FileStore Migration

Introduction

The FileStore that is included with OA4MP is intended for smallish installations and does not really scale. A scenario is that one installs it for what is intended to be a low-volume service, but which instead becomes very large. For instance, if auto-approve is on, some clients register themselves essentially for every transaction (*i.e.* single use), resulting in hundreds of thousands of "clients" = one for every time a user logs in In that case, the system is overwhelmed and performance slows dramatically.

The Right Way to deal with this is to use a Derby store (which requires no database administration and just works), but how do you get all of your stores migrated? There is a tool for this that makes it quite easy and has been tested with about a million entries. The entire process takes a couple of hours.

How File Stores work

The basic functioning is that the unique identifier is hashed and that is used as the name of a file whose contents is the XML serialization of the object. Index entries are made (if needed) by hashing the key (*e.g.* the access token) and putting the hash of the id in it. For most cases of actual access, this works pretty darned well. The problem then gets to be if the store gets a lot of cruft in it. Since you do not know what the ids are (hashes are not easily reversible) you are left with navigating the entire store if you are searching. In some cases where auto-approval is on, stores can become enormous, since the client requesting access might literally create a new client for each flow, never to be used again. Cleaning up such a system is slow going, so the best way to scale up is to migrate the entire store to a database where it can be properly monitored. That is what this tool is for.

The FSMigration tool

This is a separate jar that is downloadable from GitHub. It runs at the command line and takes various arguments and does various things. In a nutshell the sequence is

1. Take the original source FileStore
2. Create a database (using Derby) that ingests every entry in the FileStore. You do not need to install anything for Derby! This is done automatically.
3. ***Ingest*** entries into a helper database located in the original filestore. This is relatively fast and just reads directly from disk as a first pass.
4. ***Migrate*** the in batches to the new SQL store (which can be any supported SQL store).
5. Allow for reports on what was imported. Since it is assumed you do not want to install Derby itself, this takes the place of doing direct queries.

6. Allow for deleting the migration database.

The reasons for the two stage migration are integrity and performance. Integrity ensures that valid store entries are faithfully transferred. This will also transfer pending transactions so the net effect should be that the system goes down for maintenance during the migration and comes up without any losses. The performance issue means not using the internal FileStore APIs but manually accessing the store contents and optimizing the operations. This means the operations run at easily over 100 times the speed that would happen otherwise in a large FileStore (which is the chief limitation of one).

Initial Prerequisites

1. The file store exists. There is not a directory in it named **`ingest`**. This will be created and managed.
2. The target store may or may not exist. If it does not exist, then it will be created.
3. Space available on the server in question. The entire file store will be processed at some point, though storing an object in a database takes up much less space than a file. If you have as much space available as the size of the file store, you should be fine.

Running the tool

The tool is named **`fs-migrate.jar`** and is an executable jar. Starting it with no parameters shows the help:

```
java -jar fs-migrate.jar
(help is printed)
```

Any databases will be created if needed. You may also create them directly by running the tool with the **`-setup`** flag:

```
java -jar fs–migrate.jar -src … -setup
```

Make sure the source and target are set.

The Pacer

There is a command line status bar called *Pacer* that will run. It outputs percentage done, speed and such only to the command line (not to the log file). This should stay parked on one line and the aim is for a compact, easily readable way to monitor what might be a long job. A typical snapshot is

```
.................../..\..................  114000  files ingested @ 111.1110 KHz
```

The slashes seem to walk back and forth, hence pace hence the name Pacer. In this case, it tells the number of files and the speed. Depending upon the context, more information may be there.

That said, Java runs in many places and Pacer does not always quite work right (Java's unofficial motto is "write once, debug everywhere!") thanks to differences in IO, so you can turn off pacing with a boolean argument, **`-noPace`**. If you don't see a tidy little pair of workers pacing, try turning it off.

Ingestion

In order to process huge numbers of files, the entire file store's files is read (but not opened) from the file system and a specific database is created to track the progress of each file. This is the *ingestion database*. This first pass does not actually access the contents of the files, just records everything about the files. Later in the *migration phase* the actual files are processed. In this way the system is not overloaded and processing may be continued after interruptions. Ingestion is assumed to be atomic, in the sense that it is either done and completed or not done at all. If there is a problem, you should either remove the (corrupted) ingestion database manually or just run the tool with the **`-cleanup`** flag. This will remove the database *in toto*, not just remove the entries.

Typical output from ingestion is (line breaks added for readability)

```
java -jar fs-migrate.jar \
  -src $SRC_CFG\
  -srcName $SRC_NAME\
  -targetName $TARGET_NAME\
 -v \
 -showConnect
No explicit target configuration, using the source configuration for both.
FSMigrationTool database file=/home/ncsa/temp/oa4mp2/fileStore/ingest
   creating migration database
   done!
connect
'jdbc:derby:/home/ncsa/temp/oa4mp2/fileStore/ingest;dataEncryption=true;user=oa4mp;
bootPassword=Qwertyuiop321;password=Asdfghjkl123';
  :starting to ingest
  :processing 52639 files from
/home/ncsa/temp/oa4mp2/fileStore/adminClients/dataPath.
.........../..............\..........  52000  files read @ 76.9230 KHz
```

Things to note

- **`-showConnect`** means the connection string to the ingestion database is shown. If you have Derby installed and want to use their command line **ij** tool to snoop, you can
- There is a status bar that allows you to watch the ingestion. This can be turned off.
- Each store is ingested and stats for that are displayed.
- If you have upkeep logic in the target store, that will be applied in the next phase. Ingestion is strictly to grab every file name in the store and set up a system to track migrating it.
- The ingestion database is made. This is atomic in the sense that if it exists it is assumed to be intact. If for some reason the process fails, you will have to remove it and its directory, here **`/home/ncsa/temp/oa4mp2/fileStore/ingest`** and is part of the connection string.

Once the files have been read, they will be ingested, i.*e.*, the database will be updated. This is typically much slower than reading in the files for processing. Typical output is

```
  :ingesting...
  :       store : adminClients
  :       total : 52639
  :    rejected : 0
  :update speed : 522 Hz
  :       bytes : 122.9746 MB
  :  total time : 1.6910 min.
```

The system may hang for a while while ingesting, since the migration tool hands over processing to the database and cannot do more than wait. After a bit, the stats will print. The update speed is how fast the database was processing entries, here, 522 rows per second.

A practical issue with migrating a simply huge file store is tracking progress. Since the files are saved as serialized XML and the name of the file is a hash of the ID – not generally reversible – the only way to really check if a file has been migrated would be to read every file and parse it. The ingestion database pulls in the list of files and as they are migrated will track progress. This allows for restarting the process.

Migration

This is when the entries in the ingestion database are processed. If there are upkeep rules on the target, yo may have them enforced at this time. While not recommended, you can restart the process if it should stop. The ingestion database tries to ensure atomicity, but it is always best to not tempt Fate. A typical migration report looks like

```
  :migrating 15708 items for permissions
........\....................../........  10  files migrated in permissions,
rejected=15 @ 757 Hz upkept=15683
  :       total : 15708
  :      upkept : 15683
  :   bad files : 15
  :   net files : 10
  :        time : 4.8918 min.
  :   av. speed : 53 Hz
  :    no main entry:15683, file did not exist:15
```

In this case, the upkeep rules are being applied and out of 15,708 files. Pacer would have been updating with counts the entire time, so this is a snapshot of the end. This means

- 15 were bad (in this case, they had been deleted somehow by the time migration took place),
- 15,683 are excluded (no main entry means that for permissions, there was no corresponding client or admin client, hence the permission itself is unusable).
- 10 entries passed muster.
- The average processing speed was 53 entries per second and the entire process took 4.8918 minutes.
- There are now 10 fully functional permissions in the target store.

Speed

Updating a database involves writing to multiple files and updating indices. It is therefore slow. Using the upkeep facility to filter entries so they are not written is the single best way to speed up a very large migration. Usually in a massive store, the issue is that there are a lot of dead/unused/unusable entries which are dragging down performance.