Evolution of an Apache Spark Architecture for Processing Game Data

Nick Afshartous
WB Analytics Platform

May 17th, 2017
About Me

• nafshartous@wbgames.com
• WB Analytics Core Platform Lead
• Contributor to Reactive Kafka
• Based in Turbine Game Studio (Needham, MA)

• Hobbies
  • Sailing
  • Chess
Some of our games...
Intro
• Ingestion pipeline
• Redesigned ingestion pipeline
• Summary and lessons learned
Problem Statement

• How to evolve Spark Streaming architecture to address challenges in streaming data into Amazon Redshift
Tech Stack

- Hadoop
- AVRO
- Apache Spark
- Apache Kafka
  
  A distributed streaming platform
Tech Stack – Data Warehouse

Amazon Redshift
Kafka

• Message is a (key, value)
• Optional key used to assign message to partition
• Consumers can start processing from earliest, latest, or from specific offsets
Game Data

- Game (mobile and console) instrumented to send event data
- Volume varies up to 100,000 events per second per game
- Games have up to ~ 70 event types
- Data use-cases
  - Development
  - Reporting
  - Decreasing player churn
  - Increase revenue
Ingestion Pipelines

• Batch Pipeline
  • Input       JSON
  • Processing  Hadoop Map Reduce
  • Storage     Vertica

• Spark / Redshift Real-time Pipeline
  • Input       Avro
  • Processing  Spark Streaming
  • Storage     Redshift
Spark Versions

- Upgraded to Spark 2.0.2 from 1.5.2

- Load tested Spark 2.1
  - Blocked by deadlock issue
    - [SPARK-19300](#) Executor is waiting for lock
• Intro

Ingestion pipeline
  • Re-designed ingestion pipeline
  • Summary and lessons learned
Process for Game Sending Events

Avro Schema -> Schema Registry

Schema Hash -> Schema Registry

Avro Data -> Event Ingestion

Schema Hash -> Event Ingestion

Returned hash based on schema fields/types

Registration triggers Redshift table create/alter statements
Ingestion Pipeline

- Event Avro Schema Hash
- HTTPS
- Event Ingestion Service
- Kafka
- Data topic
- Micro Batch
- Spark Streaming
- S3
- Run COPY
- Amazon Redshift
- Data flow
- Invocation
Redshift Copy Command

• Redshift optimized for loading from S3

```sql
create table if not exists public.person (
    id integer,
    name varchar
)
```

• COPY is a SQL statement executed by Redshift

• Example COPY

```sql
copy public.person from 's3://mybucket/person.txt'
```

1|john doe
2|sarah smith
Ingestion Pipeline

- Event Avro Schema Hash
- HTTPS
- Event Ingestion Service
- Data flow
- Kafka
- Micro Batch
- Spark Streaming
- S3
- Data topic
- Invocation
Challenges

• Redshift designed for loading large data files
  • Not for highly concurrent workloads (single-threaded commit queue)

• Redshift latency can destabilize Spark streaming
  • Data loading competes with user queries and reporting workloads

• Weekly maintenance
• Intro

• Ingestion pipeline

Redesigned ingestion pipeline

• Summary and lessons learned
Redesign The Pipeline

• Goals
  • De-couple Spark streaming job from Redshift
  • Tolerate Redshift unavailability

• High-level Solution
  • Spark only writes to S3
  • Spark sends copy tasks to Kafka topic consumed by (new) Redshift loader
  • Design Redshift loader to be fault-tolerant w.r.t. Redshift
Technical Design Options

• Options considered for building Redshift loader
  • 2\textsuperscript{nd} Spark streaming
  • Build a lightweight consumer
Redshift Loader

- Redshift loader built using Reactive Kafka
  - API’s for Scala and Java

- Reactive Kafka
  - High-level Kafka API
  - Leverages Akka streams and Akka
Akka

• Akka is an implementation of Actors
  • *Actors: a model of concurrent computation in distributed systems, Gul Agha, 1986*

• Actors
  • Single-threaded entities with an asynchronous message queue (mailbox)
  • No shared memory

• Features
  • Location transparency
    • Actors can be distributed over a cluster
  • Fault-tolerance
    • Actors restarted on failure

http://akka.io
Akka Streams

• Hard to implement stream processing considering
  • Back pressure – slow down rate to that of slowest part of stream
  • Not dropping messages

• Akka Streams is a domain specific language for stream processing
  • Stream executed by Akka
Akka Streams DSL

- Source generates stream elements
- Flow is a transformer (input and output)
- Sink is stream endpoint
Akka Streams Example

• Run stream to process two elements
  
  \[
  \text{val } s = \text{Source}(1 \text{ to } 2)
  \]

  \[
  s.\text{map}(x => \text{println("Hello: " + x)})
  .runWith(Sink.ignore)
  \]

  Nothing happens until run method is invoked

  Output
  
  Hello: 1
  Hello: 2

  Not executed by calling thread
Reactive Kafka

• Reactive Kafka stream is a type of Akka Stream

• Supported version is from Kafka 0.10+
  • 0.8 branch is unsupported and less stable

https://github.com/akka/reactive-kafka
Reactive Kafka – Example

- Create consumer config

```scala
implicit val system = ActorSystem("Example")
val consumerSettings = ConsumerSettings(system,  
  new ByteArrayDeserializer,  
  new StringDeserializer)
.withBootstrapServers("localhost:9092")
.withGroupId("group1")
Consumer.plainSource(consumerSettings, Subscriptions.topics("topic.name"))
  .map { message => println("message: " + message.value()) }
  .runWith(Sink.ignore)
```

- Create and run stream

Deserializers for key, value
Kafka endpoint
Consumer group

message has type ConsumerRecord (Kafka API)

Creates Source that streams elements from Kafka

- Create consumer config

- Create and run stream
Backpressure

• Slows consumption when rate is too fast for part of the stream

• Asynchronous operations inside `map` bypass backpressure mechanism
  • Use `mapAsync` instead of `map` for asynchronous operations (futures)
Revised Architecture

Game Clients

Event Avro

HTTPS

Event Ingestion Service

Data topic

Kafka

COPY topic

Spark Streaming

Redshift Loader

S3

Amazon Redshift

Copy Tasks

Data flow

Invocation

Copy Tasks
Goals

• De-couple Spark streaming job from Redshift

• Tolerate Redshift unavailability
Redshift Cluster Status

- Cluster status displayed on AWS console

- Can be obtained programmatically via AWS SDK
Redshift Fault Tolerance

• Loader Checks health of Redshift using AWS SDK
  • Start consuming when Redshift available
  • Shut down consumer when Redshift not available
    Consumer.Control.shutdown()

• Run test query to validate database connections
  • Don’t rely on JDBC driver’s Connection.isClosed() method
Transactions

• With auto-commit enabled each COPY is a transaction
  • Commit queue limits throughput

• Better throughput by executing multiple COPY’s in a single transaction

• Run several concurrent transactions per job
Deadlock

• Concurrent transactions create potential for deadlock since COPY statements lock tables

• Redshift will detect and return deadlock exception
Deadlock

Transaction 1
- Copy table A
- Copy table B

Transaction 2
- Copy table B
- A, B locked
- Copy table A
- Wait for lock

Deadlock
Deadlock Avoidance

• Master hashes to worker based on Redshift table name
  • Ensures that all COPY’s for the same table are dispatched to the same worker

• Alternatively could order COPY’s within transaction to avoid deadlock
• Intro
• Ingestion pipeline
• Redesigned ingestion pipeline

Summary and lessons learned
Lessons (Re) learned

• Distinguish post-processing versus data processing
  • Use Spark for data processing

• Assume dependencies will fail

• Load testing
  • Don’t focus exclusively on load volume
    • Number of event types
    • Distribution of event volumes
Monitoring

• Monitor individual components
  • Redshift loader sends a heartbeat via CloudWatch metric API

• Monitor flow through the entire pipeline
  • Send test data and verify successful processing
  • Catches all failures
Possible Future Directions

• Use Akka Persistence

• Implement using more of Reactive Kafka
  • Use Source.groupedWithin for batching COPY’s instead of Akka
Related Note

• Kafka Streams released as part of Kafka 0.10+
  • Provides streams API similar to Reactive Kafka

• Reactive Kafka has API integration with Akka
Imports for Code Examples

import org.apache.kafka.clients.consumer.ConsumerRecord
import akka.actor.(ActorRef, ActorSystem)
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.{Keep, Sink, Source}
import scala.util.{Success, Failure}
import scala.concurrent.ExecutionContext.Implicits.global
import akka.kafka.ConsumerSettings
import org.apache.kafka.clients.consumer.ConsumerConfig
import org.apache.kafka.common.serialization.StringDeserializer, ByteArrayDeserializer
import akka.kafka.Subscriptions
import akka.kafka.ConsumerMessage.{CommittableOffsetBatch, CommittableMessage}
import akka.kafka.scaladsl.Consumer