Building a High Performance Key Store for Petascale Device Management in RESAR

Stephen A. Broeker, Ignacio Corderi, and Ahmed Amer
steve_broeker@yahoo.com, icorderi@cs.ucsc.edu, aamer@scu.edu

June 13, 2013

Abstract

Swift is a powerful and popular Cloud Storage implementation. The Swift protocol consists of the hierarchy of Cloud Items: {Account, Container, Object}. Each Cloud Item is stored on multiple Storage Pairs: {Server, Device}. Swift does not maintain device construction and parity. This critical infrastructure is left to the Administrator to manage outside of Swift. In this paper, we expand Swift to include device management: Swift RESAR. This extension greatly empowers Swift Administrators in managing large numbers of cloud devices. Swift uses the Python programming language and Swift RESAR extends Swift in a painless manner by also using Python. This paper also presents a new approach to processing data streams that is highly scalable: the stream star schema. This new type of star schema is proposed to accommodate high data stream rates: giga bits per second, by reducing insertion time to a constant. An experimental implementation of both star schema types on the RESAR data stream shows that stream star schema insertion performance is constant and superior to star schema insertion performance by a factor of over 1,000, which is 3 orders of magnitude. Our new database does not only excel in insertion performance, it also is superior in query performance. The minimum query time for the RESAR star schema is over 60 times faster than the MySQL database. The maximum query time for the RESAR star schema is over 562 times faster than the MySQL database.

1 Introduction

Swift is a Cloud Storage implementation, that is free open source software released under the terms of the Apache License. This project is managed by the OpenStack Foundation, a non-profit corporation established in September 2012. Swift is gaining wide popularity in that over 150 companies are currently participating in this project [7].

The Swift protocol consists of the hierarchy of Cloud Items: {Account, Container, Object} [5]. That is, Swift access is initiated by an Account, which contains Containers, which contains Objects. Thus a Swift Object is uniquely identified by the 3-tuple: {Account, Container, Object}.

Each Cloud Item is stored on multiple Storage Pairs: {Server, Device}. This mapping information is stored in Rings, which are actually Consistent Hashes.

For Storage Pairs, Devices are actually Logical Unit Numbers (LUNs). Thus, Swift is not concerned with LUN parity and reliability. That is, it does not identify and store LUN construction. Swift thus cannot describe how a LUN was created: the number of disk devices used by the LUN and the LUN type (mirrored or parity). It is up to the Swift Administrator to manage LUN construction on each device host, outside of Swift.

This presents tremendous problems when dealing with millions of devices in a cloud. For instance, problems with the tracking and organization of LUN construction. How should LUNs be created to minimize the effects of device failures? How should LUNs

be created to optimize LUN performance? How should LUNs be modeled to allow mathematical analysis?

In this paper, we expand Swift to include LUN construction. This not only increases the scope of the Swift Project but greatly empowers the Swift Administrator by simplifying LUN construction and management. This is of great significance when managing millions of devices in a Storage Cloud. An additional goal in this project was to minimize changes to the Swift code base. This greatly enhances new feature implementation. Additionally, this expansion brings LUN construction out of the shadows and into the open source lime light. This expansion will thus prove beneficial to the entire OpenStack community.

In addition, we leverage previous work done by Ignacio Corderi [1] on the subject of Cloud Device Management. Mr. Corderi describes his research as RESAR: Robust, Efficient, Scalable, Autonomous, and Reliable storage.

This paper is thus the application and implementation of an efficient, scalable storage methodology to the OpenStack Swift Project. We therefor refer to it as the Swift RESAR project.

2 RESAR

RESAR essentially is a mapping algorithm, that is centered around LUN construction [1]. Devices are segmented into Disklets, were there is a single disklet size for all devices in a given cloud. In RESAR, LUNs are referred to as Reliability Groups. A Reliability Group is a collection of data disklets and a single parity disklet. Data disklets can be members of multiple reliability groups. Parity disklets can only be in a single reliability group. So RESAR keeps track of devices, disklets, and reliability groups. Traditionally, disklets and reliability groups are organized into a grid, where each row and column defines a separate reliability group. The nodes on the bottom and right edges are parity disklets. All other nodes are data disklets. RESAR is unique in that it also organizes disklets and reliability groups into a mathematical dual form. Each reliability group can thus be identified as a vertex and all connecting edges, and each data element can be identified as an edge connecting two vertices. This approach allows the application of various graph techniques like coloring. Thus RESAR provides optimal device organization that is scalable.

Figure 1 provides an example of how reliability groups are organized and how a graph can be built for coloring to support data layout across disks. In the grid on the left data disklets numbered 1-16 and parity disklets lettered a-h. Each data disklet is in two reliability groups (the row and columns in the grid). Each reliability group has one and only one parity disklet. On the right hand side of Figure 1 the same data is expressed in its mathematical dual form. Each reliability group can be identified as a vertex and all connecting edges, and each data element can be identified as an edge connecting two vertices. We can observe that data element 12 protected by parity blocks c and h can be represented as vertices c and h connected by edge 12.

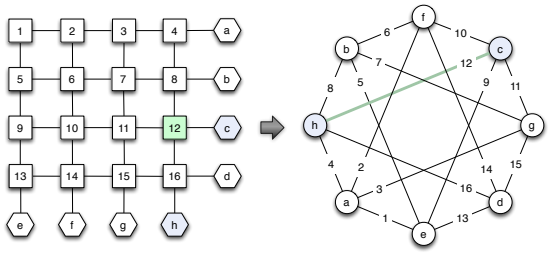

Figure 1: Left: RESAR small two-failure resilient array. Right: its design theoretical dual.

3 Swift RESAR

We would now like to describe how RESAR was implemented in Swift. During our research, we realized that there were essentially two approaches available. The first approach emphasized leveraging existing code in the Python community. Swift is implemented in the Python programming language. So the Swift RESAR project is also implemented in Python. So for the first approach, the primary goal was to minimize the amount of new code thus resulting in timely results. On the other hand, the second ap-

proach emphasized performance. In this approach, we were willing to write new code as long as it resulted in better RESAR performance.

4 RESAR MySQL

We will now describe the first approach to Swift RESAR. During this research, we quickly realized that the RESAR metadata (devices, disklets, reliability groups) needed to be stored in a database. A database is an established and optimal manner of storing persistent data that needs to be updated and queried. So for the first approach we chose MySQL [4] as the database since it is well established, free, easy to install, and has a Python interface. Figures 5 - 9 present the RESAR MySQL database. So the MySQL RESAR database consisted of five tables: {MetaData, Device, Disklet, ReliabilityGroup, ReliabilityGroupsDisklets}.

The MetaData Table contains a single row and is used to store the attributes: {CreateTime, DiskletSize}.

The MetaData Table is created when the database is initialized and is not changed henceforth.

The Device Table contains an entry for each device in a cluster. Its attributes are: {ID, CreateTime, HostName, DeviceName, DeviceStart, DeviceSize, InUse, NumDisklets}.

A device is uniquely identified by the tuple: {HostName, DeviceName}. To facilitate optimal device lookup, HostName and DeviceName are indexed. The InUse attribute allows a device to be taken out of service for maintenance purposes.

The ID attribute is required so that each Device Table row can be referenced from the Disklet and ReliabilityGroupsDisklets Tables.

The Disklet Table is used to keep track of disklet use in the cluster. Thus each device (in the cluster) results in multiple Disklet Table entries. The Disklet Table attributes are: {ID, DeviceID, DeviceIndex, Type}.

The ID attribute is required so that each Disklet Table row can be referenced from the ReliabilityGroupsDisklets Table. A disklet is uniquely identified by the tuple: {DeviceID, DeviceIndex}. To facilitate optimal device lookup, DeviceID is indexed. The Type attribute identifies that a disklet is used for "parity" or "data", but not for both. If a disklet is not used by a reliability group, then its type is irrelevant and the Type attribute is thus set to "none".

The ReliabilityGroup Table is used to manage reliability groups in the cluster. Its attributes are: {ID, CreateTime, NumDisklets, InUse}.

The ID attribute is required so that each ReliabilityGroup Table row can be referenced from the ReliabilityGroupsDisklets Table. The InUse attribute allows a reliability group to be taken out of service for maintenance purposes.

The ReliabilityGroupsDisklets Table is used to map disklets to reliability groups. Its attributes are: {ID, ReliabilityGroupID, DiskletID, DiskletType, DeviceID}.

The ID attribute is required so that each ReliabilityGroup Table row can be externally referenced. The DiskletType attribute indentifies that a disklet is used for "parity" or "data", but not for both. A ReliabilityGroupsDisklets Table entry is uniquely identified by the tuple: {ReliabilityGroupID, DeviceID, DiskletID}. To facilitate optimal table lookup, ReliabilityGroupID, DiskletID, and DeviceID are indexed.

It is of the utmost importance to employ indexing for many of the database attributes. Indexing helps optimize table query performance.

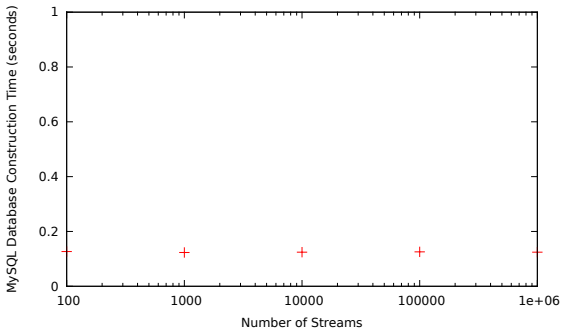

Figure 2: MySQL construction time for a given number of disk devices.

Figure 2 shows database construction time for a given number of disk devices. So for a cloud cluster

of 1 million devices, database construction time was (on average) 0.12 seconds for a single device and reliability group. It required over 34 hours to create the entire database of 1 million devices and 1 million reliability groups.

We also tested query times for the same MySQL database. Table 1 presents the results. The minimum query time was more than 0.0002 seconds. The maximum query time was less than 0.0005 seconds.

We sincerely hoped that this database creation time was excessive and that the second RESAR Swift approach would greatly improve database performance.

5 Star Schema

We were thus highly motivated to pursue the second RESAR Swift approach. That is, focus on performance and thus be willing to write new code. So we needed to create a new kind of database that was optimized for database construction.

OnLine Transaction Processing (OLTP) commonly uses relational databases to perform transactions [2, 3]. Many relational databases are implemented as star schema databases. A star schema is optimized to minimize string space - all string attributes are stored in separate dimension tables. Each dimension table is sorted to optimize query performance. Dimension table insertion time thus depends on the table size and is $O(\log n)$ where n is the number of records in a table. Star schema insertion time then, is the sum of all dimension table insert times $O(\sum_{1 \le i \le l}(\log n_i))$ where l is the number of attributes in the database and n_i is the number of values for attribute i.

We will no describe the RESAR star schema. The MetaData Table is a stand alone fact table in that it does not contain external references nor is it externally referenced. The fact tables are Disklet and ReliabilityGroupsDisklets. The Disklet Table references the Device Table. The ReliabilityGroupsDisklets Table is the most complicated and thus most interesting in that it references multiple tables: {ReliabilityGroup, Device, Disklet}. This schema thus shows that the dimension tables in the RESAR star schema are: {ReliabilityGroup, Device, Disklet}. But these dimension tables are NOT typical star schema string dimensions. They are in fact complex data structures. This would seem to indicate that string manipulation was not the cause of poor dimension table performance.

So what was causing the poor insertion performance? We theorized that the problem lies in how database attributes are indexed. Essentually all of the tables needed some of the attributes to be indexed so that query performance was reasonable. For example, a common query on the Device Table would be finding all devices supported by a given HostName. Or finding the entry for a given HostName and DeviceName.

In MySQL, index tables are B-trees. Index table insertion time thus depends on the table size and is $O(\log n)$ where n is the number of entries in a table. We felt that this insertion time could be minimized by the judicious use of hashing.

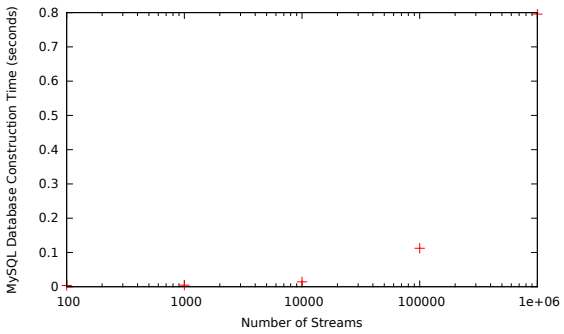

Figure 3: MySQL construction time for a given number of disk devices with memory tables.

To further substantiate this claim, we decided to conduct an additional MySQL experiment with memory tables. The previous MySQL tables were created on disk. It was possible that the poor MySQL performance was the result of disk IO. If this was true, then using memory tables should help eleviate the problem. So we did just that. The results were startling to say the least and they are presented in Figure 3. It required over 5 days to create the entire database of 1 million devices and 1 million reliability groups using memory tables in MySQL. These results show that

disk IO was not the cause of the poor MySQL performance. But rather that index table construction was the root of the problem.

We thus had the theoretical motivation to persue a new type of database.

6 Stream Star Schema

In this section, we define the *stream star schema* that will result in a database that is optimized for insertion performance. This performance improvement will primarily be the result of the improved attribute indexing mechanism.

Many relational databases like MySQL use B-trees to implement attribute indexes. Index table insertion time thus depends on the table size and is $O(\log n)$ where n is the number of entries in a table.

For the stream star schema, we proposed using Hash Tables instead of B-Trees. Hash table insertion time is a constant and thus not dependent on table size. But the problem was how to implement hashing in our new database.

The answer was in the judicious use of Python dictionaries. In Python, arrays or tables can be defined as lists or dictionaries. Lists are arrays that are indexed by position. Dictionaries are indexed by attribute name. Python dictionaries are analougous to data structures in the C programming language, where each attribute is identified by a unique name. In Python, dictionary attribute names are hashed. So a dictionary is actually a hash table.

Once we had solved the hashing problem, we needed to create a generic stream star schema engine that would allow custom databases to be created. We thus defined an XML Schema Definition (XSD) that is used to define the star schema database. This XSD allows for the creation of multiple tables. A table can include multiple attributes. Attributes have two qualities: type and index. Attribute type is one of: {int8, int16, int32, int64, uint8, uint16, uint32, uint64, string}. Attribute index is a boolean that allows attributes to be hashed. If attribute index is FALSE, then the attribute is defined as a simple data value. On the other hand if attribute index is TRUE, then the attribute will be defined as a dictionary of lists. That is, a hash table, where each entry is a list of table indecies that contain that particular attribute value.

For a given {stream star schema (SSS), fact table (F), dimension (D), dimension value (V), data row (R)} then the Python code to insert an attribute value into an attribute hash table is presented in Figure 4.

This code snippet clearly has constant performance time. Thus, performance for the stream star schema attribute indexing method was constant and thus optimal.

The final hurdle that we had to cross so that the stream star schema would be compatible and as capable as MySQL was how to store the database on disk? We envisioned writing a complex library that would flush a stream star schema from memory to disk and then allow the converse. Turns out that this problem has already been solved in Python. The solution is called "pickle" [6].

For a given file (F) and stream star schema (SSS), the Python code that shows how a stream star schema database is stored on disk is presented in Figure 5.

For a given file (F) the Python code that shows how a stream star schema (SSS) is populated from disk is presented in Figure 6.

Amazingly simple.

7 RESAR Stream Star Schema

Once the Stream Star Schema was defined, the Swift RESAR database was also implemented as a Stream Star Schema.

But there was a final piece to the puzzle. The stream star schema needed a query language, like SQL. Once again, Python came to the rescue. It turns out that manipulating Python dictionaries is a lot like executing SQL queries. Thus, in this case, Python code was remarkably similar to SQL. For example, consider the Python code that deletes a device from the RESAR MySQL database, given a device ID (DEVICE). This code is presented in Figure **??**.

Now consider the corresponding code that deletes a device from the RESAR stream star schema

```
if "index" in sss['fact_tables'][F]['dimensions'][D]:
    if V not in sss['fact_tables'][F]['dimensions'][D]['index']:
        sss['fact_tables'][F]['dimensions'][D]['index'][V] = []
        if R not in sss['fact_tables'][F]['dimensions'][D]['index'][V]:
            sss['fact_tables'][F]['dimensions'][D]['index'][V].append(R)
```

Figure 4: Python code that inserts an attribute value into an attribute hash table.

```
fd = open(F, "wb")
pickle.dump(SSS, fd)
fd.close()
```

Figure 5: Python code that stores a stream star schema database to disk.

database, given a stream star schema (SSS) and device ID (DEVICE). This code is presented in Figure **??**.

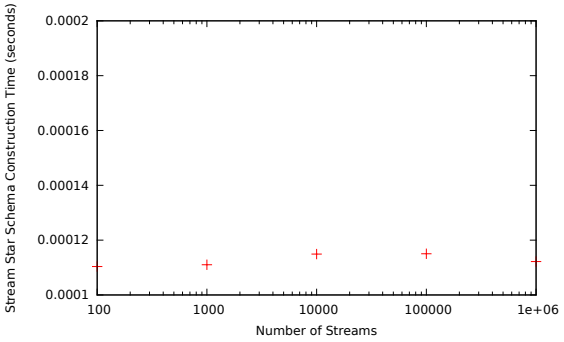

Figure 7: Stream Star Schema construction time for a given number of disk devices.

With all of the pieces in place, we could now construct our RESAR stream star schema database. Figure 7 shows database construction time for a given number of disk devices. So for a cloud cluster of 1 million devices, database construction time was (on average) 0.00011 seconds for a single device and reliability group. It only required less than 2 minutes to create the entire database of 1 million devices and 1 million reliability groups. These results were greatly significant when compared to the Star Schema. In fact, the Stream Star Schema creation time was on average 1,108 times faster than Star Schema creation. That is a order of magnitude of 3.

We also tested query times for the same RESAR star schema database. Table 2 presents the results. The minimum query time was greater than 0.0000005 seconds. The maximum query time was less than .0000065 seconds. Comparing the results from Table 1 and Table 2, we see that the minimum query time for the RESAR star schema was over 60 times faster than the MySQL database. We also see that the maximum query time for the RESAR star schema was over 562 times faster than the MySQL database.

8 Conclusion

We have described and demonstrated a powerful extension to Swift cloud storage: Swift RESAR. This facility greatly empowers Swift Administrators in managing large numbers of cloud devices. It also fully enables said administrators to employ mathematical models so that device reliability can be optimized. Previously, Swift ignored device management. Swift RESAR extends this project in a painless manner that is highly scalable. This project was implemented in the popular and powerful programming language - Python, thus enabling further development in this area.

This paper has also presented a new approach to processing data streams: the *stream star schema*. This new type of star schema is proposed to accommodate high data stream rates: giga bits per second,

```
fd = open(F, "rb")
SSS = pickle.load(fd)
fd.close()
```

Figure 6: Python code that populates a stream star schema database from disk.

by reducing insertion time to a constant. An experimental implementation of both star schema types on the RESAR data stream showed that stream star schema insertion performance is constant and superior to star schema insertion performance by a factor of over 1,000, which is 3 orders of magnitude.

Our new database does not only excel in insertion performance, it also is superior in query performance. The minimum query time for the RESAR star schema was over 60 times faster than the MySQL database. The maximum query time for the RESAR star schema was over 562 times faster than the MySQL database.

References

[1] I. Corderi, D. D. E. Long, T. M. Kroeger, and T. Schwarz, "RESAR storage: a system for two-failure tolerant, self-adjusting million disk storage clusters," tech. rep., University of California, Santa Cruz, Santa Cruz, California, 2012.

[2] R. Kimball, "www.ralphkimball.com."

[3] R. Kimball and J.Caserta, *The Data Warehouse ETL Toolkit*. Wiley, 2004.

[4] MySQL.

[5] OpenStack, "Swift 1.7.6-dev documentation."

[6] Python, "pickle python object serialization."

[7] Wikipedia, "Openstack," October 2012.

Database Table Name	Average Query Time (seconds)
Device Table by (ID)	0.00038575144
Device Table by (HostName, DeviceName)	0.00045933403
Disklet Table by (ID)	0.00028206711
Disklet Table by (DeviceID)	0.00031838307
ReliabilityGroup Table by (ID)	0.00027287826
ReliabilityGroupsDisklets Table by (ID)	0.000272622203333
ReliabilityGroupsDisklets Table by (ReliabilityGroupID)	0.00035928513
ReliabilityGroupsDisklets Table by (DiskletID)	0.000347662563333
ReliabilityGroupsDisklets Table by (DeviceID)	0.00038096357
Minimum Query Time	0.000272622203333
Maximum Query Time	0.00045933403

Table 1: MySQL query times using 1 million devices.

Database Table Name	Average Query Time (seconds)	MySQL/SSS Ratio
Device Table by (ID)	.000006425888	60
Device Table by (HostName, DeviceName)	.00000221678	207
Disklet Table by (ID)	.000000501782666667	562
Disklet Table by (DeviceID)	.000001366214	233
ReliabilityGroup Table by (ID)	.000000525165	519
ReliabilityGroupsDisklets Table by (ID)	.000000524541	519
ReliabilityGroupsDisklets Table by (ReliabilityGroupID)	.000001360337	264
ReliabilityGroupsDisklets Table by (DiskletID)	.000000857825333333	405
ReliabilityGroupsDisklets Table by (DeviceID)	.000001395876	272
Minimum Query Time	.000000501782666667	60
Maximum Query Time	.000006425888	562

Table 2: Stream Star Schema query times using 1 million devices.