HSM zu S3

Migration von Daten auf Band in einen Object Store

Carsten Grohmann

1. Juni 2022

Agenda

Agenda

Migration

Ziele

Ziele

1. alle Dateien nachweislich korrekt übertragen
2. die Migration soll robust und fehlertolerant sein
3. der Datendurchsatz ist den ersten beiden Zielen untergeordnet

Verwendete Hardware

Verwendete Hardware

HSM

- Plattencache im Frontend und Bändern als Backend
- Netzwerk: 10GBit/s

Transferserver

- RAM: ausreichend $>$ 128GB
- CPU: ausreichend $>$ 12 Cores
- Netzwerk: 10GBit/s - nie gesättigt

S3-Appliance

- vor Ort
- Netzwerk: 10GBit/s

Verwendete Software

Verwendete Software

- Python[1]
  - peewee[2]
  - Requests: HTTP for Humans[3]
  - urllib3[4] (wird von Requests benötigt)
- rsync[5]
- s3fs[6]
- Bash[7]

---

[1] https://www.python.org
[2] http://docs.peewee-orm.com/
[3] https://docs.python-requests.org/en/latest/
[4] https://urllib3.readthedocs.io/
[5] https://rsync.samba.org/
[6] https://github.com/s3fs-fuse/s3fs-fuse
[7] https://www.gnu.org/software/bash/

Ablauf

Ablauf

1. Daten in der Quelle gegen Änderungen sperren
2. Batchlauf ggf. mehrere Runden
   - 2.1 Dateiliste bilden
   - 2.2 Prüfsummen auf der Quelle bilden
   - 2.3 Dateien kopieren
   - 2.4 Prüfsummen auf dem Ziel bilden
   - 2.5 Status aktualisieren
3. Berichtswesen

Performance

Durchsatz über alle Schritte

Anzahl Dateien	Volumen	Durchsatz [MB/s]	Durchsatz [TB/d]
14k	1,4 TB	38 MB/s	3,00 TB/d
100k	850 GB	9 MB/s	0,70 TB/d
100k	49 GB	440 KB/s	0,04 TB/d
100k	1,2 GB	13 KB/s	0,00 TB/d

S3-Objekte kopieren - Bucket zu Bucket

Anwendung	Durchsatz [MB/s]	Durchsatz [TB/d]
S3CopyBucket[8] (10 Threads)	500 MB/s	45 TB/d
rclone (10 Threads)	365 MB/s	30 TB/d

[8] https://carstengrohmann.de/s3copybucket.html

Probleme unterwegs

multiprocessing.Queue blockiert beim Beenden

Ausgangssituation

- Dead Lock beim Beenden mit Ctrl-C

Lösung

- Ursache nicht wirklich gesucht, `(gdb) py-bt` war noch unbekannt
- verzeichnisbasierte Queue ohne globale Sperre
- funktioniert stabil

HSM REST-API unzuverlässig

Ausgangssituation

- REST-Service instabil
- automatischer Neustart, aber
  - dauert lange
  - Timeouts während des Neustarts
  - ungültige / unerwartete Antworten während dessen

Lösung

- `urllib3.util.Retry()` mit großen Backoff-Faktor
- Backoff-Faktor
  - Verzögerung nach dem 2. Versuch, denn die meisten Fehler werden durch einen sofortigen erneuten Versuch behoben
  - Faktor 1.0: `[0s, 2s, 4s, ...]`
  - Faktor 50.0: `[0s, 50s, 100s, ...]`
  - Berechnung: `{backoff factor} * (2 ** ({number of total retries} - 1))`

```
FACTOR = 50.0
NR_RETRY = 4

retry_strategy = urllib3.util.Retry(
    total=NR_RETRY, connect=NR_RETRY, read=NR_RETRY, status=NR_RETRY, backoff_factor=FACTOR,
)
retry_strategy.BACKOFF_MAX = 600
http = PoolManager(retries=retry_strategy)
response = http.request('GET', 'http://example.com/')
```

SQLite langsam

Ausgangssituation

Datenbankabfragen und -aktualisierungen dauern mehrere Stunden für 100k Dateien

Datenbankmodellierung mit Peewee

```
from peewee import *
db = SqliteDatabase('db.sqlite')

class File(Model):
    class Meta:
        database = db
        indexes = ((('path', 'share'), True),)
    NOT_STARTED = 0
    STARTED = 101
    FAILED = 102
    SUCCESS = 103

    path = CharField(index=True)
    share = CharField(index=True)
    size = IntegerField(index=True)
    state = IntegerField(default=NOT_STARTED, null=False)
```

Schema von Peewee erzeugt

```
CREATE TABLE IF NOT EXISTS "file" ("id" INTEGER NOT NULL PRIMARY KEY,
                                   "path" VARCHAR(255) NOT NULL, "share" TEXT NOT NULL,
                                   "size" INTEGER NOT NULL, "state" INTEGER NOT NULL);
CREATE INDEX "file_path" ON "file" ("path");
CREATE INDEX "file_share" ON "file" ("share");
CREATE INDEX "file_size" ON "file" ("size");
CREATE UNIQUE INDEX "file_path_share" ON "file" ("path", "share");
```

Beispiel: Datenbank-Update mit peewee

```
File.update(state=File.STARTED).where(File.share == 'MyShare', File.path.in_(['A', 'B', 'C']))
```

Beispiel: Test Datenbank-Update

Bei allen Tests wurden immer die gleichen 800 Dateien abgefragt/aktualisiert.

```
filenames = [
    '/file1',
    '/file2',
    '/file3',
    '...',
    '/file799',
    '/file800',
]

def run(chunk_size, new_state):
    with db.atomic():
        for chunk in chunked(filenames, chunk_size):
            query = File.update(state=new_state).where(File.share == 'MyShare',
                                                       File.path.in_(chunk))
            query.execute()
```

Verwendete Versionen

- RHEL 7.8
- Python 2.7
- SQLite 3.7

Testergebnisse

Kommentare zu den Ergebnissen

- Spalte “DpS”: Dateien pro SQL-Statement
- Spalte “Dauer”: Gesamtzeit, um den Status von 800 Dateien in der Datenbank zu ändern / abzufragen

Ausführungszeiten mit verschiedenen Transaktionstypen

Transaktionstyp	DpS	Dauer	DpS	Dauer
Eine Transaktion pro Query [9]	800	96 s	10	0,12 s
Alle Queries eine Transaktion [10]	800	75 s	10	0,12 s
Manuelles Commit am Ende [11]	800	77 s	10	0,18 s

[9] `Database.atomic()`
[10] `Database.transaction()`
[11] `Database.manual_commit()`

Ausführungszeiten verschiedener SQL-Funktionen

Test	DpS	Dauer	DpS	Dauer	DpS	Dauer
SUM	800	78 s	10	4900 s	5	0,23 s
COUNT	800	78 s	10	4971 s	5	0,28 s

```
>>> SUM = File.select(fn.SUM(File.size))
>>> SUM.sql()
('SELECT SUM("t1"."size") FROM "file" AS "t1"', [])

>>> COUNT = File.select(fn.COUNT(File))
>>> COUNT.sql()
('SELECT COUNT("t1") FROM "file" AS "t1"', [])
```

Erkenntnis und Lösung

1. (bei Bedarf) Performancetests durchführen
2. SQL-Statements anpassen

```
SQLITE_COUNT_UPDATE = 10
SQLITE_COUNT_FUNC = 5
with db.atomic():
   for chunk in chunked(filenames, SQLITE_COUNT_UPDATE):
      query = File.update(state=new_state).where(File.share == 'MyShare', File.path.in_(chunk))
      query.execute()
```

Externe Prozesse kontrollieren

Ausgangssituation

- nur `rsync` als externer Prozeß
- durch IO
  - lange Laufzeiten
  - teilweise schlecht / gar nicht zu beenden

Lösung

Starten

- externe Prozesse sauber vom startenden Prozess abtrennen
  - `setsid(2)`:
    - eigene Session - no controlling terminal
    - eigene Process Group

```
def start():
    child = subprocess.Popen(cmd, stdin=open(os.devnull, 'rwb'), stdout=outfile,
                             stderr=subprocess.STDOUT, close_fds=True, preexec_fn=os.setsid)
```

Stoppen

- Signale an alle Mitglieder der Process Group schicken
- wenn der Prozess auf IO wartet, wird er vom System terminiert, sobald er wieder "runable" ist

```
def stop():
   for sig in [signal.SIGTERM, signal.SIGKILL]:
       os.killpg(os.getpgid(child.pid), sig)  # An alle Mitglieder der Process Group
       for unused in range(300):
           if child.poll() is not None:
               break
           time.sleep(1)

       if child.poll() is not None:
           child = None
           return
```

Mögliche Verbesserungen

Mögliche Verbesserungen

- keine Prüfsumme auf der Quelle bilden, spart etwas Zeit, aber nicht viel
- Daten und Metadaten direkt (ohne s3fs) ins S3 schreiben
- mehr Parallelisieren, um verschiedene IOs auf Quelle und Ziel gleichzeitig auszuführen

Danke

Danke

- Fragen, Anregungen, Meinungen
- Vielen Dank für die Aufmerksamkeit!

Lizenz

Lizenz

[12] https://creativecommons.org/licenses/by-nc-sa/4.0/deed.de