applications" [17].

**Repartitioner**

This component is invoked by the load balancer to alter the partitioning of the data placement. The implementation of this is tightly bound to the router implementation. For example, a token based repartitioner has to be used with a DHT based router implementation.

**Cloud Burster**

Similar to the load balancer task, the cloud burster also runs as a task and monitors the load in the system. Based on the average load in the system the cloud burster task decides to bootstrap a new node into the system if it finds the system to be overloaded. It also uses the leader election component, to make sure that only a single node starts the cloud bursting process. This avoids the scenario where multiple nodes decide to start cloud bursting at the same time due to high system load.

### 2.2.4 Services

Cloudy2 services are highly dependent on the implementation of some lower-level mechanisms which are not required to be exchangeable.

**Data Streamer**

The data streaming service is used to stream a set of DPIs from a Cloudy2 node, whenever there is a change in Cloudy2 routing map. This can happen during the load balancing step when a node needs to take additional ownership of DPIs which were discarded by another node due to high load. Or during bootstrapping a new node when it is assigned an initial position in the cluster and it needs to take the store DPIs for which it would be responsible in future. The data streaming service is implemented over TCP connections using asynchronous non-blocking I/O.

**Metadata Handling**

The metadata handler is responsible for maintaining consistency for Cloudy2 metadata stored in the system. The metadata is stored in Cloudy2 as a normal DPI but with a special key "cloudy2_metadata". The version of metadata DPI is incremented with each update of Cloudy2 metadata in the system. When a node updates the Cloudy2 metadata, it schedules a task which does a put (DPI) for the updated metadata DPI. The Cloudy2 metadata DPI version number is sent along with each message exchanged between two endpoints. If an endpoint receives a message that contains a higher Cloudy2 metadata version than what it
contains locally, it immediately updates its metadata by doing a get (DPI) for the metadata DPI.

Statistics
The statistics service collects system monitoring information like memory consumption, CPU usage, number of DPIs per node and free disk space. The statistics are stored in the application state associated with endpoint state for each endpoint. The statistics gets gossiped around by the gossiper component as a part of the endpoint state. The statistics are used by the load calculator to retrieve the metrics needed to calculate the load on an endpoint.

Version Control
Cloudy2 supports a single version for each stored DPI. The version control service uses the DPI version field to make sure that only the latest version get stored in the system. If a put request is received in future for a DPI which has a newer version stored, the request is discarded by throwing an exception notifying the same. Version to a DPI is always assigned by the master node for that DPI. After assignment of the version to a DPI, the master sends the put DPI request to its store and the replicas, if any. Cloudy2 being an asynchronous system, makes use of the version field to avoid the possibility of old DPIs from overwriting the newer ones. The version could either be a timestamp or a version number.

2.2.5 Cloudy2 Configuration File
Cloudy2 system configuration can be set from a XML file named settings.xml file. It contains the configuration parameters for various Cloudy2 factories and components.

Components Configuration Parameters
Most of the Cloudy2 components consists of various parameters which can be tuned according to application requirements. For example:

- **Messaging System**: connection port numbers, message ack timeout values.
- **Gossiper**: gossiping interval, failure suspect threshold, cluster name.
- **Protocol**: replication factor, read quorum size, write quorum size, number of protocol stage threads.
- **Store**: store cache size.
- **Load Balancer**: high load thresholds, timeouts.
- **Router**: hashing strategy.
- **External Interfaces**: server listening port numbers, number of server threads.
Factory Configurations
A Cloudy2 factory comprises of a unique combination of Cloudy2 components which serve the purpose of some particular use case scenario. These components are specified as XML nodes and dynamically loaded when a new node bootstraps. The settings.xml contains multiple Cloudy2 factories, each representing a different use case. Each Cloudy2 node bootstraps in the cluster with the same factory configuration as the other cluster members.

Below is a sample factory configuration used for hosting the Cloudy2 SQL interface:

```
<factory id="mysql-cluster">
  <messaging> messaging.cassandra2.net.MessagingService </messaging>
  <gossiper> gossiper.cassandra.Gossiper </gossiper>
  <router> router.dht.DHTRouter </router>
  <protocol> protocol.quorum.QuorumProtocol </protocol>
  <store> store.bdb.mysqlbdb.MySqlBdbStorageEngine </store>
  <load-balancer> loadbalancer.maha.MahaLoadBalancer </load-balancer>
  <repartitioner> repartitioner.dht.DHTRepartitioner </repartitioner>
  <bootstrapper> bootstrapper.def.DefaultBootstrapper </bootstrapper>
  <leader-election> leadelelection.zookeeper.LeaderElectionService </leader-election>
</factory>
```

2.3 Sample Workflow
This section illustrates the workflow for each of the get (DPI), put (DPI) and delete (DPI) scenarios. For this illustration we assume that Cloudy2 is configured to use a DHT based router and the quorum protocol. The quorum settings (N/W/R) are 3/2/2, where N is the replication factor, W is the size of the write quorum and R is the size of read quorum. Figure 2.3 presents a snapshot of request processing in Cloudy2.

2.3.1 Put Request
A Client request is received in form of a DPI by the external interface in step 1. This DPI to be stored should contain a key, type (optional) and a value. In step 2, the external interface passes on the DPI to the protocol layer, quorum protocol in this workflow. The quorum protocol then contacts the router to get preference list for this DPI in step 3. The router returns a list with the endpoints EP2, EP3 and EP4 in this order, in step 4. The quorum protocol then checks if it is the master of that given DPI. Because in this case endpoint EP1 is not in charge of that DPI (checked from the preference list), the request is forwarded to the master of that DPI, EP2 in step 5. The quorum protocol of EP2 receives the redirected request in step 5. Like before, the protocol requests the preference list for this DPI from the router in step 6. Assuming that the view of the cluster did not change from the last step, this preference list would contain EP2 as the master.
Figure 2.3: Sample request workflow in Cloudy2
for the DPI in step 7. The quorum protocol now sets the version (timestamp) and sends the DPI to its replicas EP3 and EP4 (step 8). After sending the DPI to the replicas, the protocol contacts the local store and stores this DPI in (step 9) which is acknowledged by the client in step 10. The same process is executed by the replicas on receiving the DPI (step 9 and 10 on endpoint EP3 and EP4) and each of them sends an acknowledgement to the master (EP2). EP2 waits until it receives a write quorum number of acknowledgements (including from its own store), in this case $W = 2$. Once the write quorum acknowledgements are received within the protocol timeout limit, EP2 sends the acknowledgement in step 12 to EP1 which initially redirected the actual client DPI request. When EP1 receives the acknowledgement it informs the client about the successful or unsuccessful execution of the request in step 13 and 14. The reply shown in the picture 2.3, refers to a success acknowledgement or a failure notification. A failure to store a DPI can occur due to a protocol timeout while waiting for write quorum responses. A put also fails in the store, if it already contains another DPI with the same key and a newer timestamp.

### 2.3.2 Get Request

The workflow for the get request is very similar to that of a put. The main difference being that the reply messages between the various components contains the result set of DPIs which would be returned to the client. The quorum protocol instead of waiting for write quorum of responses, now waits for read quorum of responses. Once the quorum protocol receives the read quorum of responses, as explained before, it goes through an additional process called the resolution process. This is carried upon by the quorum protocol before sending the results to EP1 in step 12. During this resolution process it compares the versions of all the received DPIs from different nodes (master and replicas) and creates a result set containing DPIs with the most recent versions. Additionally, if the master (EP2) finds that one of the replicas (EP3 or EP4) has an outdated version for a DPI, it schedules a read repair process which updates the outdated replica by sending a put (DPI) request with most recent value of the DPI from EP2.

### 2.3.3 Delete Request

The delete works exactly the same way as the put. The only difference being that instead of the store doing a put for the DPI, does a delete. All subsequent get (DPI) for that DPI would return empty results.
Chapter 3

MySQL

3.1 Overview

MySQL [18] is one of the most popular open source SQL relational database management system (RDBMS). It runs as a server daemon which provides multi-user access to a number of databases. It is written in C/C++ and uses the GPL [19] license to define what user may and may not do with the software in different situations. It is one of the widely used database for delivering online applications. Many high-traffic web sites (including Facebook, Yahoo, Wikipedia, Google and YouTube) use MySQL as a database component of their LAMP software stack. In addition to fulfillment of RDBMS requirements, MySQL provides some additional features like:

- **Pluggable storage engine architecture**: This makes it reasonably easier and cleaner to use different storage engines for MySQL as per requirements. It also makes it possible to add a new storage engine to a running MySQL server dynamically at the run time.

- **Commit grouping**: It gathers multiple transactions from various connections together so as to increase the number of commits per second.

In this thesis, we primarily exploit the pluggable storage engine architecture of MySQL to plug a custom storage engine which uses Cloudy2 to store and fetch MySQL data. This also helps us to provide a SQL interface over Cloudy2.

Storage Layer

The pluggable storage architecture of MySQL provides the user with great flexibility to make decision on the storage engine based on the application requirements like consistency level, transaction support, indexes data structures, scalability, availability, replication, data storage format, caching requirements etc. A storage engine is the module of MySQL server which decides upon the storage format and mechanism for storing MySQL data. It also manages the indexing of stored data. With MySQL, a user is not limited to a single storage engine but can use multiple storage engines in a single application. This allows for optimal performance tuning of an application which can now use the best engine for each job. For example, read intensive module would prefer an engine which optimizes the read
performance or an application which does not need ACID guarantees would use an engine which can optimize the performance at the expense of ACID properties. Users are also free to develop their own storage engine [20] if none of the existing storage engine satisfies their application requirements.

There are plenty of storage engines with different properties and behaviors available for MySQL. Some of the most widely used ones are:

- **InnoDB**: Used for transaction processing applications, and supports a number of features including full ACID transaction support.
- **MyISAM**: The default MySQL pluggable storage engine and the one that is used the most in web, data warehousing, and other application environments.
- **MySQL Cluster/NDB**: The Clustered database engine of MySQL that is particularly suited for applications with high performance lookup needs and high availability.
- **BDB**: An alternative transaction engine to InnoDB that supports COMMIT, ROLLBACK, and other transactional features.
- **CSV**: References comma-separated files as database tables.

The following statement creates a table using a non-default storage engine, InnoDB [21] in this case.

```
mysql> create table example (...) engine=InnoDB;
```

If no `engine` is specified during table creation, the default storage engine would be used which is MyISAM [22] in standard MySQL source compilation.

The `show engines` statement shows the list of all storage engines available with that particular MySQL compilation.

```
mysql> show engines
```

<table>
<thead>
<tr>
<th>Engine</th>
<th>Comment</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndbcluster</td>
<td>Clustered, fault-tolerant tables</td>
<td>YES</td>
</tr>
<tr>
<td>CSV</td>
<td>CSV storage engine</td>
<td>NO</td>
</tr>
<tr>
<td>MEMORY</td>
<td>Hash based, stored in memory, useful for temporary tables</td>
<td>NO</td>
</tr>
<tr>
<td>Cloudy2</td>
<td>Cloudy2 MySQL storage engine</td>
<td>NO</td>
</tr>
<tr>
<td>ARCHIVE</td>
<td>Archive storage engine</td>
<td>NO</td>
</tr>
<tr>
<td>InnoDB</td>
<td>Supports transactions, row-level locking, and foreign keys</td>
<td>YES</td>
</tr>
<tr>
<td>MyISAM</td>
<td>Default engine as of MySQL 3.23 with great performance</td>
<td>NO</td>
</tr>
</tbody>
</table>

Table 3.1: MySQL storage engines

### 3.2 Architecture

The MySQL database software is based on a client-server architecture. The server is a multithreaded process which manages the access to multiple databases. To
execute queries, multiple clients can connect simultaneously to the server via several connection protocols. Figure 3.1 [18] shows the architecture of a MySQL server. The MySQL server receives the client queries in SQL format. Once a query is received, the parser module parses the query into internal MySQL binary structures which can be interpreted by its optimizer module. Optimizer is a server module which is responsible for creating the most efficient query execution plan based on the query requirements and additional hints from the storage engine for executing that query. The idea behind the optimization stage is to deliver the query result in least possible amount of time. The exact form in which MySQL reduces the SQL query after parsing and optimization stage, can be seen by using the explain statement. More information on explain could be found in MySQL documentation [18].

Figure 3.1: MySQL architecture with pluggable storage engines

3.2.1 Storage Engine API

The MySQL server communicates with storage engines through a well defined handler interface. A handler abstract class in the MySQL source code declares this interface. Each MySQL storage engine extends this handler abstract class and provides an implementation of its abstract methods. The first method a storage engine needs to support is the method for creation of a new handler instance. A handler is instantiated for each thread that needs to work with a table. For example, if there are four connections from a client to the same table, four different handler instances will need to be created. Once a handler instance is created, the MySQL server invokes methods of the handler to perform data storage and retrieval tasks such as opening a table, manipulating rows, and managing indexes. The following text presents some of the frequently used handler API methods and
their important function arguments.

- **create (const char *name, TABLE *form, HA_CREATE_INFO *info)**
  The *name* parameter is the name of the table. The *form* parameter is a TABLE structure that defines the table and matches the contents of the tablename.frm file already created by the MySQL server to maintain table information. This function is invoked on a new table creation and is used by storage engines to possibly create or initialize data files for the new table.

- **open (const char *name, int mode)**
  The *name* is the name of the table to be opened. This method is invoked before every read or write operations performed on a table.

- **rnd_init ()**
  This method is called before any full table scan is performed. It can be used to prepare for a table scan and initialize appropriate data structures.

- **info ()**
  Prior to commencement of a table scan, this method is called to provide extra information about the table to the optimizer. The optimizer might choose different query execution path based upon this information.

- **rnd_next (uchar* buf)**
  After the table is initialized, the MySQL server will call this method once for every row to be scanned until the server’s search condition is satisfied or an end of file is reached, in which case the handler returns HA_ERR_END_OF_FILE. In each invocation, the storage engine copies the retrieved row data from the store to the *buf* argument passed to this method.

- **close ()**
  When the MySQL server is finished with a table, it will call this method to close file pointers and release any other resources.

- **write_row (uchar* buf)**
  This method handles the insert row operations on the table. The *buf* contains the row to be inserted in internal MySQL format.

- **update_row (const uchar *old_data, uchar *new_data)**
  The MySQL server executes update statements by performing a (table/index/range) scan until it locates a row matching the where clause of the update statement and then calls this method. The *old_data* parameter contains the data that existed in the row prior to the update, while the *new_data* parameter contains the new contents of the row (in the MySQL internal row format).

- **delete_row (const uchar *buf)**
  The MySQL server executes delete statements using the same approach as for update statements. It advances to the row to be deleted using the rnd_next () method and then calls this method to delete the row. The *buf* parameter contains the contents of the row to be deleted.
3.2 Architecture

- **index_init ()**
  This method is called before an index access and can be used to allow the storage engine to perform any necessary preparation or optimization.

- **index_read (uchar * buf, const uchar * key, key_part_map map )**
  This method is used to retrieve a row based on a index (primary or secondary). The map is a bitmap of the various index parts comprising the index. The *buf parameter is populated with the row data which matches the *key attribute.

- **index_next (uchar* buf)**
  This method is invoked during index scanning. The *buf parameter is populated with the row that corresponds to the next matching key value according to the internal cursor set by the storage engine during operations such as index_read ().

3.2.2 SQL Query Execution

This section shows the sequence of handler API method invocations done in case of a series of query executions on a example table created with the default MySQL engine.

**Table Creation**

```sql
mysql> create table example (e_id int, e_name varchar(10), e_age int, primary key (e_id))
```

1. `create ()`

create () is invoked to inform the storage engine to initialize any data structures required to handle queries on the new table.

**Insert Query**

```sql
mysql> insert into example values (1, "raman", 25)
```

1. `open ()`
2. `write_row ()`

open () is invoked first to initialize the table structures. Subsequently, to insert the row, write_row () is invoked with the row data buffer in MySQL internal row format.

**Full Table Scan**

```sql
mysql> select * from example where e_age=25
```

1. `info ()`
2. `rnd_init ()`
3. `rnd_next ()`
4. `rnd_next()`

Since the where clause contains the condition on a non key attribute, a full table scan would be done to get all results. The `info()` is invoked in the beginning to provide any hints to the MySQL optimizer for optimizing the query. After examining the optimization information given by the storage engine, if the handler decides to do a full table scan, it calls `rnd_next()` each time to read the next record until all records are scanned. In the above execution the `rnd_next()` would be called two times as the table contains just one record. It will return the only row in the first call and `HA_ERR_END_OF_FILE` when called the second time. This notifies the handler that there is no more data to come.

**Index Query**

```sql
mysql> select * from example where e.id=1
```

1. `info()`
2. `index_read_map()`
3. `index_next()`

This is simple index query which does an index scan. Similar to `rnd_next()`, `info()` is invoked first to provide hints and then `index_next()` is called each time to read the next record which matches the requested index value.

**Update Query**

```sql
mysql> update example set e.age=40 where e.age=25
```

1. `info()`
2. `rnd_init()`
3. `rnd_next()`
4. `update_row()`

After scanning through the table, `update_row()` is called for each row which matched the query conditions.

**Delete Query**

```sql
mysql> delete from example where e.age=40
```

1. `info()`
2. `rnd_init()`
3. `rnd_next()`
4. `delete_row()`

Similar to update query execution.
Chapter 4

MySQL On Cloudy2

4.1 Overview

This chapter explains the architecture and implementation of MySQL on Cloudy2 which enables MySQL to use Cloudy2 for its data storage. Cloudy2 as a storage layer for MySQL offers the following features and properties:

- **Scalability**: Spreads across a distributed system and is highly scalable. Cloudy2 can use application desired data partitioning strategy to partition the stored data.

- **Availability**: Replicates data to prevent any single point of failure and offers high availability.

- **Eventual Consistency**: Trade offs the consistency properties of data to offer high scalability and availability.

- **Indexing**: Maintains primary and secondary indexes for data to improve its read throughput.

- **Durability**: Provides data persistence.

- **SQL support**: Allows users to use SQL queries to access data. However, currently SQL constraints, triggers and transactions are not supported.

Cloudy2, when plugged as a storage layer for MySQL enables standard MySQL RDBMS applications with above desired or acceptable properties to be ported in cloud environments without any modifications. Though, it can be inferred from the above properties that these ported applications might have restricted functionalities and guarantees of cloud environments.

Each Cloudy2 node runs a MySQL server instance which allows MySQL clients to connect anywhere in the Cloudy2 cluster as shown in the figure 4.1. For example, a table created with MySQL instance running on EP1 would also be available on all other endpoints EP2, EP3 and EP4. The user can then connect to the MySQL instance running on any of these endpoints and execute SQL queries on the table.
4.2 Architecture

To enable MySQL server to use Cloudy2 as its storage layer, a custom storage engine was developed for MySQL. As explained in chapter 3, the MySQL server communicates with the storage engine using the MySQL storage engine API. To store and fetch MySQL data, the Cloudy2 storage engine interacts with Cloudy2 over the external client interface just like any other external Cloudy2 client. The next part of this section explains the architecture of various modules involved in MySQL and Cloudy2 communication.

4.2.1 Cloudy2 Storage Engine

The Cloudy2 storage engine extends the MySQL handler abstract class and implements the necessary methods required for handling full table scan, index, write, update, delete and create table queries. It interacts with Cloudy2 over the thrift library interface (section 2.2.2) which can be used to implement all types of MySQL data access. Since this particular engine needs to interact with Cloudy2 thrift server using a thriftDPI, it needs to transform the MySQL data access requests to an appropriate \(<\text{key, type, value, hints}>\) \(^1\) structure and then create a thriftDPI from it. As explained in section 2.2.2, Cloudy2 thrift server converts the thriftDPI into a Cloudy2 DPI so that it can be recognized by the Cloudy2 system.

\(^1\)This engine uses the hints field to provide index or query filters information for each query.
Figure 4.2 shows the communication flow between MySQL server and Cloudy2 over the thrift external interface.

1. The received SQL query goes through the parsing and optimization stage in step 1.

2. Once the query execution plan is created after step 1, the MySQL server invokes the appropriate handler methods of the Cloudy2 storage engine. The storage engine receives the information on the query as its input in internal MySQL row format in step 2.

3. The storage engine interprets the MySQL internal format to get the information about how to store or fetch the required data in the method called by the handler. The information is then represented as <key, type, value, hints> which is converted to a thriftDPI. This thriftDPI is sent to Cloudy2 over the thrift interface in step 3.

4. The Cloudy2 thrift server on getting the thriftDPI request, converts it into a Cloudy2 DPI and sends it to the Cloudy2 protocol layer in step 5.
The protocol handles the DPI request by contacting the Cloudy2 node responsible for the DPI and gives back the result to the thrift server in step 8. In case of a get query, the result is a set of DPIs each containing a table record (in value field). In case of a put query, just an acknowledgement is sent for a successful put or delete.

Next the thrift server converts the results into thriftDPI to send them over the thrift library to the Cloudy2 storage engine in step 9.

Finally, the storage engine extracts the data from the results after step 10. The result would contain the table records in case of get (thriftDPI) and an acknowledgement of success or failure in case of a put (thriftDPI) or delete (thriftDPI). Since the table records are already in MySQL internal row format, they are returned to the MySQL server which filters them according to query conditions in step 11 and finally sends them to the user in step 12.

The next section explains how the Cloudy2 storage engine simplifies the data storage and retrieval requests on a sample customer table into a <key, type, value, hints> form.

Consider the customer table 4.1 with primary index on c_id and two secondary indexes one on c_uname and another on c_fname, c_lname created under a database named tpcw.

<table>
<thead>
<tr>
<th>c_id</th>
<th>c_uname</th>
<th>c_passwd</th>
<th>c_fname</th>
<th>c_lname</th>
<th>c_addr_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>raman</td>
<td>ram</td>
<td>raman</td>
<td>mittal</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>nilay</td>
<td>test</td>
<td>nilay</td>
<td>gupta</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>ravi</td>
<td>ravs</td>
<td>ravi</td>
<td>goel</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 4.1: Customer table

Data Storage

Each row of a SQL table is stored in Cloudy2 as a DPI which contains the <key, type, value, hints> where key is the unique identifier obtained by combination of type (database:table) and primary key of the table. The value consists of the byte array of complete row record in MySQL internal row format. Not modifying the MySQL row format of the row, prevents us from any additional overhead of conversion to MySQL row format upon data retrieval from Cloudy2. The Cloudy2 storage engine reduces the data storage request in form of <key, type, value, hints> and then creates a thriftDPI from it. Then it can use Cloudy2 thrift interface put (thriftDPI) to store or update the data whenever API methods like write_row (), update_row () and delete_row () are invoked by the MySQL handler. For example: The <key, type, value, hints> form to insert the first row in customer table would be [tpcw:customer:1, tpcw:customer, val, 0], where val is the byte array of the row in MySQL row format. Since insertions are always done using the primary key, the hints field carries number 0 signifying that the primary index should be used to process it. A thriftDPI would be created from it and it would be used to issue a put (thriftDPI) to insert the row in the table.
4.2 Architecture

Data Retrieval

The Cloudy2 storage engine uses the `get` (thriftDPI) to retrieve data or a data-range from API methods like `index_init()` and `rnd_init()`. Again the thriftDPI encodes the retrieval information in form of `<key, type, value, hints>`. Data retrieval could either be an index access or a full table scan.

- **Index Access**: In this case, the unique key identifier is obtained by combination of the type (database:table) and the index attributes on which the query is done.
  For example: In case of primary index query for first row on `customer` table, the unique identifier key would be `tpcw:customer:1` and in case of secondary index query it would be `tpcw:customer:raman`.

- **Full Table Scan**: The full scan is implemented with a range scan over the table. The storage engine creates a key range to fetch full table data. The key range consists of a `startKeyRange` and `endKeyRange` where `startKeyRange` is the lexicographically minimum key of the table and `endKeyRange` being the maximum.
  For example: In case of full table scan on `customer`, `startKeyRange` would be `tpcw:customer:+` and `endKeyRange` be `tpcw:customer:}`.

To avoid potential problems due to different character encoding on MySQL and Cloudy2 nodes, the Cloudy2 storage engine encodes the key into a Base64 form before generating the unique key identifier. Since Base64 encoding encodes the key in a fixed character set, it also allows us to use `[+,{]` to guarantee a full table scan. Using `'+` and `'}' makes sure that complete data between minimum and maximum key for the table is retrieved as `'+` and `'}' are the minimum and maximum characters in the encoding scheme.

4.2.2 Cloudy2 Storage Layer

To allow Cloudy2 storage to store and manage MySQL data and indexes, the base implementation of the Cloudy2 BDB store described in section 2.2.3 was extended. The extended functionality of the Cloudy2 BDB storage includes handling of table metadata, maintaining indexes and some other data access optimizations. The additional functionalities and optimizations are explained in detail in the next chapter.

4.2.3 Data Partitioning

For scalability reasons Cloudy2 supports different levels of partitioning for each table. This partitioning strategy would be used by Cloudy2 to distribute table data during load balancing in the system. Following partitioning strategies are currently supported by Cloudy2:

- **Database Level Partitioning**: User’s complete database would always reside on the same Cloudy2 node. This might limit the scalability for that database but would give good performance even for queries which involve multiple tables.

- **Table Level Partitioning**: User’s database might be distributed in the Cloudy2 cluster but each table would reside fully on a single node. This
is more scalable than the database level partitioning but might cause additional performance overhead in case of queries involving multiple tables which are distributed over multiple nodes.

- **Row Level Partitioning**: User’s database and tables might be distributed in the Cloudy2 cluster. This horizontal partitioning can result in table rows being distributed in the cluster. This is the most scalable but would give least performance because multiple Cloudy2 nodes might be needed to contact for query execution. Also this partitioning scheme doesn’t support secondary indexes.²

### 4.3 Query Execution Workflows

The following section describes the control flow in execution of some SQL queries on *customer* table created with the Cloudy2 storage engine.

#### 4.3.1 Table Creation

```sql
mysql> create table customer (c_id int not null, c_uname varchar(20), c_passwd varchar(20), c_fname varchar(20), c_lname varchar(20), c_addr_id int, primary key(c_id), index(c_uname), index(c_fname, c_lname)) Engine=Cloudy2;
```

The above query involves the following steps:

1. The handler invokes the `create()` method of the storage engine.

2. The create method initializes a data structure which contains the complete description of the *customer* table. The description contains information about the table columns, data type, column length, indexes, index parts etc. This information can be obtained from the `table_share` structure of the handler.

3. The create method invokes the `insertTableMetadata()` of the Cloudy2 thrift interface and sends the table description structure to Cloudy2.

4. Now, the *customer* table needs to be created on MySQL server instance running on each Cloudy2 node so that each running instance in the cluster knows about it. To do this `insertTableMetadata()` stores the *customer* table description as metadata and updates the metadata version of Cloudy2. This version update notifies all Cloudy2 nodes about change in metadata.

5. On receiving the metadata update, each node performs necessary actions for that update. This involves creation of MySQL table (using the local MySQL server instance) and BDB store secondary indexes on that node for the *customer* table. After this, all Cloudy2 nodes have complete information about the new *customer* table and fully prepared to handle queries on it.

²The reason is that a record is always inserted on a Cloudy2 node based on its DPI key which is created based on the primary key of the record. The secondary index is maintained by the BDB store on that node and the routing component has no routing information to route a DPI which carries a secondary key value in its key field. This works in partitioning by type (table/database) or when there is no Cloudy2 load balancing as the table records are not distributed and always reside together on a single node. The router then can always route each query on a table to the node storing that table.
4.3 Query Execution Workflows

4.3.2 Insert Query

```mysql
insert into customer values (1, "raman", "ram", "raman", "mittal", 10);
```

1. The handler invokes the `write_row()` storage engine method and passes the buffer containing the record in internal MySQL row format.

2. Inside `write_row()`, the primary index key is extracted from the record and a unique identifier is created which is used to perform a `put (thriftDPI)` in Cloudy2. For example, the above query would invoke `put (tpcw:customer:1, tpcw:customer, val, 0)` where `val` contains the bytes of the complete record in internal MySQL row format and hints field carries the number 0 to signify a primary index insertion.

4.3.3 Update Query

```mysql
update customer set c.uname="alias" where c.id=1;
```

1. The handler invokes the `update_row()` storage engine method and passes two buffers containing the old record and new record data respectively in internal MySQL row format.

2. Inside `update_row()`, the primary index key is extracted from the old record data and a unique identifier is created which is used to perform a `put (thriftDPI)` in Cloudy2. For example, the above query would invoke `put (tpcw:customer:1, tpcw:customer, val, 0)` where `val` contains the bytes of the new record data in internal MySQL row format.

4.3.4 Delete Query

```mysql
delete from customer where c.id=1;
```

1. The handler invokes the `delete_row()` storage engine method and passes the buffer containing the record in internal MySQL row format.

2. Inside `delete_row()`, the primary index key is extracted from the record and a unique identifier is created which is used to perform a `delete (thriftDPI)` in Cloudy2. For example, the above query would invoke `delete (tpcw:customer:1, tpcw:customer, , 0)`.

Information on data retrieval and index query execution would be covered in detail in the next chapter.
Chapter 5

Indexes and Optimizations

This chapter explains about the implementation of indexes in the Cloudy2 BDB storage layer for MySQL data. Later in the chapter various possible performance optimizations are discussed.

5.1 Index Maintenance

When the Cloudy2 BDB store receives a put (DPI), it stores the DPI as key-value pair where key is the key of DPI and value is the serialized DPI structure. The Cloudy2 BDB storage layer creates secondary indexes upon creation of a new MySQL table. Supporting secondary indexes in BDB includes writing a secondary index key extractor which, given a key-value pair extracts the secondary key from it. In case of MySQL data, the key would be DPI key (primary key of the record) and value contains the serialized DPI structure which in its value field contains the table row record in internal MySQL row format. Extracting the secondary key from the value requires parsing the MySQL internal row format and retrieving it. In BDB each secondary index is implemented with a unique secondary database which needs to be plugged with the primary database upon instantiation. After plugging of the secondary database with the primary database, whenever a put with key-value pair is done on the primary database, BDB internally sends the request to each of its associated secondary databases. In Cloudy2 BDB store, then each secondary database uses its key extractor to examine the key, type and hints attributes from the DPI in the key-value pair. If it finds that it is responsible for that key, type and index (from hints), it extracts the relevant secondary key from the value field of the DPI and updates itself.

For example, in the figure 5.1, there are three secondary databases associated with the primary database pd

\texttt{tpcw}, two for table \texttt{tpcw:customer} and one for table \texttt{tpcw:address}. When put (DPI) arrives for a customer table record, through the Cloudy2 storage interface, it is stored in the BDB store as a key-value pair. In the figure this key-value pair is sent to the primary database pd

\texttt{tpcw} using BDB put method. The BDB database \texttt{pd

\texttt{tpcw}} before persisting the key-value pair, sends a put request to all its associated secondary databases. Since, the secondary databases \texttt{sd

\texttt{tpcw:customer\_1}} and \texttt{sd

\texttt{tpcw:customer\_2}} hold the index for \texttt{customer}, they need to update themselves. The other index \texttt{sd

\texttt{tpcw:address\_1}} for the table \texttt{address} ignores this update and return false. When the \texttt{pd

\texttt{tpcw}} receives responses from all its associated secondary databases, it persists the key-value pair. When
a get (DPI) is received, the Cloudy2 store accesses the appropriate secondary database associated with the index in the query and retrieves the primary key for that record. After retrieving the primary key, it simply contacts the primary database to fetch the associated value and returns it.

![Cloudy2 store interface diagram]

Figure 5.1: Berkeley DB secondary indexes

### 5.1.1 Index Query Execution

An index query execution on the customer table 4.1 from previous chapter involves the following steps:

1. The MySQL handler invokes the `index_read_map()` storage engine method. It passes two buffers as arguments to this function. One of them contains the index value and another one is used as a result buffer to store the results retrieved by the storage engine.

2. The `index_read_map()` extracts the index value from the buffer and checks the type of index access. The index access could be either primary, secondary or an index range scan. Based on the kind of index query, an appropriate thriftDPI is created which is later on is converted to a Cloudy2 DPI to retrieve the requested data from Cloudy2.

- **Primary Index Query**

  ```sql
  mysql> select * from customer where c_id=id
  ```

  In the above primary index query a `get (tpcw:customer:id, tpcw:customer,.
5.2 Optimizations

, 0 ) would be sent to Cloudy2 thrift server. The value 0 in the hints field signifies a primary index query.

- **Secondary Index Query**

```sql
mysql> select * from customer where c uname="ram"
```

In the above secondary index query a get (tpcw:customer:ram, tpcw:customer, , 1 ) would be sent to Cloudy2 thrift server. The value 1 in the hints field signifies the index number to be used.

- **Index Range Scan Query**

```sql
mysql> select * from customer where c fname="raman"
```

The above query would cause a index range scan, since in the table schema this index actually consists two parts (c fname and c lname). Since the query only specifies the value of one index part, it sends a range get ([tpcw:customer:raman+, tpcw:customer:raman], tpcw:customer, , 2) to scan through the index for records in this range. The value 2 in the hints field signifies the index number to be used.

As shown above, the index queries apart from setting the key, also adds hints for the Cloudy2 DPI hints field. This provides information to Cloudy2 storage layer to choose the right index to fetch data.

3. The Cloudy2 storage layer processes the get (DPI) query using its hints and sends back the results to the Cloudy2 storage engine over the thrift interface.

4. If the number of records obtained from Cloudy2 is more than one, then handler invokes index.next () method until all results are read and copied to the result buffer.

### 5.2 Optimizations

#### 5.2.1 Batching

Batching allows Cloudy2 to retrieve client data in multiple batches. Batching was introduced to prevent clients from overwhelming the Cloudy2 server by performing huge range scans. It enforces an upper limit over the amount of data which can be transferred per query. In case, the amount of data for a query is greater than this limit, the Cloudy2 protocol layer adds an additional Token DPI in the result set signifying the presence of more data. The Token DPI is a range DPI which should be used by the client as the next query to fetch further data from the system. The client holds the responsibility to capture this token DPI and issue the token DPI query. For example, in figure 5.2, when the quorum protocol at EP1 receives the data for the DPI from other Cloudy2 nodes, after the resolution process, it checks if the amount of fetched data is more than the limit enforced upon by Cloudy2. In that case, it appends a Token DPI in the resolved result set. To avoid dropping huge amounts of data at the protocol layer, the Cloudy2 store only sends that amount of result data which complies to the Cloudy2 limit. If store still has more data, it adds a similar Token DPI to let the protocol layer know about it.
5.2.2 Prefetching

Prefetching was introduced in the Cloudy2 storage engine to fetch data in the background while MySQL server is processing the already retrieved data. Prefetching tries to make sure that once MySQL finishes processing with a result buffer, another buffer is already waiting for processing or has already sent the fetch request to Cloudy2 and is waiting for data to arrive. For example during batching, if client receives Token DPIs from Cloudy2, it can prefetch the data by sending the Token DPI query in a separate thread. While the next batch of data is still being fetched, the MySQL processes the previously fetched batch. It switches to the new batch as soon as it has finished processing the old batch and a new one is available to be processed.

5.2.3 Condition Pushdown

In MySQL architecture, the attribute selection and conditional filtering is done at the MySQL level. This means that the storage engine retrieves complete data for a query which is then filtered by the MySQL server based on query conditions. However, MySQL does provide the storage engine with a mechanism to filter the data itself by pushing the query conditions to it. In case of Cloudy2 storage engine, this mechanism of condition pushdown could give better performance for the queries in which MySQL has to drop majority of the retrieved data due to the query conditions. Using this mechanism the Cloudy2 storage engine can push down the conditions with the DPI (hints) and the Cloudy2 storage layer can use them to send only the filtered data. This optimization is not yet implemented in the current version.
5.2 Optimizations

5.2.4 Digest Query

The digest query optimization is implemented at the Cloudy2 protocol level. In case an application requires data replication, it uses a protocol which can support replication, e.g. the quorum protocol. This optimization significantly increases the performance of get (DPI) in cases when data replicas are not divergent.

Now, when the protocol receives the get (DPI) request from the client, it issues get (DPI) requests to the master for that DPI and to each of its replicas as explained in the sample workflows section in chapter 2. Since the quorum protocol only needs to compare the versions of the DPIs from master and the replicas, it is not a good idea to fetch the full DPI from the replicas. A full DPI from replicas should only be fetched in case, the replica contains a more recent version of it. So with this optimization, instead of fetching a full DPI from the replicas, the quorum protocol only fetches the DPIs with their version field. This DPI is also called a digest for the actual DPI at the replica. During the resolution process, the protocol compares the digests from the replicas with the data from master. If it finds that the version of a DPI at the replica is more recent than at the master, it explicitly issues a get (DPI) request for that DPI to fetch it fully.

In case of range queries, the replicas are requested to send a single hash (MD5 or Adler) value obtained from the versions of DPIs in the requested range. If during the resolution process, the protocol finds that there is hash mismatch between the master and the replica, it requests the replicas to send the digest DPIs for the given range DPI. Now after getting the digests from the replicas, the protocol checks if any replica holds a more recent DPI version than the one held at the master. If this is the case, the protocol sends the get (DPI) request to that replica and fetches the complete DPI.

Since most of the time the replicas are consistent with each other, normally this approach boosts the performance of the get (DPI) with the quorum protocol. The replicas can become divergent in situations of concurrent reads and writes on same DPI.

Implementations other then the one introduced above are also possible for this optimization. Using Merkle trees [23] to obtain information about the point of divergence between the replicas is one of them. Below we compare the Merkle trees approach with the digest query approach.

Replica Synchronization Approaches

Digest Query approach:

- Send digest query for the key to its replicas. This would retrieve the version of the value associated with the key.

- After resolution process in the protocol, fetch the complete key-value from the replicas if the digest with higher versions (as compared to master) were received from the replicas.

- Send read repair requests for the keys to replicas for which master has a higher version from the ones obtained in the digests from the replicas. The read repair updates the key-value at replicas with the latest version from the master.

- Pros:
- Reduces network traffic from replicas when they do not have a higher version to send.
- Works efficiently even in case when updates are occurring concurrently when digests are requested.
- Simpler to implement.

**Cons:**
- If it is just desired to know that if the replicas and master are divergent for a given key range, this approach causes bigger messages (compared to Merkle trees approach) to be sent from the replicas containing the digest for each key in the requested range.

**Merkle Tree approach:**
- Each node maintains a Merkle tree for each of its key range.
- When the master needs to check for divergence with replicas, it request the hash value of the Merkle tree root from all replicas.
- If the hash of roots of both replicas and the master are equal, it means that they are consistent with each other.
- If the hash of roots of replicas and master are unequal, the master requests the hash of the root of the next subtree from the replicas. This process is repeated until a hash value is obtained from the replica which matches the hash value at the master. The idea is to reach to that root of a subtree of the Merkle tree after which the replicas diverge.

**Pros:**
- In case the replicas are not divergent, it is very efficient to know with just a simple comparison of Merkle tree root hash values from the replicas and master.

**Cons:**
- Might cause problems in the event when concurrent updates are happening in the key range for which the hash is requested by the master.
- Does not work if the replicas are missing some keys altogether due to message delays. In this case the Merkle trees would always mismatch completely as they would have been constructed upon different key bases.
- More complex to implement without any obvious benefits in Cloudy2 scenario.
- Additional overhead of Merkle tree maintenance during updates and leaving/joining of nodes in the system.
5.2 Optimizations

5.2.5 Partial Fetching

This optimization was introduced to speed up the retrieval of version associated with a DPI from the Cloudy2 storage layer (BDB store) in case of a digest query. The BDB store API provides the functionality to fetch partial keys or values from the store. When the store needs only to send the digest for a DPI, it does a partial fetch and fetches only the version of the value rather than the complete value associated to that DPI. The key and the version are used to create the digest DPI. Partial fetching could give huge performance gain in cases when DPIs contain huge values.

5.2.6 Serialization

When a DPI is received by the protocol, it either accesses the local store (if the node is in the preference list for that DPI) or it sends a get request to a remote node which contains the requested DPI. Both these access patterns are differentiated to prevent an additional round of serialization in case of remote access. In case of local access, the store sends the results as a Set<DPI> to the protocol. In case of remote access, the store sends the results as a Set<byte[]> where byte[] contains the serialized DPIs. The protocol or the messaging layer at the requesting node, holds the responsibility to deserialize this received Set<byte[]> into a Set<DPI>. This prevents an additional serialization step because in case of remote access, if the store still sends the result as a Set<DPI>, the messaging layer has to serialize those DPIs again to a bytes[] so as to send them to the requesting node.

5.2.7 Configuration

There are various parameters in the Cloudy2 Berkeley DB storage implementation configuration which can improve the overall throughput of the system. Some of these are:

- Cache Size : configures the cache size of BDB.
- Flush-Transactions : used to configure if it is desired to write and synchronously flush the log on transaction commit.
- Write-Transactions : similar to flush-transactions. If true, write but do not synchronously flush the log on transaction commit. Can be used to provide more durability than asynchronous transactions but higher performance than synchronous transactions.
- B+ tree fanout : configures the fanout of the B+ tree used to maintain the BDB index.
Chapter 6

Experiments and Benchmarks

Benchmarking and experiments were performed on the ETH bach cluster with each machine having the following configuration:

- Running Linux on 16 Intel(R) Xeon(R) CPUs (2.27 GHz).
- 24.0 GB main memory.

To allow external clients to make JDBC connections anywhere in the Cloudy2 cluster, each node, apart from running Cloudy2 daemon was also running a MySQL server instance plugged with the Cloudy2 storage engine. The Cloudy2 storage engine, upon SQL query request, communicates with the Cloudy2 daemon running on the same node. All values in the below experiments were rounded off to two places of decimal. Table 6.1 shows the fix configuration which was used during all experiments.

<table>
<thead>
<tr>
<th>Cloudy2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM heap size</td>
<td>16.0 GB</td>
</tr>
<tr>
<td>Logging level</td>
<td>WARN</td>
</tr>
<tr>
<td>External interface</td>
<td>Thrift</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BDB store</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Write transactions</td>
<td>false</td>
</tr>
<tr>
<td>Flush transactions</td>
<td>false</td>
</tr>
<tr>
<td>B+ tree fanout</td>
<td>4096</td>
</tr>
<tr>
<td>Batch Size</td>
<td>3.5 MB</td>
</tr>
</tbody>
</table>

Table 6.1: Fixed configuration for all experiments.

6.1 Microbenchmarks

Microbenchmarking was done with the customer and the address tables from the TPC-W [24] benchmark, both created with the Cloudy2 storage engine. InnoDB
[21] was used as a baseline for comparison.

Table schema:

- customer (c_id int not null, c_uname varchar(20) not null, c_passwd varchar(20), c_fname varchar(20), c_lname varchar(20), c_addr_id int, primary key(c_id), index(c_uname))

- address (addr_id int not null, addr_street1 varchar(40), addr_street2 varchar(40), addr_city varchar(30), addr_state varchar(20), addr_zip varchar(10), addr_co_id int, primary key(addr_id))

Table queries used in experiments:

- Insert: insert into customer values(...);
- Primary Key: select * from customer where c_id = x;
- Secondary Key: select * from customer where c_uname = y;
- Range Scan: select * from customer;
- Join Query: select * from customer, address where c_addr_id = addr_id;

Table datasets used in experiments:

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Customers records</th>
<th>Customer table size</th>
<th>Address records</th>
<th>Address table size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small dataset</td>
<td>2880</td>
<td>1.25 MB</td>
<td>5760</td>
<td>5.7 MB</td>
</tr>
<tr>
<td>Medium dataset</td>
<td>288000</td>
<td>127.6 MB</td>
<td>576000</td>
<td>56.7 MB</td>
</tr>
<tr>
<td>Large dataset</td>
<td>2880000</td>
<td>1293.53 MB</td>
<td>5760000</td>
<td>567.3 MB</td>
</tr>
</tbody>
</table>

Table 6.2: Experiment datasets.

Note:

- All queries were performed on the customer table. The address table was used only for the join queries with the customer table.
6.1 Microbenchmarks

- A separate bach machine was used to send the SQL query requests to the MySQL instances running in Cloudy2 cluster.

- CSR = Client side routing enabled which means that the Cloudy2 storage engine can directly contact the MySQL instance running on the Cloudy2 node which stores the desired data.

- IDB = InnoDB, BDB = Cloudy2 BDB store.

- Quorum\( (n,r,w) \) = Quorum protocol with replication factor \( n \), read quorum \( r \) and write quorum \( w \).

### 6.1.1 InnoDB vs Cloudy2 BDB Store

This experiment was done to compare the performance of InnoDB and Cloudy2 BDB store for the above queries. Cloudy2 was running as a single machine cluster with a BDB store. In this setup, the Cloudy2 thrift server on receiving the client thriftDPI, instead of going through the Cloudy2 protocol layer, directly accesses the store. This prevented any additional overhead caused due to the Cloudy2 protocol layer. Although when compared to InnoDB, the Cloudy2 BDB setup still has an additional overhead of thrift communication between the Cloudy2 storage engine and itself. InnoDB and Cloudy2 BDB cache size were set at 3.0 GB.

![Figure 6.1: InnoDB vs Cloudy2 BDB store](image)

From figure 6.1, it can be observed that for all datasets, InnoDB has better insert response time (almost one half) than Cloudy2 BDB store. The primary index read in both the setups is comparable for all benchmarked data sets even when Cloudy2 BDB includes a small thrift overhead. We can see that the secondary indexes really get slower with large dataset in case of Cloudy2 BDB. This is due to the fact that the large data set could not fit into the 3.0 GB cache of BDB and caused disk I/O. Unlike Cloudy2 BDB, InnoDB scaled well for index reads even with a large dataset.
Figure 6.2 shows that Cloudy2 BDB beats the InnoDB response time for range queries. This is due to the fact that range scan is highly optimized in Cloudy2 BDB store. The Cloudy2 BDB stores the index in a B+ tree and a range scan involves a sequential scan of the range, preventing any random reads. The join query always performed much faster with InnoDB as compared to Cloudy2 BDB. The reason is that InnoDB primary key joins are highly optimized as the data is clustered together with index. On the other hand Cloudy2 BDB store is currently not optimized to run join queries.

6.1.2 Database Level Partitioning

Following experiments were done to measure the response time for the above queries when complete tpcw database resides on a single Cloudy2 node. This node was one of the nodes from a five node Cloudy2 cluster on which these experiments were performed. Experiments were done with Cloudy2 BDB store and different settings of quorum protocol (n,r,w), with and without client side routing. During these experiments, the tpcw database contained a customer and an address table. The values shown in below experiments were averaged over three benchmark runs. During each benchmark run, a single thread executed fifteen random select queries of each type (primary, secondary, range, join).

Small Dataset

Figure 6.3 shows that the insert response time for the customer table with Quorum(1,1,1) CSR settings was about one half with that of Quorum(3,2,2) CSR setting. Similar behavior is seen with Quorum(1,1,1) and Quorum(3,2,2). This is due to the fact that for an insert in Quorum(3,2,2) setting, the protocol waits for acknowledgements from at least two replicas (including itself) as opposed to the Quorum(1,1,1) case where the protocol simply waits for the acknowledgement from its local store. As expected, both Quorum(1,1,1) and Quorum(3,2,2) with CSR performed better than their non CSR setting where it involves an additional
6.1 Microbenchmarks

![Diagram showing response time comparison for insert and index query](image1)

Figure 6.3: Response time comparison for insert and index query

Routing overhead to the master node storing the customer table. For insertion, InnoDB on an average took only one third of the time taken by the Quorum(1,1,1) CSR setup. This is because InnoDB has a better response time than Cloudy2 BDB for inserts as seen from the last experiment (figure 6.1) and also the Cloudy2 quorum protocol introduces an additional protocol layer overhead. The behavior of index read (primary and secondary) is similar to the insertion case. The experiments with client side routing enabled (CSR) gave comparable response time to InnoDB. Quorum(3,2,2) is slower than Quorum(1,1,1) since the protocol layer has to wait for additional messages from the replicas before sending data to the client.

![Diagram showing response time comparison for range scan and join query](image2)

Figure 6.4: Response time comparison for range scan and join query

From figure 6.4 we see that the range scan with Quorum(1,1,1) CSR or Quorum(3,2,2) CSR setting gives comparable response time to InnoDB. However, as explained before, joins are very slow in all Cloudy2 settings when compared to InnoDB.
Medium Dataset

![Bar chart](chart1.png)

Figure 6.5: Response time comparison for insert and index query

The medium dataset experiment (figure 6.5) shows the same behavior as seen with the small dataset.

Figure 6.6 shows that, range scans in Quorum(1,1,1) and Quorum(3,2,2) CSR setting performed better than InnoDB due to the range scan optimizations in Cloudy2 BDB store.

![Bar chart](chart2.png)

Figure 6.6: Response time comparison for range scan and join query

Large Dataset

A major slowdown in secondary index read response time can be seen in figure 6.7. This can be explained due the large dataset size which overshot the Cloudy2 BDB cache size. Although the combined size of the customer and address tables was approximately 1.8 GB, the actual size of the table records stored as DPIs in Cloudy2 BDB store was more than 2.5 GB. And since the 3.0 GB cache size of
6.1 Microbenchmarks

BDB is shared among both its primary index and secondary index, it resulted in cache misses thereby increasing the average response time. Owing to this, during range scans and joins with large dataset, high delays and timeouts were seen with all Cloudy2 protocol settings. Most of the range scan and join queries returned incomplete results. We can say that unlike InnoDB, BDB index read response time starts increasing as soon as the dataset goes out of its cache and it needs disk I/O to fetch it.

The next experiment in figure 6.8 was done by increasing the cache size of Cloudy2 BDB store to 6.0 GB so that the data set can reside completely in the cache. It can be seen that primary index read and range queries response time was much better than previous experiment and back in numbers with the small and medium data set response time.

Figure 6.7: Response time comparison for insert and index query (cache = 3.0 GB)

Figure 6.8: Response time comparison for index read (cache = 6.0 GB)
Figure 6.9 shows the range scan response time which was able to return complete records with this increase in cache size. Like before the Cloudy2 BDB range scan now performs better than InnoDB.

![Figure 6.9: Response time comparison for range scan (cache = 6.0 GB)](image)

### 6.1.3 Key Level Partitioning

Following experiments were done to measure the response time for the queries on the customer table with horizontal partitioning in a five node Cloudy2 cluster. During Cloudy2 load balancing, the load balancer partitioned the table based on the record’s primary key which resulted in a spread of table records on multiple machines. Since the table was partitioned, CSR is not possible as the data might move before the next request from the client. The experiments do not contain the statistics about secondary index read because it is not supported in partitioning by key as explained in section 4.2.3. In all experiments, the cache size of InnoDB and Cloudy2 BDB store was kept at 3.0 GB. The values shown in below experiments were averaged over three benchmark runs. During each benchmark run, a single thread executed fifteen random select queries of each type (primary, range, join).

**Small Dataset**

In this experiment, since the dataset was small, it did not overload any Cloudy2 node and no data partitioning took place. CSR was enabled in this experiment and the client always contacted the Cloudy2 node storing the required data.

It can be observed in figure 6.10 that the insert response time has increased from the database level partitioning experiment (CSR enabled case in figure 6.3). This is because Cloudy2 load balancing needs load statistics to be gossiped in the cluster. These load statistics also include statistics about the number of DPIs stored on each node within different ranges. In current implementation, to get these DPI statistics, the store has to be traversed to count them. This decreases the put throughput when puts are being done concurrently into the system while statistics are being gathered.

The primary index read (figure 6.10), range scan and join response time (figure 6.11) was similar to the values obtained in the experiment with database level partitioning experiment since the table still did not get partitioned.
Figure 6.10: Response time comparison for insert and index query

Figure 6.11: Response time comparison for range scan and join query
Medium Dataset

During this experiment, the medium dataset increased the load on a single Cloudy2 node above the cluster average load. This lead to data partitioning, where the load balancer partitioned the data on two Cloudy2 nodes. The distribution of data depends on various factors like CPU load, memory load, token range of the node etc. and was not recorded during this experiment.

Figure 6.12: Response time comparison for insert and index query

In figure 6.12 there is a slight increase in the primary index read when compared to the database level partitioning case for medium dataset with CSR. This is again expected due to the data statistics calculation overhead during load balancing steps. No other difference in general behavior was seen for different protocol settings. It can be seen from figure 6.13 that the response time for range scans and joins has increased when compared to the database level partitioning CSR

Figure 6.13: Response time comparison for range scan and join query
case. This is understandable as now multiple Cloudy2 nodes have to be contacted to collect full table data.

**Large Dataset**

Since the dataset was large, the load balancing step partitioned the data over all five Cloudy2 machines. Again the data distribution pattern was not recorded in this experiment.

It can be seen in figure 6.14 that even with a large dataset, the system responded with similar response time for index reads as with small and medium dataset. This shows that the load balancing was able to distribute the load on a single store to the complete cluster by partitioning the data appropriately. This also implies that the total store cache size in the system was enough to cache the large data set and prevent any disk I/O during retrieval.

![Graph](image)

**Figure 6.14:** Response time comparison for insert and index query

Since index read response time improved with load balancing, no additional de-
lays or timeouts were seen during range scans and joins and complete results were returned. Figure 6.15 shows the response time for the range scan and join queries.

Results summary:

- In Section 6.1.1. it was found that the insertion response time in InnoDB is approximately half than the Cloudy2 BDB store. The Cloudy2 BDB index read response time is comparable to InnoDB only until the BDB cache can hold the dataset completely. As the dataset grows larger than the cache, unlike InnoDB, big overheads are seen with BDB.

- All experiments shows that range scans are either comparable or faster in Cloudy2 BDB store as compared to InnoDB, if the dataset does not get out of the store cache.

- InnoDB is highly optimized for joins and beats Cloudy2 BDB in all experiments.

- In all experiments, setups with client side routing (CSR) always performed better than non CSR setups.

- Figure 6.16 summarizes the observations for the Quorum(3,2,2) settings with database level partitioning (DLP) and key level partitioning (KLP) on different datasets. As explained in section 6.1.3, in some cases, with KLP there is a small increase in the insertion and index read response time due to statistics overhead. However, KLP could successfully partition the data and improve the index read response time for larger datasets. On the other hand with DLP, the index read response time increases with large dataset.

![Response time comparison for insert and index query](image)

Figure 6.16: Response time comparison for insert and index query

### 6.2 TPC-W Benchmark

Since, one of the goals of Cloudy2 SQL interface is to allow easy porting of SQL based RDBMS applications to cloud environments, we used the TPC-W bench-
6.2 TPC-W Benchmark

mark [24] to analyze the system behavior during high loads. Although this benchmark has been deprecated by the TPC organization, it is still widely used in both academia and industry. The TPC-W benchmark models an online bookstore with different kinds of requests such as searching for items, displaying item information, placement and tracking of customer orders, updating information (customer, items, author etc.). The current implementation of Cloudy2 SQL interface still lacks some performance optimizations and functionalities. These are described later in section 7.1. Due to this, timeouts (according to TPC-W response time constraints) were seen for queries involving multiple table joins. For example, to get a customer with complete address information a join between the customer, the address and the country table is required. This did not allow the full benchmark to run successfully. However, the TPC-W queries which do not involve complex joins and other missing functionalities of Cloudy2 SQL interface, were used to complete this experiment. In this experiment, the load on a cluster of five Cloudy2 nodes was increased by increasing the number of concurrent MySQL clients. The Cloudy2 cluster hosted the complete tpcw database with database level partitioning and a Quorum(3,2,2) protocol setting.

Table 6.3 shows the sizes of the various tables in the tpcw database used in this experiment.

<table>
<thead>
<tr>
<th>Table name</th>
<th>#Records</th>
<th>Size(KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country</td>
<td>92</td>
<td>2.38</td>
</tr>
<tr>
<td>Authors</td>
<td>250</td>
<td>90.92</td>
</tr>
<tr>
<td>Items</td>
<td>1000</td>
<td>529.78</td>
</tr>
<tr>
<td>Address</td>
<td>57600</td>
<td>5673.56</td>
</tr>
<tr>
<td>Customer</td>
<td>28800</td>
<td>12602.7</td>
</tr>
<tr>
<td>Orders</td>
<td>25920</td>
<td>1758.04</td>
</tr>
<tr>
<td>Orderlines</td>
<td>77799</td>
<td>6537.56</td>
</tr>
<tr>
<td>CCXacts</td>
<td>25920</td>
<td>2058.22</td>
</tr>
</tbody>
</table>

Table 6.3: tpcw database size

The MySQL clients used in the experiment executes the following sequence of TPC-W queries 1.

1. Get item
   This query returns the complete item description and author details for a given item id. This query is executed as a join between the item and the author table.

2. Get related items
   This query returns all five related items of a given item. This query is executed as a join between the item and the item table.

3. Get bestseller
   This query returns item description and author details of fifty random items. This query is executed as a join between the item and the author table.

1The order of the queries in the sequence is random and is of no significance
4. Get customer
   This query returns the customer details for a given user name. This query
   is executed as a secondary index query.

5. Get new items
   This query returns the item description and author details for a maximum
   of fifty most recent items order by their titles. This query is executed as a
   join between the item and the author table and orders the results by their
   publication date and titles.

6. Search author
   This query returns the author description and at most fifty of his authored
   items. This query is executed as a join between the item and the author
   table and search for author last name using the SQL LIKE operator.

7. Insert order
   This inserts a new order in the orders table using a normal insert query.

8. Update item
   This updates the item information a given item using an update query.

9. Update item stock
   This updates the stock information of a given item using an update query.

6.2.1 System Behavior under Load

Figure 6.17 shows the effect of increasing load on Cloudy2 over SQL query re-
response time. The response time shown in the graph is averaged on the basis of per
query average response time from three benchmark runs. During one benchmark
run, each MySQL client executes the above nine query sequence ten times. It can
be observed from the graph that the response time of a SQL query increases lin-
early with the increase in concurrent MySQL clients. The throughput of Cloudy2
system increases rapidly with the increase in the number of concurrent clients
and then saturates around 440 kB/s. This is because with the initial increase
in the number of clients, Cloudy2 could concurrently process more requests per
second. However, after certain number of concurrent connections, Cloudy2 could
not handle more client requests in a concurrent manner thereby saturating the throughput of the system. Once the total number of concurrent clients increased over 350, no more clients could make connections with the MySQL instances running on Cloudy2 nodes. Also, Cloudy2 protocol timeouts were observed for some requests after 320 concurrent clients because of the increased contention for shared resources such as Cloudy2 store locks.
Chapter 7

Conclusions

Increased business costs of software and services have made cloud computing a very attractive option for resource management. Nowadays, a huge variety of cloud storage systems is available, all with different guarantees and functionalities. This thesis describes Cloudy2 which is a highly modular cloud storage system. It aims to provide a generalized architecture with a data model to support different use cases. Cloudy2 has been designed to work efficiently under high load conditions with high availability and scalability. Flavio Pfaffhauser’s thesis [25] presents Cloudy2 as a highly-scalable key-value store. Stephan Merkli’s master thesis [26] on "Streaming in the Cloud", implements a streaming service use case over Cloudy2 which was achieved by exchanging its various components. Implementation of the SQL query interface described in this thesis is another such use case accomplished through the modular design and generalized data model of Cloudy2. Users can use this interface for data storage in Cloudy2 by plugging the Cloudy2 storage engine to MySQL. The performance results show that range scans are highly optimized in Cloudy2 BDB store and in most cases outperform InnoDB. The partitioning experiments show that Cloudy2 is able to partition the data, thereby successfully distributing the load on multiple nodes. This data partitioning also helps in achieving better response time with BDB store which was previously degrading with larger datasets on a single node. The work in this thesis shows that the SQL interface can be used with different Cloudy2 protocol configurations. Each protocol configuration offers distinct guarantees and performance. A user can configure Cloudy2 according to his needs to obtain the desired behavior. If the desired guarantees are not met by the system, Cloudy2 also allows the user to extend it according to his requirements. With this SQL query interface, Cloudy2 aims to fill in the missing SQL functionalities in modern cloud storage services. The TPC-W benchmark shows that the Cloudy2 SQL interface is able to port non-trivial real world database applications to cloud based environments. Although, there are still some optimizations and functionalities which are missing from the current version of the SQL interface, it is the right step towards its goal of allowing easy portability of full database applications to cloud environments.

7.1 Future Work

The SQL interface for Cloudy2 and the Cloudy2 system are still in development stage. There are various functionalities and performance enhancements which
could not be implemented in the scope of this thesis. This section briefly describes those functionalities and improvements:

### 7.1.1 Transaction Support

Cloudy2 currently does not support transactions. Since Cloudy2 SQL interface aims for easy migration of RDBMS based applications to cloud environments, a support for transactions would be desired. However, cloud environments are not really friendly for providing ACID guarantees. So, the particular challenge would be implement a set of restricted functionalities which can still provide acceptable performance for the queries.

### 7.1.2 Cloudy2 Storage Engine

The Cloudy2 storage engine has a wide scope of performance optimizations. Some of these desired optimization are mentioned below:

- **Condition pushdown**: MySQL provides different mechanisms to storage engines for optimization of data storage and retrieval process. One of these mechanisms is the condition pushdown which pushes down the SQL query conditions to the storage engine level. In Cloudy2 scenario, since the data is distributed over a network of nodes, filtering the desired data at each respective node would greatly reduce the network traffic and provide better response times.

- **Optimizer hints**: A MySQL storage engine can provide information to the MySQL server for optimization of the query. Cloudy2 storage engine chooses default implementation of these optimization methods. For example, `info()`, `position()`, `rnd_pos()` etc. Implementation of these optimization methods might give a performance boost for the SQL queries.

- **Locking level**: The current version of Cloudy2 storage engine uses table level locks. Implementing a lower locking level like row level would greatly improve the performance of concurrent queries on same tables.

- **Multi-Column indexes**: Cloudy2 storage engine currently does not support multi-column indexes functionality. This might be desirable for real world applications like the TPC-W benchmark.

### 7.1.3 Cloudy2 Storage Layer

Cloudy2 SQL interface uses BDB store for data storage. During the experiments, it was found that BDB store did not scale well as dataset gets larger than its cache size. The BDB store would need more performance tuning and benchmarking to confirm this claim. Since, InnoDB was found to scale well with increasing dataset, it might be useful to implement and benchmark a Cloudy2 storage layer which uses InnoDB.

### 7.1.4 Full TPC-W Benchmark

Due to the absence of storage engine performance optimizations (e.g. condition pushdown, optimizer hints) and some missing functionalities (multi columns keys),
the TPC-W benchmark could not be fully executed. During benchmarking, many queries involving joins of multiple tables, timed out. It would be desired to explore the benchmark a bit more to optimize its queries apart from the implementation of missing functionalities in the Cloudy2 storage engine.

7.1.5 Failure Scenarios

Analysis of the performance and impact on SQL queries during node failures was out of the scope of this thesis. Since Cloudy2 aims to build a highly scalable and available storage systems, the SQL interface needs to be tested and benchmarked during node failures.
Bibliography


