High performant in-memory datasets with Netflix Hollow

Who am I?

Roberto Perez Alcolea

- Mexican
- Streaming Platform @ Target
- Groovy Enthusiast
- roberto@perezalcolea.info
- @rpalcolea

Dataset Distribution

The problem

- Dissemination of small or moderately sized data sets (no big-data)

  Common approaches:

  - Sending data to a data store (RDMS, NoSQL)
  - Serializing and keeping a local copy

Is there a solution?


NETFLIX
OSS
H0110W

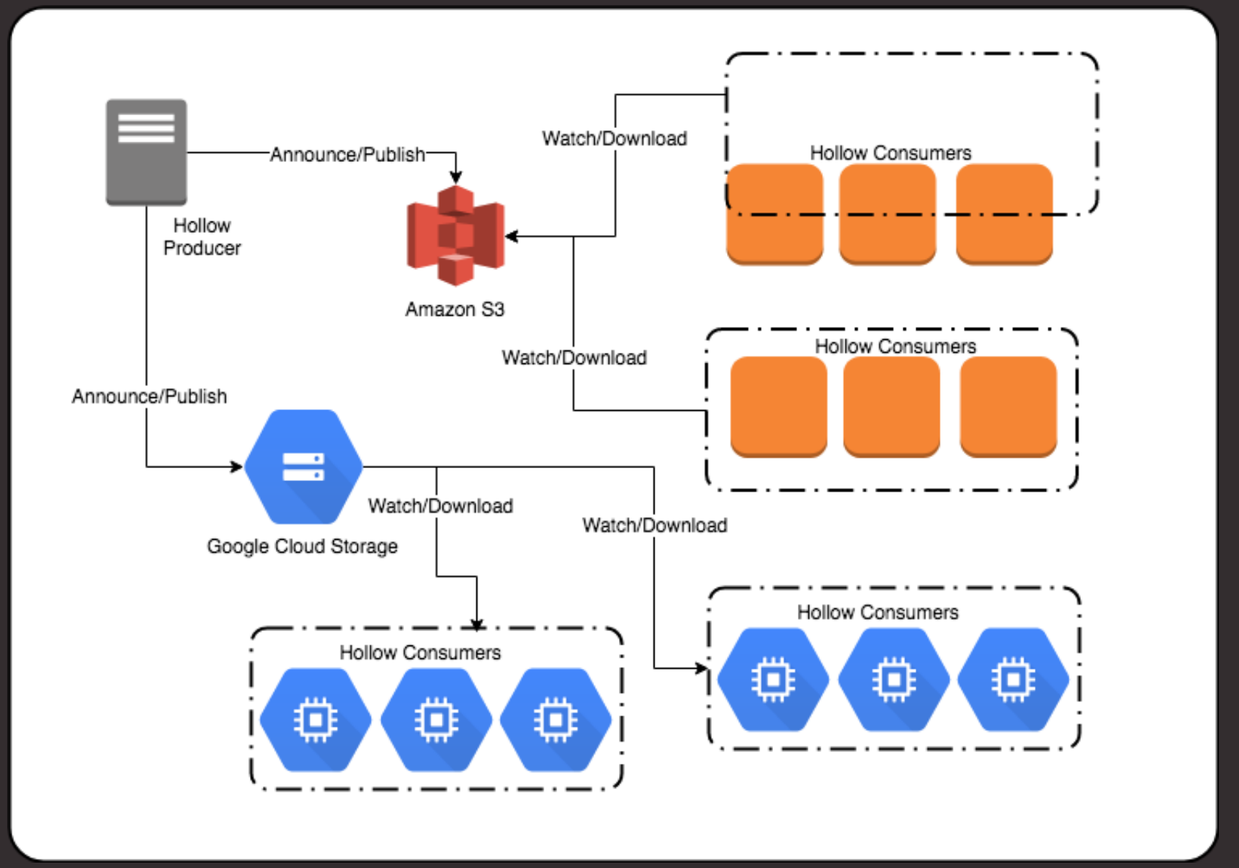
What is Hollow?

- Java library and toolset for disseminating in-memory datasets from a single producer to many consumers

  Goals:

  - Maximum development agility
  - Highly optimized performance and resource management
  - Extreme stability and reliability

How it works


Announce/Publish
Hollow
Producer
Amazon S3
Watch/Download
Hollow Consumers
Watch/Download
Hollow Consumers
Announce/Publish
Google Cloud Storage
Watch/Download
Watch/Download
Hollow Consumers
Hollow Consumers

Key concepts

- Blob / Blob Store
- Snapshot/Delta
- Type
- Deduplication

Key concepts

- State Engine
- Ordinal
  - unique identifier of the record within a type
  - sufficient to locate the record within a type
  - Immutability
- Cycle

Hollow Producer

- A single machine that retrieves all data from a source of truth and produces a delta chain
- Encapsulates the details of compacting, publishing, announcing, validating, and (if necessary) rollback of data states

Hollow Producer

```
class MyProducer {
  HollowProducer buildProducer() {
    ...
    HollowProducer
        .withPublisher(publisher)          /// required: a BlobPublisher
        .withAnnouncer(announcer)          /// optional: an Announcer
        .withValidators(validators)        /// optional: one or more Validator
        .withListeners(listeners)          /// optional: one or more HollowProducerListeners
        .withBlobStagingDir(dir)           /// optional: a java.io.File
        .withBlobCompressor(compressor)    /// optional: a BlobCompressor
        .withBlobStager(stager)            /// optional: a BlobStager
        .withSnapshotPublishExecutor(e)    /// optional: a java.util.concurrent.Executor
        .withNumStatesBetweenSnapshots(n) /// optional: an int
        .withTargetMaxTypeShardSize(size) /// optional: a long
        .build()
  }
}
```

Hollow Producer - Capabilities

- Restore at Startup
- Rolling Back
- Validating Data
- Compacting Data

Hollow Consumer

- Encapsulates the details of initializing and keeping a dataset up to date
  - At initialization time, loads snapshot
  - After initialization time, keeps a local copy of the dataset current by applying delta transitions
- Each time a new version is announced, `triggerRefresh()` should be called on the HollowConsumer

Hollow Consumer

```
class MyConsumer {
  HollowConsumer buildConsumer() {
    ...
    HollowConsumer
        .withBlobRetriever(blobRetriever)                /// required: a BlobRetriever
        .withLocalBlobStore(localDiskDir)                /// optional: a local disk location
        .withAnnouncementWatcher(announcementWatcher)    /// optional: a AnnouncementWatcher
        .withRefreshListener(refreshListener)            /// optional: a RefreshListener
        .withGeneratedAPIClass(MyGeneratedAPI.class)     /// optional: a generated client API class
        .withFilterConfig(filterConfig)                  /// optional: a HollowFilterConfig
        .withDoubleSnapshotConfig(doubleSnapshotCfg)     /// optional: a DoubleSnapshotConfig
        .withObjectLongevityConfig(objectLongevityCfg) /// optional: an ObjectLongevityConfig
        .withObjectLongevityDetector(detector)           /// optional: an ObjectLongevityDetector
        .withRefreshExecutor(refreshExecutor)            /// optional: an Executor
        .build()
  }
 }
```

Hollow Consumer API Generation

- We can initialize the data model using our POJOs

```
HollowAPIGenerator generator = new HollowAPIGenerator.Builder().withAPIClassname("BooksAPI")
                    .withPackageName("io.perezalcolea.hollow.api")
                    .withDataModel(Book.class)
                    .withDestination("./data-model/src/main/java")
                    .build();

            generator.generateSourceFiles();
```

Insight Tools

Hollow Explorer

- UI which can be used to browse and search records within any dataset

```
class MyExplorer {
  void startExplorer() {
    //Initialize consumer
    HollowConsumer hollowConsumer = HollowConsumerBuilder.build("publish-dir", BooksAPI.class)
    hollowConsumer.triggerRefresh()
    HollowExplorerUIServer server = new HollowExplorerUIServer(hollowConsumer, 8080)
    server.start()
    server.join()
  }
}
```

Hollow History

- UI which can be used to browse and search changes in a dataset over time

```
class MyHistory {
  void startHistory() {
    //Initialize consumer
    HollowConsumer hollowConsumer = HollowConsumerBuilder.build("publish-dir", BooksAPI.class)
    hollowConsumer.triggerRefresh()
    HollowHistoryUIServer server = new HollowHistoryUIServer(hollowConsumer, 8090)
    server.start()
    server.join()
  }
}
```

Heap Usage Analysis

- Given a loaded HollowReadStateEngine, it is possible to iterate over each type and gather statistics about its approximate heap usage

```
class MyMemoryInsight {
  void printMemoryUsage() {
    HollowReadStateEngine stateEngine = consumer.getStateEngine()

    long totalApproximateHeapFootprint = 0

    for(HollowTypeReadState typeState : stateEngine.typeStates) {
        String typeName = typeState.schema.name
        long heapCost = typeState.approximateHeapFootprintInBytes
        println(typeName + ": " + heapCost);
        totalApproximateHeapFootprint += heapCost
    }

    println("TOTAL: " + totalApproximateHeapFootprint)
  }
}
```

Other tools

- Metrics Collector
- Blob Storage Cleaner
- Filtering
- Combining/Splitting
- Patching

Interacting with a Hollow Dataset

Sample model

```
@HollowPrimaryKey(fields={"bookId"})
public class Book {
    long bookId;
    String title;
    String isbn;
    String publisher;
    String language;
    Set<Author> authors;
}

public class Author {
    long id;
    String authorName;
}
```

Indexing/Querying

Default Primary Keys

- Each type in our data model gets a custom index class called <typename>PrimaryKeyIndex
- Backed by Hollow Consumer
- Will automatically stay up-to-date as your dataset updates

```
class MyPrimaryKeyIndex {
  Book findBook(long bookId) {
    HollowConsumer hollowConsumer = //my consumer
    BookPrimaryKeyIndex bookPrimaryKeyIndex = new BookPrimaryKeyIndex(consumer)
    Book book = bookPrimaryKeyIndex.findMatch(bookId)
  }
}
```

Consumer-specified Primary Keys

- A primary key index is not restricted to just default primary keys

```
class MyPrimaryKeyIndex {
  Book findAuthor(long authorId) {
    HollowConsumer hollowConsumer = //my consumer
    AuthorPrimaryKeyIndex authorPrimaryKeyIndex = new AuthorPrimaryKeyIndex(consumer, "id")
    Author author = authorPrimaryKeyIndex.findMatch(authorId)
  }
}
```

Hash Index

- Records based on keys for which there is not a one-to-one mapping between records and key values
- Must specify each of a query type, a select field, and one or more match fields

```
class MyHashIndex {
   Iterable<Book> findBooksByPublisher(String publisher) {
    HollowConsumer hollowConsumer = //my consumer
    BooksAPIHashIndex publisherHashIndex = new BooksAPIHashIndex(consumer, "Book", "", "publisher.value")
    return publisherHashIndex.findMatch(publisher)
  }
}
```

Data Modeling

Primary Keys

- @HollowPrimaryKey annotation
- Provide a shortcut when creating a primary key index

```
@HollowPrimaryKey(fields={"bookId"})
public class Book {
    long bookId;
    String title;
    String isbn;
    String publisher;
    String language;
    Set<Author> authors;
}
```

Inlined vs Referenced Fields

- Inline: fields that they are no longer REFERENCE fields, but instead encode their data directly in each record

```
@HollowPrimaryKey(fields={"bookId"})
public class Book {
    long bookId;
    @HollowInline
    String title;
    String isbn;
    String publisher;
    String language;
    Set<Author> authors;
}
```

Namespaced Record Type Names

- Fields with like values may reference the same record type, but reference fields of the same primitive type elsewhere in the data model use different record types
- Types can be filtered

```
@HollowPrimaryKey(fields={"bookId"})
public class Book {
    long bookId;
    ...
    @HollowTypeName(name="Publisher")
    String publisher;
    @HollowTypeName(name="Language")
    String language;
    ...
}
```

Maintaining Backwards Compatibility

- Adding a new type
- Removing an existing type
- Adding a new field to an existing type
- Removing an existing field from an existing type.

Demo

Useful links

https://hollow.how/
https://github.com/Netflix/hollow-reference-implementation
https://github.com/Netflix/hollow

Thank You