Aparavi Open Data Format

Document Purpose

After reading this document, a software engineer with C++ coding skills will understand how Aparavi software stores archived data. With this knowledge you will be able to develop programs to read your organization's archived data to reconstruct your archived files without the use of Aparavi's web application.

Suggested Preliminary Reading

Read the "APARAVI Storage Model Overview" white paper which describes Checkpoints, Snapshots, and Archives as well as topics on linking, pruning and policies.

https://www.aparavi.com/resources/whitepapers/

Contents

Tags

Aparavi stores data in a structure called a Tag. A Tag is a type of container that includes both metadata (to describe the content and structure of the following data) and data.

Tags include the following

- ***Signature*** to designate that a new tag begins
- ***Tag Type*** to define the structure of the data in the tag
- ***Size*** to designate the variable length of the data
- ***Offset*** sometimes data is too big to be stored in a single tag, so it must be stored in multiple sequential tags with the offset tying the tags together

The complete Tag definition looks like the following.

signature
tag
size
offset
data []

An example Tag of type ODAT with an offset would look like the following

signature:	TAG-	signature:	TAG-
tag:	ODAT	tag	ODAT
size:	65,536	size:	3,406
offset:	0	offset:	65,536
data [65,536]		data [3,046]	

Tag Transformation

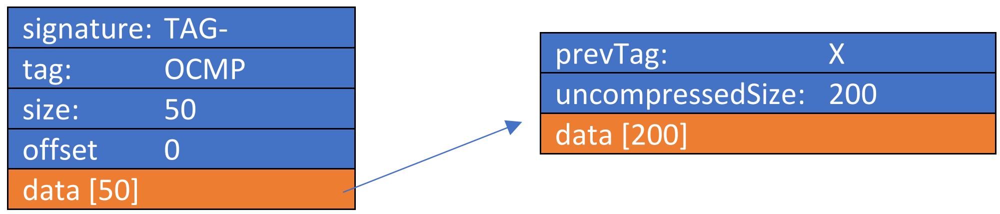
Tag transformations are used to transform data from one tag type to another. As example, a compression tag tells you the data was compressed, and an encryption tag tells you the data has been encrypted.

Compression Transformation

The following is an example of the compression transformation.
For example, a simple Tag of Type X is written with 200 Bytes of data in its original format.

signature:	TAG-
tag:	X
size:	200
offset	0
data [200]	

A compression transformation is then used to compress the original Tag into a compressed format producing a smaller size.



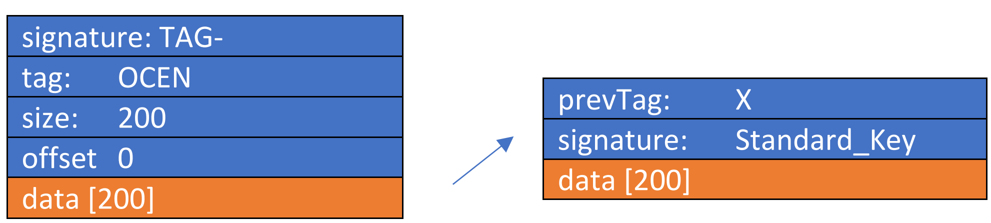
To retrieve the original data, you would first need to read data member of TAG-OCMP for 50 bytes (no offset in this case). Each tag type has an associated structure. In this case OCMP uses struct TAG_DATA_COMPRESSED_INFO. As such, you need to cast the data bytes to a struct pointer of type TAG_DATA_COMPRESSED_INFO.

The TAG_DATA_COMPRESSED_INFO has three members: prevTag, uncompressedSize, and data.

The prevTag member (4 bytes) will be “X” (the original tag type), the uncompressedSize (4 bytes) will be “200”, and the rest of the structure (50-4-4 = 42 bytes) contains the compressed data. To decompress the data, use LZ4 decompression to transform the data back to its original form to Tag type X.

Encryption Transformation

Another common transformation is the Encryption transformation. As is the case for all transformations, the encryption transformation uses the same mechanisms as the compression transformation. For the encryption transformation the OCEN tag type is used with an associated data struct type of TAG_DATA_ENCRYPTED_INFO.



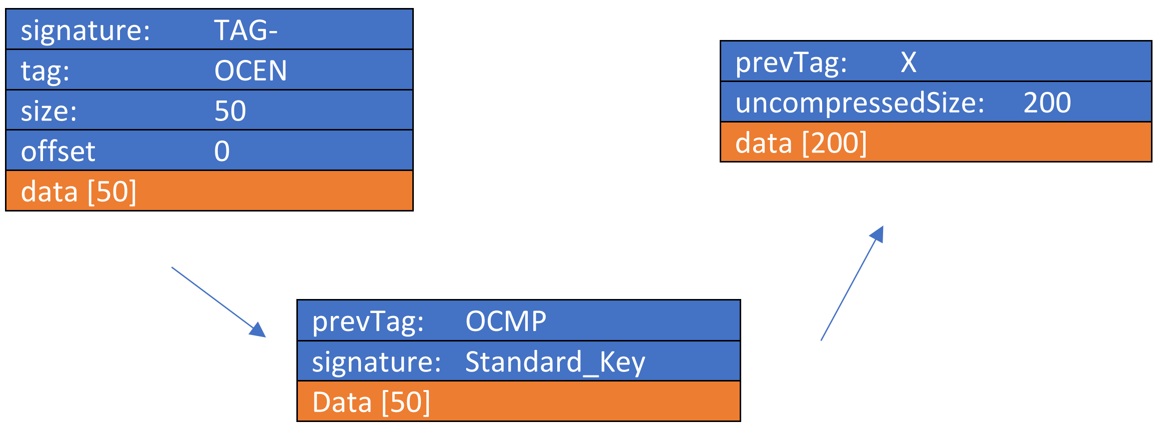
To retrieve the original data, you would first need to read the data member of TAG- OCEN for 200 bytes from the offset (in this case 0). Then cast the bytes to a struct pointer of type TAG_DATA_ENCRYPTED_INFO.

The TAG_DATA_ENCRYPTED_INFO has three members: prevTag, signature, and data.

The prevTag member (4 bytes) will be “X”. The signature (4 bytes) will be the CRC32 of the encryption key name (not the pass phrase) which was used to encrypt the data. The rest of the structure contains the encrypted data. To decrypt the data, use AES256 decryption with the passphrase associated with the encryption key. This will transform the data back to its original form to Tag type X.

Multiple Transformations

Multiple transformations can be strung together. A typical multi-transformation combination is compression followed by encryption.



To retrieve the original data, you would first need to read the data member of TAG-OCEN for 50 bytes from the offset (in this case 0). Then cast the bytes to a struct pointer of type TAG_DATA_ENCRYPTED_INFO.

The prevTag member (4 bytes) will be type OCMP. The signature (4 bytes) will be the CRC32 of the encryption key name (as is the case when the encryption key transformation is used by itself). Once you decrypt the data, the final step is to perform the decompression transformation as described above.

Data Retrieval Process Flow

To retrieve all the data out of the Aparavi data files, follow the steps highlighted below.

1. Read the data file (stored in binary format) tag by tag. Each tag will start with “TAG- “signature
2. For tags that contain data (indicated by a non-zero size), cast the data member into the relevant structure identified by the tag type
3. Perform the type-specific required algorithm (e.g., decompression, decryption …)
4. Repeat steps 2 and 3 for all additional transformations until you get back to the original data
5. Repeat the process until there are no tags left in the file

Technical C++ Implementation

Aparavi uses C++ structs to store the tags. For a list of all the tag types and their associated structure, please see the appendix.

As example, the struct for _tagTAG_ITEM is defined as follows:

```
typedef struct _tagTAG_ITEM {
      dword signature; // the signature - (TAG_ITEM_SIG)
      dword tag;       // the tag type
      dword size;      // the size of following data
      dword reserved;  // reserve for future use
      qword offset;    // the offset of following data
      byte data[1];    // the data array
} TAG_ITEM;
```

Tags are stored as TAG_ITEM after TAG_ITEM until the end of the file.
Members of the structure have specific meaning and usage defined as:

- **signature**: validator that a new tag starts, defined as:
  ```
  #define    TAG_ITEM_SIG    0x2D474154
  ```
- **tag**: used to define structure of the data found in the data member. Each tag type has a specific data structure.
- **size**: size in bytes of the data field
- **offset:** needed when the data member value exceeds the maximum size allowed and must be stored in multiple sequential tags

Beginning Information (CBEG) Example Tag

Let's look at an example of how a single tag is stored. For this example, the tag type is CBEG (component begin), with a data size of 8, and an offset of 0.

signature: remember that for all tags the signature is

```
#define    TAG_ITEM_SIG   0x2D474154
```

The hex value is written to disk in reverse order in little-endian form. Converting the hex value of the TAG_ITEM_SIG, you will get:

```
0x54 = 'T'
0x41 = 'A'
0x47 = 'G'
0x2D = '-'
```

tag: observe how the tag type CBEG (component begin) is stored as hex:

```
0x43 = 'C'
0x42 = 'B'
0x45 = 'E'
0x47 = 'G'
```

size: 8
offset: 0
data: the data member will begin after 24 bytes since there are four dword members of 4 bytes each and one qword member at 8 bytes (4 x 4 + 8 = 24).

The data member has a predefined structure based upon the tag type. As example, the CBEG data structure is defined as:

```
typedef struct _tagTAG_COMPONENT_BEGIN_INFO {
      dword    componentId;       // the offset within the stream of the component
      dword    componentFlags;    // flags for this component
} TAG_COMPONENT_BEGIN_INFO;
```

Using a binary viewer, let's examine how a CBEG tag with 8 bytes of data (i.e., two dword types) as defined in TAG_COMPONENT_BEGIN_INFO might look like.

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	54	41	47	2d	43	42	45	47	08	00	00	00	fa	7f	00	00	TAG-CBEG....ú...
00000010	00	00	00	00	00	00	00	00	01	00	00	00	01	00	00	00	....

Tag data is stored as:

- 54 41 47 2d = signature as TAG-
- 43 42 45 47 = tag type as CBEG
- 08 00 00 00 = size of the data member at 8 bytes
- fa 7f 00 00 = reserved
- 00 00 00 00 00 00 00 00 = data offset. Since this is the first tag and limited at 8 bytes, it is 0
- 01 00 00 00 01 00 00 00 = data for CBEG as two dwords (2 x 4 bytes = 8 bytes)

Compression (OCMP) Example Tag

Aparavi uses LZ4 compression to minimize storage. As explained above, one tag can embed another tag via the associated transformation logic.

Many metadata tags contain very little data. There is no benefit to compress these metadata tags. Because the tag type defines the data structure that is used for the data array, this means that some tags will always be stored without compression. For example, the tag type CBEG has a data size of 8 and won't be compressed.

Let's examine a tag that has large enough data for compression to be used. Tag OGEN (generic object) contains information about the file being stored. The data struct is defined as:

```
typedef struct _tagTAG_OBJECT_GENERIC_INFO {
       qword   fileSize;       // Object size
       qword   accessTime;     // Last file access time
       qword   modifiedTime;   // Time of last modification
       bool    isDirectory;    // Is object a directory?
       utf8    name[1];        // Name
} TAG_OBJECT_GENERIC_INFO;
```

For this example, we store and compress the object named 'myFolder'.

The data structure for the compressed TAG is:

```
typedef struct _tagTAG_DATA_COMPRESSED_INFO {
	dword	prevTag;		// previous tag
	dword	uncompressedSize;	// original uncomp size
	byte	data[1];		// lz4 compressed data
} TAG_DATA_COMPRESSED_INFO;
```

Observe how the OCMP (object compression) tag is written to disk:

```
000001f0 54 41 47 2d 4f 43 4d 50 1d 00 00 00 cc cc cc cc TAG-OCMP....ÌÌÌÌ
00000200 00 00 00 00 00 00 00 00 4f 47 57 4e 1c 00 00 00 ........OGWN....
00000210 c7 4e 21 7d 81 e6 d1 d3 01 10 e9 3a 88 08 00 50 ÇN!}.æÑÓ..é:ˆ..P
00000220 01 10 00 00 00 54 41 47 2d 4f 41 4c 54 14 00 00 .....TAG-OALT...
00000230 00 cc cc cc cc 00 00 00 00 00 00 00 00 03 00 00 .ÌÌÌÌ...........
00000240 00 02 00 00 00 c0 00 00 00 00 00 00 00 00 00 00 .....À..........
```

Tag data is stored as:

- 54 41 47 2d = signature as TAG-
- 4f f3 f3 50 = tag type as OCMP (object compression)
- 1d 00 00 00 = size of the data member at 1d hex = 29 bytes
- cc cc cc cc = reserved
- 00 00 00 00 00 00 00 00 = data offset. Data stored in one tag. So no offset
- 4f 47 57 4e = data

Snapshot 'Dump' Example

As you have seen from the previous examples, each tag holds specific information with a designated structure specific to the tag type. The data within that structure could be the transformation of another tag and so on. Showing a binary viewer screenshot containing all the tags in a snapshot is not practical and quite redundant.

To visualize all the tags and their data, Aparavi includes a 'Dump' utility. Dumping a snapshot with a single file would look like the following:

```
-------------------------------------------------------------
Dumping DATAFILE://SN0000000000/PATCH
-------------------------------------------------------------
          0 (0008): TAG_COMPONENT_BEGIN (componentId=1, flags=00000001)
         32 (0000):     TAG_BEGIN
         56 (0000):     TAG_METADATA_BEGIN
         80 (0011):         TAG_METADATA_TIMESTAMP (timestamp=1524074990)
        115 (0013):         TAG_METADATA_OUTPUT (output=SN0000000000)
        152 (0000):     TAG_METADATA_END
        176 (0008): TAG_COMPONENT_END (phyBeginComponentPos=0)
          0 (0008): TAG_COMPONENT_BEGIN (componentId=2, flags=00000000)
         32 (0008):     TAG_OBJECT_CONNECTOR (connector=FILE://)
         64 (0008):     TAG_OBJECT_PATH (path=C:/Test)
         96 (0108):     TAG_OBJECT_BEGIN
                                 Name                : myFolder
                                 Flags               : Container, Cache Self, Cache Base
                                 Attributes          : 00000010
                                 Size                : 0
                                 Generation          : 1.0
                                 This                : SN0000000000/2
                                 Link                : SN0000000000/2
                                 Base                : SN0000000000/2
        228 (0036):         TAG_DATA_COMPRESSED (size=36, offset=0)
                                 Original tag        : OGEN
                                 Uncompressed size : 37
        288 (0029):         TAG_DATA_COMPRESSED (size=29, offset=0)
                                 Original tag        : OGWN
                                 Uncompressed size : 28
        341 (0020):         TAG_ALT_STREAM (size=20, offset=0)
        385 (0145):         TAG_DATA_COMPRESSED (size=145, offset=0)
                                 Original tag        : OALT
                                 Uncompressed size : 192
        554 (0020):         TAG_ALT_STREAM (size=20, offset=0)
        598 (0054):         TAG_DATA_COMPRESSED (size=54, offset=0)
                                 Original tag        : OALT
                                 Uncompressed size : 64
        676 (0004):     TAG_OBJECT_END (ccode=00000000)
        704 (0008): TAG_COMPONENT_END (phyBeginComponentPos=208)
          0 (0008): TAG_COMPONENT_BEGIN (componentId=3, flags=00000000)
         32 (0017):     TAG_OBJECT_PATH (path=C:/Test/myFolder)
         73 (0110):     TAG_OBJECT_BEGIN
                                 Name                : myFile.txt
                                 Flags               : Cache Self, Cache Base
                                 Attributes          : 00000020
                                 Size                : 3
                                 Generation          : 1.0
                                 This                : SN0000000000/3
                                 Link                : SN0000000000/3
                                 Base                : SN0000000000/3
        207 (0042):         TAG_DATA_COMPRESSED (size=42, offset=0)
                                 Original tag        : OGEN
                                 Uncompressed size : 39
        273 (0032):         TAG_DATA_COMPRESSED (size=32, offset=0)
                                 Original tag        : OGWN
                                 Uncompressed size : 28
        329 (0020):         TAG_ALT_STREAM (size=20, offset=0)
        373 (0124):         TAG_DATA_COMPRESSED (size=124, offset=0)
                                 Original tag        : OALT
                                 Uncompressed size : 172
        521 (0020):         TAG_ALT_STREAM (size=20, offset=0)
        565 (0003):         TAG_DATA_STREAM (size=3, offset=0)
        592 (0020):         TAG_ALT_STREAM (size=20, offset=0)
        636 (0054):         TAG_DATA_COMPRESSED (size=54, offset=0)
                                 Original tag        : OALT
```

Notice how many different tags are used to a store the folder “myFolder” with a single file “myFile.txt”. You can see that in this example encryption was not used (i.e., no encryption tags) and that compression was used (see TAG_DATA_COMPRESSED).

We can also visualize which tags are stored without compression after the decompression transformation is processed.

```
--------------------------------------------------------------
Dumping DATAFILE://SN0000000000/PATCH
--------------------------------------------------------------
         0 (0008): TAG_COMPONENT_BEGIN (componentId=1, flags=00000001)
        32 (0000):     TAG_BEGIN
        56 (0000):     TAG_METADATA_BEGIN
        80 (0011):         TAG_METADATA_TIMESTAMP (timestamp=1524074990)
       115 (0013):         TAG_METADATA_OUTPUT (output=SN0000000000)
       152 (0000):     TAG_METADATA_END
       176 (0008): TAG_COMPONENT_END (phyBeginComponentPos=0)
         0 (0008): TAG_COMPONENT_BEGIN (componentId=2, flags=00000000)
        32 (0008):     TAG_OBJECT_CONNECTOR (connector=FILE://)
        64 (0008):     TAG_OBJECT_PATH (path=C:/Test)
        96 (0108):     TAG_OBJECT_BEGIN
                            Name              : myFolder
                            Flags             : Container, Cache Self, Cache Base
                            Attributes        : 00000010
                            Size              : 0
                            Generation        : 1.0
                            This              : SN0000000000/2
                            Link              : SN0000000000/2
                            Base              : SN0000000000/2
       228 (0037):         TAG_OBJECT_GENERIC
                            Name              : myFolder
                            IsDirectory       : 1
                            Size              : 0
                            Access Time       : 1523486618
                            Modified Time     : 1523486618
       288 (0028):         TAG_OBJECT_WINDOWS
                            Created Time      : 131679602069021006
                            Access Time       : 131679602182121744
                            Modified Time     : 131679602182121744
                            Attributes        : 00000010
       341 (0020):         TAG_ALT_STREAM (size=20, offset=0)
       385 (0192):         TAG_ALT_STREAM (size=192, offset=0)
       554 (0020):         TAG_ALT_STREAM (size=20, offset=0)
       598 (0064):         TAG_ALT_STREAM (size=64, offset=0)
       676 (0004):     TAG_OBJECT_END (ccode=00000000)
       704 (0008): TAG_COMPONENT_END (phyBeginComponentPos=208)
         0 (0008): TAG_COMPONENT_BEGIN (componentId=3, flags=00000000)
        32 (0017):     TAG_OBJECT_PATH (path=C:/Test/myFolder)
        73 (0110):     TAG_OBJECT_BEGIN
                            Name              : myFile.txt
                            Flags             : Cache Self, Cache Base
                            Attributes        : 00000020
                            Size              : 3
                            Generation        : 1.0
                            This              : SN0000000000/3
                            Link              : SN0000000000/3
                            Base              : SN0000000000/3
       207 (0039):         TAG_OBJECT_GENERIC
                            Name              : myFile.txt
                            IsDirectory       : 0
                            Size              : 3
                            Access Time       : 1523486615
                            Modified Time     : 1523486624
       273 (0028):         TAG_OBJECT_WINDOWS
                            Created Time      : 131679602159737201
                            Access Time       : 131679602159737201
                            Modified Time     : 131679602240253115
                            Attributes        : 00000020
       329 (0020):         TAG_ALT_STREAM (size=20, offset=0)
       373 (0172):         TAG_ALT_STREAM (size=172, offset=0)
       521 (0020):         TAG_ALT_STREAM (size=20, offset=0)
       565 (0003):         TAG_DATA_STREAM (size=3, offset=0)
       592 (0020):         TAG_ALT_STREAM (size=20, offset=0)
```

To get a better understanding of the dump file, let's examine the TAG_OBJECT_BEGIN in more detail.

```
 73 (0110):     TAG_OBJECT_BEGIN
                        Name                : myFile.txt
                        Flags               : Cache Self, Cache Base
                        Attributes          : 00000020
                        Size                : 3
                        Generation          : 1.0
                        This                : SN0000000000/3
                        Link                : SN0000000000/3
                        Base                : SN0000000000/3
```

Generation = 1.0 means that this is the first time this file was stored. This is called "the base". On large files, when the user changes just a small part of the file, Aparavi will store just the changes (also known as "the delta"). On that delta snapshot, the generation would be 1.1. Much like a "minor version" of a release.

Link = SN0000000000/3 means that it's the 3rd component of snapshot 0. Snapshot 0 contains the data, so later snapshots can just copy that from snapshot 0.

If a large file is stored and more than 50% of its data was changed, Aparavi will perform a new full copy of the file. Examining its generation would see 2.0 where the major number would increase, and the delta is set back to zero.

Example Snapshot History

The following example looks at snapshots 1 to 8 as file "HelloWorld" is created and changed over time.

```
SN000001: First full copy of file "HelloWorld"
type       ET_THIS        object data is contained within this set
generation 1.0            set to 1.0 as the base since there is no previous version
this       1/SN000001     TAG_OBJECT_BEGIN contained in current set
link       1/SN000001     no link. Data contained in current set
base       1/SN000001     set to current snapshot as this is full copy

SN000002: No changes were made to file "HelloWorld"
type       ET_LINK        object data is contained in link to previous set
generation 1.0            inherited from previous snapshot
this       0/(empty)
link       1/SN000001     inherited from previous set
base       1/SN000001     inherited from previous set

SN000003: Small changes were made to file "HelloWorld"
type       ET_THIS        object data once again is contained within this set
generation 1.1            minor version incremented by 1
this       3/SN000003     TAG_OBJECT_BEGIN contained in current set
link       3/SN000003     no link. Data contained in current set
base       1/SN000001     inherited from previous set
```

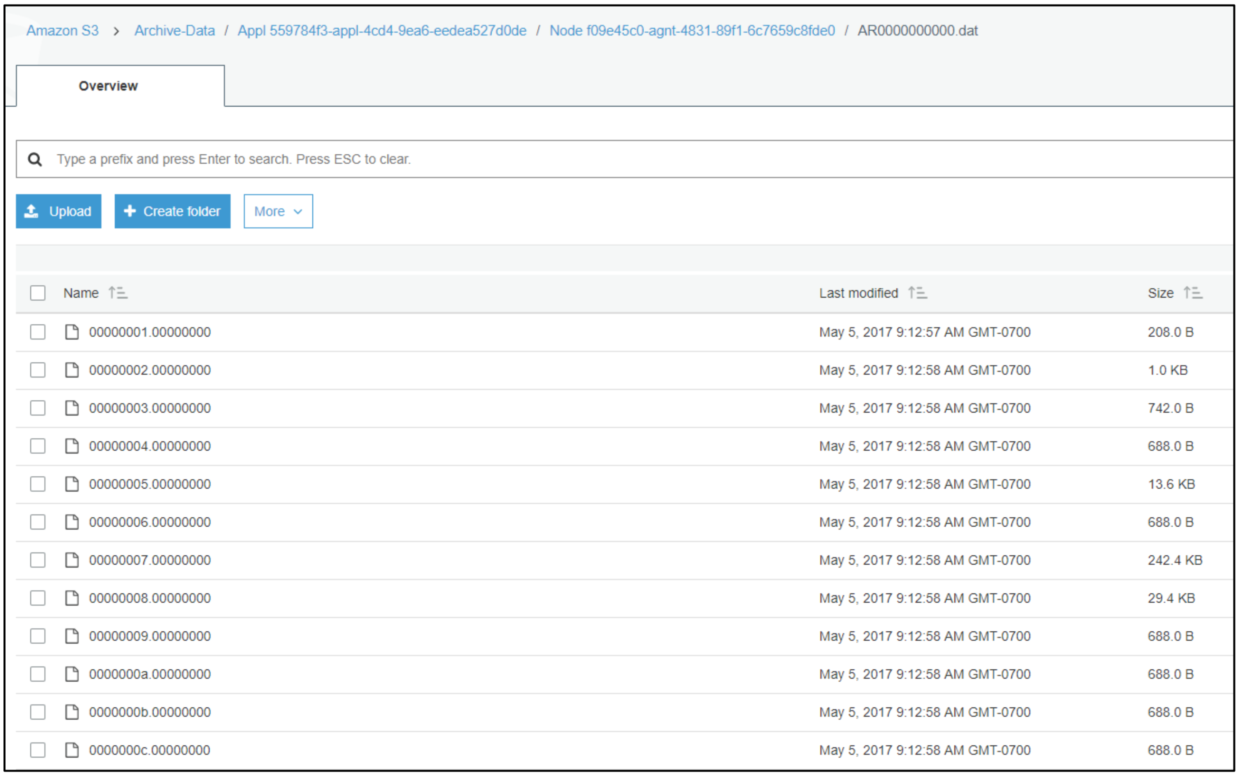
```
SN000004: No changes were made to file "HelloWorld"
type        ET_LINK        object data is contained in link to previous set
generation  1.1            inherited from previous set
this        0/(empty)
link        3/SN000003     inherited from previous set
base        1/SN000001     inherited from previous set

SN000005: No changes were made to file "HelloWorld"
type        ET_LINK        object data is contained in link to previous set
generation  1.1            inherited from previous set
this        0/(empty)
link        3/SN000003     inherited from previous set
base        1/SN000001     inherited from previous set

SN000006: Small changes were made to file "HelloWorld"
type        ET_THIS        object data once again is contained within this set
generation  1.2            minor version incremented by 1
this        6/SN000006     TAG_OBJECT_BEGIN contained in current set
link        6/SN000006     no link. Data contained in current set
base        1/SN000001     inherited from previous set

SN000007: No changes were made to file "HelloWorld"
type        ET_LINK        object data is contained in link to previous set
generation  1.2            inherited from previous set
this        0/(empty)
link        6/SN000006     inherited from previous set
base        1/SN000001     inherited from previous set

SN000008: Over 50% of file "HelloWorld" was changed
type        ET_THIS        object data once again is contained within this set
generation  2.0            increment major version by one to indicate a full copy
this        8/SN000008     TAG_OBJECT_BEGIN in this set
link        8/SN000008     no link. Data contained in current set
base        8/SN000008     set to current snapshot as this is full copy
```

Storage Destinations and File Structures

Snapshots are stored as .dat files on the appliance within the appliance's configured data directory. Checkpoints are also stored as .dat files, but are stored on the agent within the agent's configured data directory.

Archives are stored in a folder structure typically within a cloud storage provider. The folder structure is organized by appliance, then agent, then archive operation. The appliance and agent folder names use their "node ID". The node ID's are globally unique identifiers generated by Aparavi.

Each time an archive operation (i.e., a new archive) is performed, a new archive folder is created as an ARnnnn.dat folder within the appliance->agent folder structure as seen in the example below.



The data tags are stored within sequential files in the ARnnnn.dat folders. Each sequential file contains up to 5MB of data until the source file is completely written. The files are organized with a hex number suffix.

Agent and Appliance Identification

To identify the agent and appliance ID's and names, perform the following steps:

1. Determine where your data is stored for long-term data retention by examining the providers section of the policy
2. Access the cloud storage provider and navigate to the bucket defined by the policy
3. Inside that bucket you will see a folder called "Appl <id>" for the Appliance Node ID
4. Inside that folder you will see a set of folders named "Node <id>". Each of these Node ID folders represents a unique agent assigned to the current appliance
5. Within the agent folder you will see multiple ARnnnn.dat folders. Each of these folders are the archive operations
6. Within each ARnnnn.dat folder are the files that contain the tags representing your archive
7. Access the agent and appliance machines and navigate to the installation folder. Open the node_modules\agent (or appliance) folder and view the config.json file. Here you will see the nodeID values. If you are looking at the agent's file, the "parentObjectId" will contain the appliance's ID.

Appendix

Tag Types and Structures

```
typedef struct _tagTAG_COMPONENT_BEGIN_INFO {
    #define   COMPFLAG_NO_COPY   BIT(0)
    Dword     componentId;       // offset within stream where component begins
    Dword     componentFlags;    // flags for this component

} TAG_COMPONENT_BEGIN_INFO;

typedef struct _tagTAG_COMPONENT_END_INFO {
    Qword     phyComponentBegin;    // offset within stream where component begins
} TAG_COMPONENT_END_INFO;

typedef struct _tagTAG_DATA_COMPRESSED_INFO {
    Dword prevTag;               // previous tag
    Dword uncompressedSize;      // orginal uncompressed size
    Byte data[1];                // lz4 compressed data
} TAG_DATA_COMPRESSED_INFO;

typedef struct _tagTAG_DATA_ENCRYPTED_INFO {
    Dword prevTag;               // previous tag
    Byte signature[4];           // CRC-32 signature to ensure correct key
    Byte data[1];                // keyName + encrypted data
} TAG_DATA_ENCRYPTED_INFO;

typedef struct _tagTAG_OBJECT_GENERIC_INFO {
    Qword     fileSize;          // object size
    Qword     accessTime;        // last file access time
    Qword     modifiedTime;      // time of last modification
    Bool      isDirectory;       // is object a directory?
    Utf8      name[1];           // name of file / directory
} TAG_OBJECT_GENERIC_INFO;

typedef struct _tagTAG_OBJECT_LINUX_INFO {
     Qword    changeTime;        // time of last status change
    Dword     attributes;        // file attributes for fchmod
    Dword     ownerId;           // owner id of a file
    Dword     groupId;           // group id of a file
} TAG_OBJECT_LINUX_INFO;

typedef struct _tagTAG_OBJECT_WINDOWS_INFO {
    Qword     createdTime;  // time of file creation (FILETIME converted to Qword)
    Qword     accessTime;   // last file access time (FILETIME converted to Qword)
    Qword     modifiedTime; // time of last modification (FILETIME converted to Qword)
    Dword     attributes;   // file attributes
} TAG_OBJECT_WINDOWS_INFO;

typedef struct _tagTAG_LINK_INFO {
    BOOL      symbLink;
     Utf8     linkTo[1];         // Name of the file to link to
```

```
} TAG_LINK_INFO;


typedef struct _tagTAG_DELTA_VERSION_INFO {
    Dword version;              // version of delta class
    Dword fragSize;             // size of fragment
    Dword generationId;         // generation id
    Dword deltaId;              // delta id
    Dword baseComponent;        // component in base snapshot
    Utf8  baseName[MAX_SET_NAME_SIZE + 1];     // name of base snapshot
} TAG_DELTA_VERSION_INFO;

typedef struct _tagTAG_DELTA_REFER_INFO {
    Qword offsetBase;           // base input offset
    Dword count;                // number of referals
    Dword reserved;             // reserved area
    REFERAL referal[1];         // the referal
} TAG_DELTA_REFER_INFO;

typedef struct _tagTAG_DELTA_SIG_INFO {
    Dword count;                // number of signatues in sigs
    SIGNATURE sigs[1];          // an array of signatures
} TAG_DELTA_SIG_INFO;

typedef struct _tagTAG_DELTA_DEFINE_INFO {
    Qword blockPosition;   // position of the block relative to TAG_OBJECT_BEGIN
    Dword prevTag;         // previous tag (we changed it to TAG_DELTA_DEFINE)
    Dword blockId;         // not used
    Byte data[8];          // the data itself (more than 8, 8 is used for alignment)
} TAG_DELTA_DEFINE_INFO;
```