Real-Time Data Loading into Teradata using Kafka and Teradat-Python

Teradata への real time に近い形でデータ挿入実行するための Prototype を Kafka と python を使って作ってみました。

This page introduce a sample program for real-time data loading into Teradata using Kafka and Teradata-Python.

毎分数万件のデータを Teradata にほぼリアル・タイムでデータを挿入したいという案件があったため、どのようにすると実現できるかを考えるため、作成方法と性能を検証してみました。
Kafka を使うのは、Teradata Listener demo でも使われており興味があったのと、このような案件に組み込むことで容易に連続的なデータ挿入が実現できそうなので、勉強もかねて使ってみました。

今回の Prototype では以下のSWを使用します。
The following components are used.

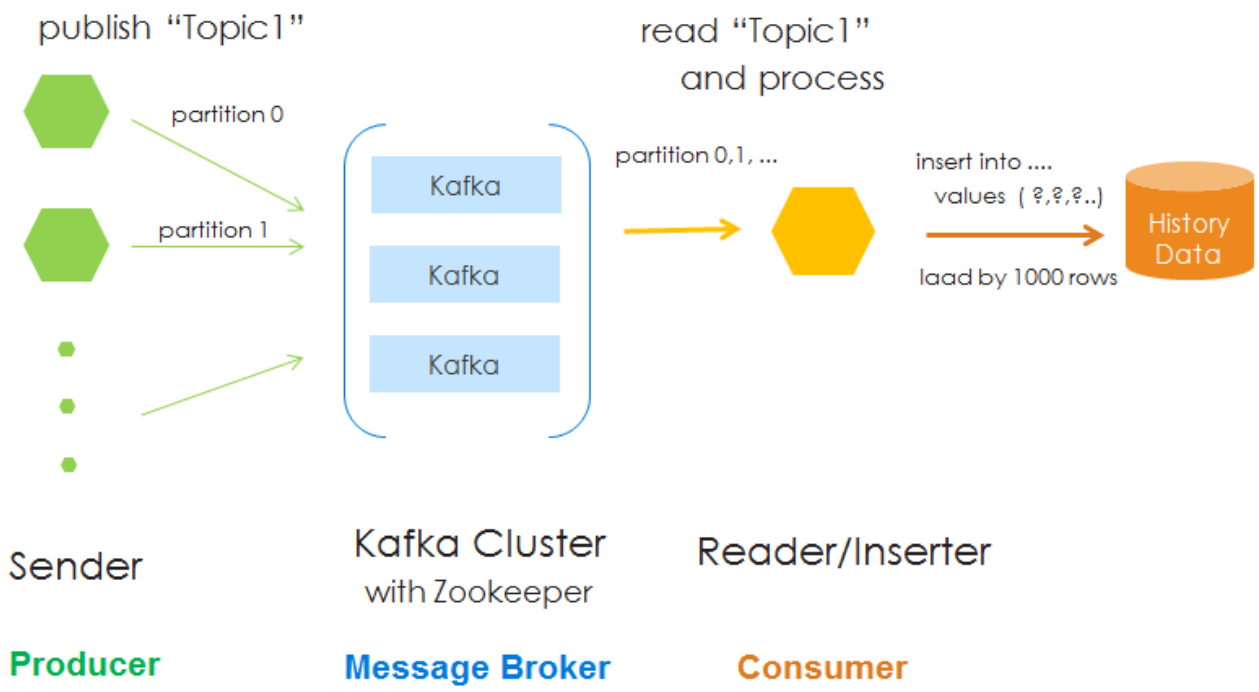
● Kafka-Python module： GitHub - dpkp/kafka-python: Python client for Apache Kafka
Python による Kafka のクライアントです。 Python から official な Java クライアントと同様のAPIを利用できます。

● Teradata Python Module： Teradata Python Module | Teradata Developer Exchange
Python を使って Teradata へアクセスするためのAPIを提供しています。ODBC、或いは Teradata REST Service を使って Teradata へアクセスします。

● Apache Kafka： Apache Kafka
Apache Kafka については、Apache Kafka のページに分かりやすい解説があります。Kafkaに初めて触れるときの手順についても詳しい記述があるので、Kafka についてより深く知りたい時は本家のサイトを参考にすることを推奨します。

システム概要 - System Overview

今回作成するシステムの概要は、以下の図のようなものです。
真ん中にメッセージ・ブローカーして Kafka Cluster があり、Producer（Sender）がデータをメッセージとしてKafkaに書き込んでいきます。
Consumer 側ではKafka から常にメッセージを読み込んでおり、まとまった件数（例えば500件 or 1000件）がたまった時、あるいはKafka からメッセジーの到着が一時知的に途切れた時に、まとまった行を一括して Teradata にインサートします。



Producer はTopicを指定して message を publish します。Consumer が message を読み込むときも Topic を指定して読み込みます。Kafka は１つのクラスターで複数の Topic をサポートできるので、複数の Source-to-Target の stream を共存させることができます。また、1つの Topic を複数の partition に分けて伝送することができ、これにより message 転送の throughput を向上させることができます。例えば複数の Producer がそれぞれ異なる partition のメッセージを publish するような構成にすることでそれを実現できます。また、Consumer の方も複数起動して、それぞれが特定の partition を読むようにすることもできます。なお、各 partition 内では message の順番は変更されずにConsumer に届けられることは保障されています。

今回の Prototype では、3つの Virtual Machine を使います。Teradata は Teradata Express 15.10。Kafka には１つの VM を割り当てます。もう１つのVMから Producer と Consumer のプログラムを実行します。Kafka 及び Producer/Consumer のシステムには vCore を１つ、メモリを 1GB 割り当てています。（小さい！）
なお、図の中では Kafka は複数のインスタンスによるクラスター構成になっていますが、今回は１インスタンスのみで構成しました。

Kafka の実行 - Starting Kafka programs

Kafka のアーカイブを Apache Kafka よりダウンロードし展開します。
Download Kafka a package from Apache Kafka and extract files from it.

```
[root@myLinux ~]# tar -zxvf kafka_2.11-0.9.0.1.tgz
kafka_2.11-0.9.0.1/
kafka_2.11-0.9.0.1/LICENSE
kafka_2.11-0.9.0.1/NOTICE
kafka_2.11-0.9.0.1/bin/
kafka_2.11-0.9.0.1/bin/connect-distributed.sh
kafka_2.11-0.9.0.1/bin/connect-standalone.sh
kafka_2.11-0.9.0.1/bin/kafka-acls.sh
kafka_2.11-0.9.0.1/bin/kafka-configs.sh
```

kafka_<version-number> という形式のディレクトリができるので、そこへ移動します。

```
[root@myLinux ~]# ls
kafka_2.11-0.9.0.1  kafka_2.11-0.9.0.1.tgz
[root@myLinux ~]# cd kafka_2.11-0.9.0.1
[root@myLinux kafka_2.11-0.9.0.1]# ls
bin  config  libs  LICENSE  NOTICE  site-docs
```

bin、config、libs などのディレクトリがあります。libs にはkafka の本体にあたる jar ファイルが格納されています。
bin ディレクトリには、そのまま実行可能な shell スクリプトがあり、簡単に kafka を試すことができます。config ディレクトリはそれらの shell script をじっこうするときに必要なパラメータを定義した proiperties ファイルが格納されています。

```
[root@myLinux kafka_2.11-0.9.0.1]# ls bin config
bin:
connect-distributed.sh            kafka-mirror-maker.sh               kafka-simple-
consumer-shell.sh
connect-standalone.sh             kafka-preferred-replica-election.sh  kafka-topics.sh
kafka-acls.sh                     kafka-producer-perf-test.sh          kafka-verifiable-
consumer.sh
kafka-configs.sh                  kafka-reassign-partitions.sh         kafka-verifiable-
producer.sh
kafka-console-consumer.sh         kafka-replay-log-producer.sh         windows
```

```
kafka-console-producer.sh         kafka-replica-verification.sh        zookeeper-security-
migration.sh
kafka-consumer-groups.sh          kafka-run-class.sh                   zookeeper-server-
start.sh
kafka-consumer-offset-checker.sh  kafka-server-start.sh                zookeeper-server-
stop.sh
kafka-consumer-perf-test.sh       kafka-server-stop.sh                 zookeeper-shell.sh

config:
connect-console-sink.properties    connect-file-source.properties  log4j.properties
tools-log4j.properties
connect-console-source.properties  connect-log4j.properties        producer.properties
zookeeper.properties
connect-distributed.properties     connect-standalone.properties   server.properties
connect-file-sink.properties       consumer.properties             test-log4j.properties
```

最初に zookeeper を、続いて kafka server を立ち上げます。
Start zookeeper and kafka server using the following commands.

```
[root@myLinux kafka_2.11-0.9.0.1]# ./bin/zookeeper-server-start.sh  -daemon config/
zookeeper.properties
[root@myLinux kafka_2.11-0.9.0.1]#
[root@myLinux kafka_2.11-0.9.0.1]# ./bin/kafka-server-start.sh -daemon config/
server_HT.properties
[root@myLinux kafka_2.11-0.9.0.1]#
[root@myLinux kafka_2.11-0.9.0.1]# ps -ef | grep kafka
kafka      1608      1  0 12:41 ?        00:00:14 java -Xmx512M -Xms512M -
server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:+DisableExplicitGC -Djava.awt.headless=true -Xloggc:/home/kafka/
kafka/bin/../logs/zookeeper-gc.log -verbose:gc -XX:+PrintGCDetails -XX:
+PrintGCDateStamps -XX:+PrintGCTimeStamps -Dcom.sun.management.jmxremote -
Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
-Dkafka.logs.dir=/home/kafka/kafka/bin/../logs -Dlog4j.configuration=file:./
bin/../config/log4j.properties -cp :/home/kafka/kafka/bin/../libs/*
org.apache.zookeeper.server.quorum.QuorumPeerMain config/zookeeper.properties
kafka      1760      1  0 12:43 ?        00:01:07 java -Xmx1G -Xms1G -server -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+DisableExplicitGC
-Djava.awt.headless=true -Xloggc:/home/kafka/kafka/bin/../logs/kafkaServer-gc.log
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -
Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false -Dkafka.logs.dir=/home/kafka/kafka/bin/../logs -
Dlog4j.configuration=file:bin/../config/log4j.properties -cp :/home/kafka/kafka/bin/../
libs/* kafka.Kafka config/server_HT.properties
root       2570   2392  0 17:22 pts/0    00:00:00 grep --color=auto kafka
```

```
```

ここではほぼデフォルトの properties ファイルを使っていますが、kafka が稼働する VM のメモリが少ないため、cache が大きくなり過ぎないうちに flush するよう設定を変更しています。また、retention （データを保持する）期間も小さくしています。
Here, only small portion of the configuration file is modified.

```
[root@myLinux kafka_2.11-0.9.0.1]$ diff config/server.properties config/
server_HT.properties
83a84
> log.flush.interval.messages=10000
86a88
> log.flush.interval.ms=10000
96c98,99
< log.retention.hours=168
---
> #log.retention.hours=168
> log.retention.hours=24
119a123,127
>
```

Producer (sender.py)

Producer/Consumer is written using kafaka-python module.

Producer/Consumer は python を使って書いています。

kafka-python モジュールからいくつかの関数、Class などを import します。

```
#!/usr/bin/python

import sys
import re
import codecs
import time
import random
```

```
from kafka import KafkaProducer
from kafka.structs import TopicPartition
from kafka.structs import OffsetAndMetadata
import kafka.errors as Errors

hosts=['myLinux:9092']
```

KafkaProducer を読んで、Kafka サーバーと接続します。
Connect to the Kafka server by calling KafkaProducer().

```
if client_id:
        producer=KafkaProducer( bootstrap_servers=hosts,
                        acks=acks,
                        retries=retries,
                        batch_size=batch_size,
                        linger_ms=linger_ms,
                        buffer_memory=buffer_memory,
                        max_request_size=max_request_size,
                        request_timeout_ms=request_timeout_ms,
                        client_id=client_id)
else:
        producer=KafkaProducer( bootstrap_servers=hosts,
                        acks=acks,
                        retries=retries,
                        batch_size=batch_size,
                        linger_ms=linger_ms,
                        buffer_memory=buffer_memory,
                        max_request_size=max_request_size,
                        request_timeout_ms=request_timeout_ms)
```

while loop の "key_id = .,.... " から "message= ....." の行までは、random module、time module を使ってテスト用データを作成しているところです。
作成されたデータは producer.send() method を使ってその都度送っています。
また、rows 行のデータを送ったところで interval 秒だけ sleep するようになっています。
デフォルトでは1秒毎に100行程度を送るようになっています。

This is a part that creating sample test data which is created using random module.

```
try:
        while True:
                key_id= '%08d' % (random.randrange( key_min,key_max))
                tstmp=time.strftime('%Y-%m-%d %H:%M:%S',time.localtime())
                latitude=str(random.gauss(35,20))
                longitude=str(random.gauss(140,20))
                message= key_id + delim + tstmp + delim + latitude + delim + longitude

                try:
                        rMeta=producer.send( topic=topic , partition=partition,
value=message)
                        if debug > 1000000:
                                print(rMeta.get())
                except(Errors.KafkaTimeoutError) as e:
                        print('KafkaTimeoutError: ' + str(e))

                row += 1
                if row == rows:
                        row=0
                        cycle += 1
                        if debug == 1:
                                print(rMeta.get())
                        if interval:
                                time.sleep(interval)
                        if max_cycle and cycle == max_cycle:
                                break

except(KeyboardInterrupt):
                print('Receive user interrupt.')
                print('exitting.....')
                producer.close()
                sys.exit(0)
```

Topic "Test2" として message を投入します。 partition 0 及び 1 と 2 つの Producer sender.py を立ち上げます。各 sender は毎分約100 message、合計で毎分約 200 message を投入します。
また各 sender.py に投入する message の key range を指定しています。これは、partition 毎に異なる key range の message を投入させることを意図しています。

Two producers are started. Each producer create and submit 100 messages per minute.

```
[hisashi@mykdc kafkaTera]$ sender.py --topic Test2/0 --key_range 0,2500 &
```

```
[1] 1945
[hisashi@mykdc kafkaTera]$ sender.py --topic Test2/1 --key_range 2500,5000 &
[2] 1947
[hisashi@mykdc kafkaTera]$ jobs
[1]-  Running                 sender.py --topic Test2/0 &
[2]+  Running                 sender.py --topic Test2/1 &
[hisashi@mykdc kafkaTera]$
```

Consumer (inserter.py)

Consumer is written using teradata-python module other than kafka-python module.
Producer と同様に kafka-python の module を import しますが、それにくわえて teradata-python の module も import します。

```
#!/usr/bin/python

import sys
import re
import codecs
import time
import datetime
from kafka import KafkaConsumer
from kafka.structs import TopicPartition
from kafka.structs import OffsetAndMetadata
import kafka.errors as kafkaerror
import teradata

tdpid=""

rows=500
topic=''
hosts=['myLinux:9092']
consumer_timeout_ms=5000
```

Teradata に接続します。接続には odbc を使用しています。

Connect to Teradata using odbc.

```
uda=teradata.UdaExec()
if tdpid:
        session=uda.connect("${ODBC}",system=tdpid)
else:
        session=uda.connect("${ODBC}")
```

Kafka サーバーに接続します。
Connect to Kafka server by calling KafkaConsumer(),

```
if group:
       consumer=KafkaConsumer( group_id = group,
                 bootstrap_servers=hosts,consumer_timeout_ms=consumer_timeout_ms)
else:
       consumer=KafkaConsumer(
                  bootstrap_servers=hosts,consumer_timeout_ms=consumer_timeout_ms)
```

Consumer が読む Topic, Partition を assign/subscribe します。
Specify the topic, partitions which the consumer will subscribe.

```
topicPartition=[]
for p in partition:
        topicPartition.append(TopicPartition(topic,p))

if topicPartition:
        try:
                consumer.assign(topicPartition)
        except(kafkaerror.IllegalStateError) as e:
                print("IllegalStateError: %s" % (e))
        reading_partition=partition
else:
        reading_partition=consumer.partitions_for_topic(topic)
```

次のコードで幾つかのパラメータを指定します。Kafka に保存されているもっとも古いメッセージから、最新のメッセージから、あるいは offset を指定して特定のメッセージから読み込みを開始することができます。
Set the option to specify from which messages the consumer will read.

```
if seek_to_beginning:
        try:
                consumer.seek_to_beginning()
        except(AssertionError) as e:
                print("AssertionError: %s" % (e))
                sys.exit(1)
elif seek_to_end:
        try:
                consumer.seek_to_end()
        except(AssertionError) as e:
                print("AssertionError: %s" % (e))
                sys.exit(1)
if offset:
        i=0
        for tp in topicPartition:
                try:
                        consumer.seek(tp,offset[i])
                        if len(offset)>1:
                                i+=1
                except(AssertionError) as e:
                        print("AssertionError: %s" % (e))
                        sys.exit(1)
```

Inserter は "for message in consumer:" loop により、常に message を読み続けます。そして指定された数(rows)の message を受け取ると、insert_rows() を呼び出して、まとめて Teradata に insert します。
また、指定された時間 (consumer_timeout_ms: default 5000ms) message の受信が無かった場合も、それまでに受け取った message を insert します。（以下の 13行目の " if row: " の部分）

Consumer continues to read messages and after getting specified number of messages it load the received data into Teradata. It also loads data if it does not receive messages for specified timeout period (default timeout value is 5000 milliseconds).

```
row=0
mvalues=[]
try:
        while True:
                try:
                        for message in consumer:
                                row += 1
```

```
                                mvalues.append(message)
                                if row == rows:
                                        insert_rows( session, row, mvalues)
                                      row=0
                                        mvalues=mvalues[:0]
                        if row:
                                insert_rows( session,row, mvalues)
                                row=0
                                mvalues=mvalues[:0]

                except(StopIteration) as e:
                        print('StopIteration: %s' % (e))
                        continue

except(KeyboardInterrupt):
        insert_rows( session,row, mvalues)
        print('Receive user interrupt...')
        consumer.close()
        session.close()
        sys.exit(0)
```

以下は insert_rows() 関数の詳細です。SQL は executemany(SQL, parameters.. , option) で parameter query として BATCH mode で実行します。受け取った message ("values") を一旦 param_array にまとめた後、tuple params の形にしたうで executemany（）に渡しています。こうすることで、まとまった行のデータを高速に insert することが可能になります。
This is a main part of insert_row(). Messages are loaded using BATCH mode, after formatting arrayed parameters into tuple.

```
SQL="""insert into continuous_load_10m values (?,?,?,?) """

def insert_rows( session, nrow, values):
        param_array=[]
        offsets=''


        for m  in values:
                vs = m.value.split(delim)
                param=( vs[0], vs[1], vs[2], vs[3] )
                param_array.append(param)

        params=tuple(param_array)
        for  p in reading_partition:
                v=[ x for x in  values if x.partition==p ]
                if v:
                        offsets += '(%d,%d)' % (p,v[-1].offset)

        if params:
```

```
                st_time=time.strftime('%Y-%m-%d %H:%M:%S',time.localtime())
                start=datetime.datetime.now()
                try:
                        session.executemany(SQL, params,batch=True,logParamFrequency=rows)
                        end=datetime.datetime.now()
                        dur=end-start
                        print("%s: insert %d rows,  %02d.%06d sec, (parition,offset)=[%s]"
                                        % (st_time, nrow, dur.seconds,
dur.microseconds,offsets))
                except(teradata.api.DatabaseError) as e:
                        print(e)
```

Consumer inserter.py を起動します。
最初は、先に起動されていた sender.py が投入した message が kafka に溜まっているため、それを連続して処理していきます。処理が追いつくと、2秒、或いは3秒間隔で insert 処理を実行します。
各 insert 処理は 0.03秒から、遅くても1秒以内で完了しています。

Start Consumer program.

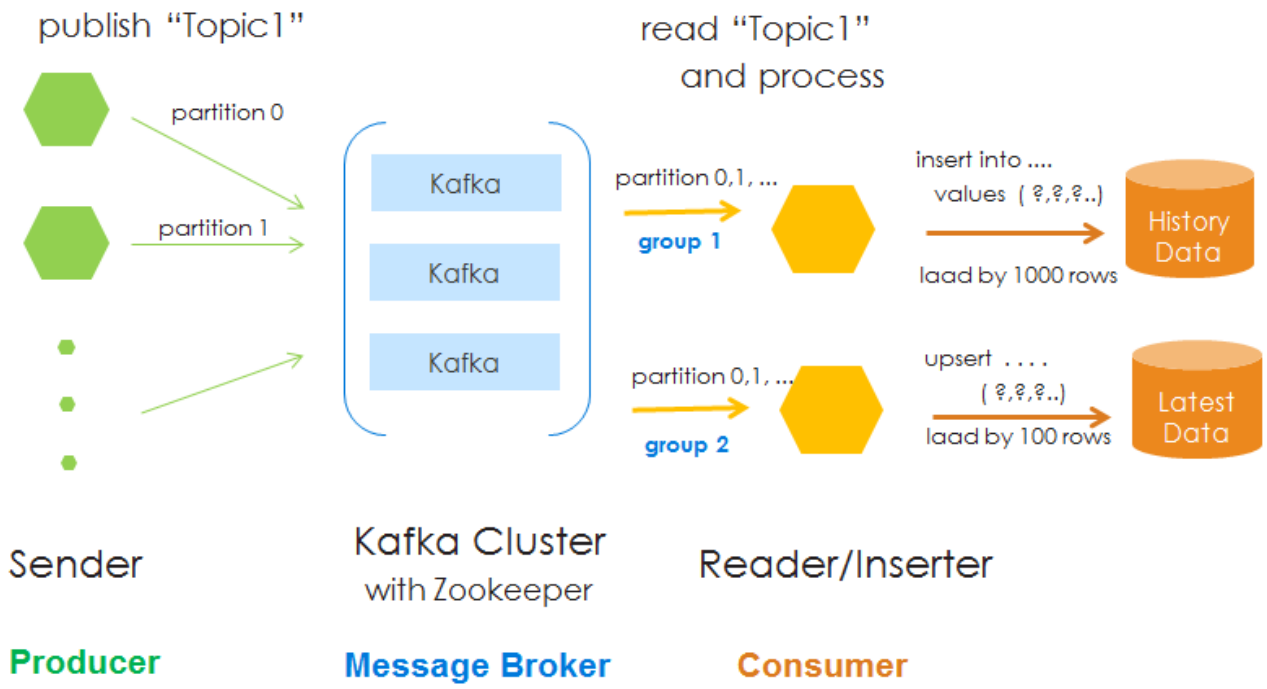
```
[hisashi@mykdc kafkaTera]$ inserter.py --topic Test2 &
[3] 1955
[hisashi@mykdc kafkaTera]$ Setting up Teradata Connection....
Setting up Kafka Connections....
Conntecting server, and subscribe Test2
set([0, 1, 2])
Consumer Timeout
2016-08-11 15:10:27: insert 500 rows,  00.171174 sec, (parition,offset)=[(0,5316600)]
2016-08-11 15:10:27: insert 500 rows,  00.032057 sec, (parition,offset)=[(0,5317100)]
2016-08-11 15:10:27: insert 500 rows,  00.031206 sec, (parition,offset)=[(0,5317600)]
2016-08-11 15:10:27: insert 500 rows,  00.029831 sec, (parition,offset)=[(0,5318100)]
2016-08-11 15:10:27: insert 500 rows,  00.091643 sec, (parition,offset)=[(0,5318600)]

2016-08-11 15:10:29: insert 500 rows,  00.042453 sec, (parition,offset)=[(0,5328100)]
2016-08-11 15:10:29: insert 500 rows,  00.040720 sec, (parition,offset)=[(0,5328600)]
2016-08-11 15:10:29: insert 500 rows,  00.494065 sec, (parition,offset)=[(0,5328915)
(1,4833708)]
2016-08-11 15:10:30: insert 500 rows,  00.038252 sec, (parition,offset)=[(1,4834208)]
2016-08-11 15:10:30: insert 500 rows,  00.042458 sec, (parition,offset)=[(1,4834708)]
2016-08-11 15:10:30: insert 500 rows,  00.235408 sec, (parition,offset)=[(1,4835208)]
2016-08-11 15:10:30: insert 500 rows,  00.043084 sec, (parition,offset)=[(1,4835708)]

2016-08-11 15:10:41: insert 500 rows,  00.557918 sec, (parition,offset)=[(1,4855208)]
2016-08-11 15:10:42: insert 500 rows,  00.037498 sec, (parition,offset)=[(1,4855708)]
2016-08-11 15:10:42: insert 500 rows,  01.872249 sec, (parition,offset)=[(0,5329400)
(1,4855723)]
```

```
2016-08-11 15:10:50: insert 500 rows,  00.055253 sec, (parition,offset)=[(0,5338700)
(1,4855923)]
2016-08-11 15:10:50: insert 500 rows,  00.055402 sec, (parition,offset)=[(1,4856423)]
2016-08-11 15:10:50: insert 500 rows,  00.046360 sec, (parition,offset)=[(1,4856923)]
2016-08-11 15:10:50: insert 500 rows,  00.190940 sec, (parition,offset)=[(1,4857423)]
2016-08-11 15:10:50: insert 500 rows,  00.090820 sec, (parition,offset)=[(0,5339100)
(1,4857523)]
2016-08-11 15:10:50: insert 500 rows,  00.076443 sec, (parition,offset)=[(0,5339600)]
2016-08-11 15:10:50: insert 500 rows,  00.068591 sec, (parition,offset)=[(0,5340100)]
2016-08-11 15:10:51: insert 500 rows,  00.951319 sec, (parition,offset)=[(0,5340500)
(1,4857623)]
2016-08-11 15:10:53: insert 500 rows,  00.072213 sec, (parition,offset)=[(0,5340800)
(1,4857823)]
2016-08-11 15:10:56: insert 500 rows,  00.037657 sec, (parition,offset)=[(0,5341000)
(1,4858123)]
2016-08-11 15:10:58: insert 500 rows,  00.038033 sec, (parition,offset)=[(0,5341300)
(1,4858323)]
```

Consumer を追加（upserter.py）

Another consumer is added. The second consumer does "upsert" instead of "insert".

１つの Topic を読む Consumer を 2 group 実行することで、１つの Topic 上の message に対して複数の処理を並列して実行することができます。
例えば、１つの Consumer gorup "group 1" では、受け取った message を全て Teradata に insert し、詳細な History Table を作成します。
もう１つの Consumer group "group 2" は、Key Column に対して Upsert を実行し、常に各 Key Value に対して最新の情報だけを保持する Table を作成する、というような使い方ができます。



upserter.py の一部

```
SQL="""update continuous_upsert set datatime = ?, latitude= ? , longitude = ? where key_id
= ?
       else insert into continuous_upsert values (?,?,?,?)"""


def upsert_rows( session, nrow, values):
        param_array=[]
        offsets=''

        for m  in values:
                vs = m.value.split(delim)
                if debug > 10000000000000000:
                        print ( m.value )
                        print ( vs )
                param=( vs[1],vs[2],vs[3],vs[0],vs[0], vs[1], vs[2], vs[3] )
                param_array.append(param)

inserter.py
```

Producer として sender.py を2つ、Consumer として inserter.py と upserter.py を実行してみます。Consumer 側では、inserter には group "insert_process"、upserter には group "upsert_process" を指定しています。
なお、この環境では upsert の性能が落ちるため、各 sender は2秒毎に約100行の割合で message を投入しています。

```
[hisashi@mykdc kafkaTera]$ sender.py --topic Test2/0 --key_range 0,2500 --interval 2 &
[1] 2098
[hisashi@mykdc kafkaTera]$ sender.py --topic Test2/1 --key_range 2500,5000 --interval 2 &
[2] 2099
[hisashi@mykdc kafkaTera]$ jobs
[1]-  Running                 sender.py --topic Test2/0 --key_range 0,2500 --interval 2 &
[2]+  Running                 sender.py --topic Test2/1 --key_range 2500,5000 --interval 2
&
[hisashi@mykdc kafkaTera]$
[hisashi@mykdc kafkaTera]$ inserter.py --topic Test2/0,1 --group insert_process --end &
[3] 2102
[hisashi@mykdc kafkaTera]$ upserter.py --topic Test2/0,1  --group upsert_process --row 100
--end &
[4] 2103

[hisashi@mykdc kafkaTera]$ Setting up Teradata Connection....
Setting up Teradata Connection....
Setting up Kafka Connections....
Conntecting server, and subscribe Test2. group_id is upsert_process.
Setting up Kafka Connections....
Conntecting server, and subscribe Test2. group_id is insert_process.
[0, 1]
[0, 1]
[hisashi@mykdc kafkaTera]$ 2016-08-11 15:51:49: upsert 100 rows(100),  03.618871 sec,
(parition,offset)=[(0,5362100)]
2016-08-11 15:51:52: upsert 100 rows(100),  00.363381 sec, (parition,offset)=[(1,4859323)]
2016-08-11 15:51:53: insert 500 rows,  00.067963 sec, (parition,offset)=[(0,5362300)
(1,4859423)]
2016-08-11 15:51:53: upsert 100 rows(100),  00.301385 sec, (parition,offset)=[(0,5362200)]
2016-08-11 15:51:53: upsert 100 rows(100),  00.297038 sec, (parition,offset)=[(1,4859423)]
2016-08-11 15:51:53: upsert 100 rows(100),  00.468352 sec, (parition,offset)=[(0,5362300)]
2016-08-11 15:51:54: upsert 100 rows(100),  00.670420 sec, (parition,offset)=[(1,4859523)]
2016-08-11 15:51:55: upsert 100 rows(100),  01.129110 sec, (parition,offset)=[(0,5362400)]
2016-08-11 15:51:56: upsert 100 rows(100),  00.505967 sec, (parition,offset)=[(1,4859623)]
2016-08-11 15:51:57: insert 500 rows,  00.133445 sec, (parition,offset)=[(0,5362500)
(1,4859723)]
2016-08-11 15:51:57: upsert 100 rows(100),  02.218900 sec, (parition,offset)=[(0,5362500)]
2016-08-11 15:51:59: upsert 100 rows(100),  00.348552 sec, (parition,offset)=[(1,4859723)]
2016-08-11 15:51:59: upsert 100 rows(100),  00.738251 sec, (parition,offset)=[(0,5362600)]
2016-08-11 15:52:00: upsert 100 rows(100),  00.668554 sec, (parition,offset)=[(1,4859823)]
2016-08-11 15:52:01: upsert 100 rows(100),  00.327659 sec, (parition,offset)=[(0,5362700)]
2016-08-11 15:52:01: upsert 100 rows(100),  00.313172 sec, (parition,offset)=[(1,4859923)]
2016-08-11 15:52:03: insert 500 rows,  00.959588 sec, (parition,offset)=[(0,5362800)
(1,4859923)]
2016-08-11 15:52:03: upsert 100 rows(100),  03.735944 sec, (parition,offset)=[(0,5362800)]
2016-08-11 15:52:07: upsert 100 rows(100),  00.233658 sec, (parition,offset)=[(1,4860023)]
2016-08-11 15:52:07: insert 500 rows,  00.064570 sec, (parition,offset)=[(0,5363000)
(1,4860223)]
2016-08-11 15:52:07: upsert 100 rows(100),  00.262866 sec, (parition,offset)=[(0,5362900)]
2016-08-11 15:52:07: upsert 100 rows(100),  03.011491 sec, (parition,offset)=[(1,4860123)]
2016-08-11 15:52:10: upsert 100 rows(100),  00.285926 sec, (parition,offset)=[(0,5363000)]
2016-08-11 15:52:10: upsert 100 rows(100),  00.269760 sec, (parition,offset)=[(1,4860223)]
2016-08-11 15:52:11: upsert 100 rows(100),  00.258040 sec, (parition,offset)=[(1,4860323)]
```

```
2016-08-11 15:52:11: upsert 100 rows(100),  00.259812 sec, (parition,offset)=[(0,5363100)]
2016-08-11 15:52:13: insert 500 rows,  00.375172 sec, (parition,offset)=[(0,5363300)
(1,4860423)]
```

ログから、upsert は100行ごと、insert は500行毎に実行されています。また、ログには各処理での最終 message の offset が表示されていますが、ログ各行の offset の値から、inserter、upserter ともそれぞれが全ての message を読み込み、処理していることが確認できます。

Target Table

Target Table is partitioned by timestamp column and partition size is '1' minute.

この例のように連続的にデータを挿入し続けるとき、テーブルが空の時は高速な挿入が実現できますが、データが溜まるにつれ性能は劣化します。これに対する対策としては、 Parition Table を使って、常に空、或いはそれに近い状態の partition にデータを挿入させるようにすることで、性能の劣化を最小限にすることが可能です。今回の prototype では、ターゲット・テーブルは 10分 単位で partition を作成しました。今回のテストは TD Express で、非常に貧弱な環境なためこのような小さな partition としました。実環境の場合は、HWの性能や挿入されるデータ量などによって、partition サイズを決定する必要があるでしょう。

```
CREATE SET TABLE databasename.continuous_load ,NO FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO
     (
      key_id VARCHAR(12) CHARACTER SET LATIN NOT CASESPECIFIC,
      datatime TIMESTAMP(0),
      latitude DECIMAL(15,12),
      longitude DECIMAL(15,12))
PRIMARY INDEX ( key_id )
PARTITION BY RANGE_N(datatime  BETWEEN TIMESTAMP '2016-07-20 00:00:00+00:00' AND TIMESTAMP
'2016-12-31 23:59:59+00:00' EACH INTERVAL '1' MINUTE );
```

考察

今回のように連続してデータを挿入し続けたというような要件の場合、ここで紹介するようなアーキテクチャにすることのメリットとデメリットを考えてみました。

メリット

- Insert プロセスが 、上流側のデータ生成プロセス（Producer）の性能に影響を受けない
  - Insert のセッション数などは、純粋に Insert 性能のみで決めることが可能で、上流プロセスのスレッド数に依存しない
  - 上流プロセスは、必要に応じてスレッドを増やせる
- Message Broker が Buffer となり、上流、下流のプロセスにパフォーマンスの揺らぎなどが起きた場合でも吸収できる
- Kafka は一定期間データを保持できるので、後からの再処理も可能。メンテナンスなどへの対応が容易
- 各プロセスの 役割が Simple になるので、それぞれの実装が容易
- Cluster 構成とすることで、Fault Tolerant なシステムを構築可能
- Kafka 自体の性能は、Cluster を拡張することで容易に Scale 可能
- 新規のIOTデータの要件が現れても、容易に新規Streamを追加可能

デメリット

- Kafka を使ったシステムのProduction システムでの実装経験がない
- 追加のサーバー・インスタンスを必要とする
- システムの構成要素が増えることは、トラブルの要因を増やすことになるかも
- 今回の要件に対しては、オーバースペック？

Teradata RDBMS (by BAL Japan)