A REPORT

ON

Modern Techniques for Effective Risk Management

BY

Name of the Student	ID. No
KUNAL KUMAR CHOWDHURY	2019HT12471

AT

Wells Fargo International Solutions Private LTD, Bangalore

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE

PILANI (RAJASTHAN)

April 2021

A REPORT

ON

Modern Techniques for Effective Risk Management

By

Name of the Student	ID No	Discipline
Kunal Kumar Chowdhury	2019HT12471	M. Tech. Software Systems

Prepared in partial fulfilment of the

WILP Dissertation

AT

Wells Fargo International Solutions Private LTD, Bangalore

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE

April 2021

ACKNOWLEDGEMENTS

I would like to thank my supervisor Mr. Rajat Pant for his guidance and support during the preparation of the dissertation. His valuable suggestions and direction helped me to enhance the quality of the work. I would also like to express my gratitude to the Risk and Pricing development team here in Wells Fargo for continuous recommendation and assistance.

Special thanks to Mr. K Girish for his additional support and guidance.

I am also grateful to Wells Fargo International Solutions Private LTD, Bangalore for giving me this opportunity to complete this dissertation.

Table of Abbreviations

Abbreviation	Meaning
AAPL	Ticker Symbol for Apple
FBSNN	Forward Backward Stochastic Neural Network
GAN	Generational Adversarial Network
GOOGL	Ticker Symbol for Google
GPU	Graphics processing unit
NFLX	Ticker Symbol for Netflix
PDE	Partial Differential Equation
QQQ	Ticker symbol for Invesco QQQ Trust Series 1
.SPX	Ticker symbol for S&P 500

Contents

Abstract

Derivatives risk management platform forms the basis of effective hedging and trading strategy of the front office trading unit in an investment bank. Various factors come into play to setup such an ecosystem, however, primarily there are two main constituents –

a. *Efficient Quantitative Models*
b. *Robust server- side pricing engine*

To put this into perspective, there are various trading desks in an investment bank or a hedge fund and each one of them has potentially dozens of asset types. Depending on the product type, the compute cycle of the most complicated asset type should not exceed ~1-2 seconds even in the most volatile market conditions.

Modern software principles suggest using micro services rather than a pure monolithic system for reliability, scalability, extensibility and maintenance.

The focus of this dissertation is mainly around (b) wherein we take a deep dive approach of design and implementation of various components of a real time pricing engine by using available open-source frameworks. Moreover for (a) we would be using ***QuantLib*** (Ballabio, QuantLib, n.d.)which is a C++ based pricing platform with a Java based wrapper. We keep an eye on the stability, reliability and low latency of the system while embarking on this path. The output from various test cases are used to demonstrate the working of these modules.

Although trading forms the driving force behind the revenue generation from a front office perspective, the middle office (Limited, 2006) does need to take care of the accumulated liability by holding positions on risky assets and exposing these to severe market circumstances. These are called scenario reports and they mostly mimic the difficult market conditions e.g., 2008 market crisis (Dempster, 2002). These scenarios specify the market parameters like interest rate, volatility etc. For the portfolio of holdings held by the desk, it must calculate the price of these individual trades under the given parametric setting. As these conditions are manually fed to the system from regulatory authorities it becomes a dependency on an upstream system.

The second part of the dissertation focusses on a way to generate such conditions based on the asset types using deep learning in conjunction with the hypothesis used behind some modeling techniques in pricing derivatives risk.

1. Introduction

1.1 Background on Derivatives

As stated in (Hull & Basu, 2016) a derivative can be defined as a financial instrument whose value depends on (or derives from) the values of other, more basic or underlying variables. As an example, an option on a single stock is dependent on the stock price, volatility, current interest rate, yield curve etc.

Therefore, we must note that we are referring to two different prices here i.e., the price of the underlying or the stock and price of the option or the derivative.

Obviously, derivative prices can change with respect to any of the underlying variables and as such apart from the price, the rate of change of the option (or any derivative) price with respect to the underlying, volatility, interest rate and time are respectively known as *delta, vega, rho and theta*. The rate of change of delta with respect to the underlying is called *gamma* (this is the second derivative with respect to underlying price). These are called the Greeks for the derivative and details regarding the same can be found in any treatise on the subject e.g., (Hull & Basu, 2016)

Options can be further divided into various types depending on the underlying nature of the contract and time of execution–

- A *call* / *put* option gives the holder the right to *buy/sell* the underlying asset by a certain date for a certain price.
- Depending on the date of expiry of the contract *an American / a European* contract is one that can be executed *any time up to the expiration/ on the date* of expiration.

Option pricing can be done using various methods, but some well-known among them are Black Scholes (and its variants for European) and lattice-based methods viz. binomial, trinomial trees etc. for American options (Hull & Basu, 2016). However, there are many other variations of these pricing algorithms that are used in the industry on a need basis.

The underlying contract of the derivatives can be further customized by the traders. To extend this concept, there are some traders who specialize on a certain kind of underlier(s) e.g. option contracts solely based on "*.SPX*" index (S&P 500, n.d.) or maybe on a collection of underlying stocks which are actively traded on *New York Stock Exchange* (NYSE, n.d.) or any such trading exchange across the world.

This gives rise to the concept of desks where each desk is a group of traders focused on a certain kind of underliers or maybe associated with some specific contracts (Allen, 2012).

1.2 Stress Tests

Corporate governance demands that an investment bank must also follow regulatory practices and assure the investors and the customers that it has enough liquidity in the event of any market crisis. (Szylar, 2013), (Dempster, 2002)

Since the trading book positions also add to the risky assets in the bank's liabilities it must simulate various scenarios at the end of day to evaluate the value at risk.

These scenarios define the parameters that need to be bumped up or down and subsequently these must be run as separate pricing request and their prices validated by market risk officers.

However, one of the known issues in this setup is that these scenarios are always fed from a separate system and this introduces a dependency. Rather than having such a reliance, we can explore separate avenues to figure out their generation using deep learning techniques and proactively evaluate the same in the absence of clearly spelled out directives.

1.3 Pricing Engine

Derivative prices change rapidly during the day, due to the factors mentioned above and every desk must monitor the price of their holdings and effectively hedge the associated risk if any. For this purpose, continuous pricing using the algorithms mentioned above are essential.

A continuous pricing engine would need parameters to be fed to the pricing algorithms from various sources that generate the market data like spot prices and volatility surfaces whenever they tick/change. And few of them may depend on custom rules that the traders prefer e.g., *a trader A may prefer to use a customized volatility surface depending on the underlier. Or, after market close, we may like to use a separate key for the underlier prices.* These bring in the necessity of a rule engine that is flexible enough to be used by the traders as well as to be integrated in the main pricing code.

Bottomline is - a convoluted rule engine will make things worse to track when we evaluate rules.

Another dimension to lookout is the associativity between the price change of a stock and its derivative.

An underlier can be a part of multiple options or derivative agreements and these instruments may take varied amount of time to price with each tick in the market. If these prices are not shown consistently i.e., if the derivative prices are displayed to the trader independently it would be insufficient for him/her to effectively hedge in the market as he/she would be possibly grouping those instruments by various attributes and one of them would be the underlier. So, this also falls as a necessary design requirement.

Many times a pricing process may become irresponsive, and a new process may need to take over. For this reason, a persistent snapshot of the market data is essential. However, this snapshot should not be merely a cache or a database of the incoming market data. This kind of approach

would increase the latency, as only a fraction of the total input data gets modified during each tick cycle.

1.4 Software Frameworks and Pricing Library

Throughout our implementation we would be using a few standard Java based opensource frameworks for achieving the following –

A. Containerization – Spring Boot (Spring Boot, n.d.)
B. Messaging – Kafka (Kafka, n.d.)
C. Persistence – Redis Cache (Redis, n.d.)
D. Event generation and processing – Rx Java (Nurkiewicz, Christiansen, & Meijer, 2016) (Samoylov & Nield, 2020)

For running deep learning implementation, we would be using Tensorflow/Keras (Keras, n.d.) (Tensorflow, n.d.) in Python.

As the focus of this discussion is around the software design and implementation, we are going to use QuantLib (Ballabio, QuantLib, n.d.) as the preferred library for calculating the risk and pricing numbers.

QuantLib is an opensource framework and details about the same can be found at (Ballabio, Implementing QuantLib: Quantitative Finance in C++: an Inside Look at the Architecture of the QuantLib Library, 2020)

However, in a real investment bank setup there would be a separate Quants team who would design these algorithms in collaboration with the desk traders. (Tulchinsky, 2015) (Patterson, 2010)

1.5 Riskserver

To differentiate between the actual server-side design, implementation of the project about which this work is all about and the underlying QuantLib (Ballabio, QuantLib, n.d.) and other opensource frameworks, we have named this implementation as ***Riskserver*** in the rest of our discussion.

1.6 Design considerations

1.6.1 Distributed System Design

The Chandy Lamport algorithm as described here(Chandy & Lamport, 1985) defined the steps to capture the consistent global state of a distributed system. The initiator process saves its local state and sends a special marker message along its outgoing channels to other processes from where it wants to receive the snapshot. Any process that receives this marker message first sends it state information and recursively sends a similar request through its outgoing channel.

In case the processes have already sent the snapshot previously it can send the differential updates between the two snapshot timing requests.

This approach can be used to capture the local market data updates and generate a pricing snapshot when a predesignated time arrives to perform a pricing valuation. This can be a strategy implementation or even a vanilla timer.

However, for all this to work we must put a timestamp on every message so that we can effectively manage the snapshot timings.

In the book (Akidau, Chernyak, & Lax, 2018), the authors have analyzed the details on the concept of Event time and Processing time. The first one refers to the moment when the event occurred whereas the second alludes to when these are observed in the system. For the purpose of our discussion, we would be largely concerned with processing time.

Closely related to timestamp is the concept of windowing. This defines how we group the updates and again in (Akidau, Chernyak, & Lax, 2018), the authors have described various kinds of windowing methods like fixed, sliding windows etc. in details.

1.6.2 Rule engine

Rete's algorithm (Rete algorithm, n.d.) conceptualizes special nodes in a graph for parsing rules based on entities. Each entity is first matched to its type and evaluated and is called a match or alpha node. These are stored in memory, referred to as alpha memory. There are other nodes like join nodes that perform conditional operations on top of the alpha nodes. Such an approach increases the overall performance by storing the partially evaluated information and then using the same when processing the actual data. Thus, revaluation is not needed for every rule.

Our idea is to customize this concept to evaluate trading rules related to spot, volatility surface and other market data.

1.6.3 Stress scenario generator

1.6.3.1 Forward Backward Stochastic Neural Network (FBSNN)

In the paper (Raissi, Forward-Backward Stochastic Neural Networks: Deep Learning of High-dimensional Partial Differential Equations, 2018) (also (Laignelet, 2019)), there is a detailed explanation of design and implementation of Forward Backward Stochastic Neural Network in Python (Raissi, FBSNNs, 2018)

As mentioned in (Wilmott, Dewynne, & Howison, 1994) to derive the price of an option one can use the following standard *Black Scholes* partial differential equation.

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \qquad Eq\ (1)$$

The above is then transformed to the heat equation (initial value problem).

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad , \qquad Eq\ (2)$$

$$t > 0\ , -\infty < x\ < \infty,$$

$$with\ initial\ data\ u(x,0) = u_0(x)\ \ and\ u \to 0\ as\ x \to \pm\infty$$

However, in cases where we have special boundary conditions, we need to handle them accordingly e.g., pricing a European put option in eq. (2)

As explained in the references there are primarily four equations to connect the stochastic process X_t (underlier e.g., spot prices), volatility σ , rate of interest r

$$\begin{aligned} dX_t &= \sigma\ diag(X_t)\ dW_t \\ X_0 &= \xi \\ dY_t &= r\ (Y_t - Z_t^T X_t)dt + \sigma Z_t^T diag(X_t) dW_t \\ Y_T &= g(X_t) \end{aligned}$$

And these are equivalent to the Black Scholes with the following condition,

$$\begin{aligned} Y_t &= u(t, X_t) \quad and \\ Z_t &= \nabla u(t, X_t)\sigma(t, X_t) \end{aligned}$$

In the paper (Laignelet, 2019) the conditions assumed are.

$$\begin{cases} g(x) = x^2\ [\,terminal\ condition] & Eq\ (3) \\ u(x,t) = e^{(r+\sigma^2)(T-t)}g(x) & Eq\ (4) \end{cases}$$

Here $u(x,t)$ is a solution of the Black Scholes equation as it satisfies *Eq* (1).

The same has been used in the Python implementation by the authors. (Raissi, FBSNNs, 2018)

1.6.3.2 Generational Adversarial Network (GAN)

In the paper (Goodfellow, et al., 2014), the authors have proposed the concept of Generational Adversarial Network. The purpose of this framework is to generate samples by passing random noise through a deep learning dense network and a discriminative model which also is a multilayer perceptron.

Thus, to summarize we can use a dense network generator that can generate samples and composition of a generator and discriminator to filter out the correct samples. On the other hand, in the initial stage, the discriminator is trained to reject everything that is produced by the generator.

Figure 1-1 High level GAN overview

1.6.3.3 Scenario Generator using GAN based FBSNN using function pairs:

Stress test is the process of simulating tough conditions in the market by manipulating parameters like interest rate, volatility etc. and using them to price the current portfolio of holdings.

As described in 1.6.3.1, FBSNN (Raissi, FBSNNs, 2018) is already designed to output portfolio prices for a high dimensional input vector of underlying prices.

Eq (3) can now serve as the loss function of the generator component of 1.6.3.2 along with *Eq* (4) as the true value for the output.

For discriminator, we have designed the following functions.

$$\begin{cases} g(x) = e^{\frac{-1}{2}\left(\frac{2r}{\sigma^2}-1\right)x} & Eq\ (5) \\ u(x,\tau) = e^{-\tau\left[\frac{1}{4}\left(\frac{2r}{\sigma^2}+1\right)^2+\left(\frac{\pi}{T}\right)^2\right]}\sin\frac{\pi x}{T}\,g(x) & Eq\ (6) \end{cases}$$

Although this is a sample function pair it uses the ratio of rate of interest r and volatility σ and as such the scenario conditions can be controlled better.

It will be worthwhile to mention that most of the derivatives have a corresponding *Partial Differential Equation* to *Heat Equation* transformation and as such it is always possible to have a functional approximation as mentioned above.

2. Business Entities and Rules

For pricing any derivative asset type, we need the following input parameters (Hull & Basu, 2016)

a. *Instrument data e.g., Vanilla Option, Basket Option*
b. *Parameters e.g., spot price, volatility surface, dividend curve*
c. *User defined rules*

A sample pricing invocation in QuantLib (Ballabio, QuantLib, n.d.) needs the first two parameters and these are implemented in C++. To use the same in Java we need a JNI wrapper to translate the Java objects into corresponding C++ objects and call the necessary C++ pricing functions.

As a first attempt to build the end-to-end framework for pricing derivatives we define the high-level project modules. They are as follows –

a. Common - *Plain maven project*
b. Marketdata - *Spring Boot application*
c. Snapshot-orchestrator - *Spring Boot application*
d. Engine - *Spring Boot application*

We first investigate the class diagrams of some standard business objects used in QuantLib and subsequently define a strategy to use these in the overall framework.

2.1 Instrument Object

The class diagram of VanillaOption is -

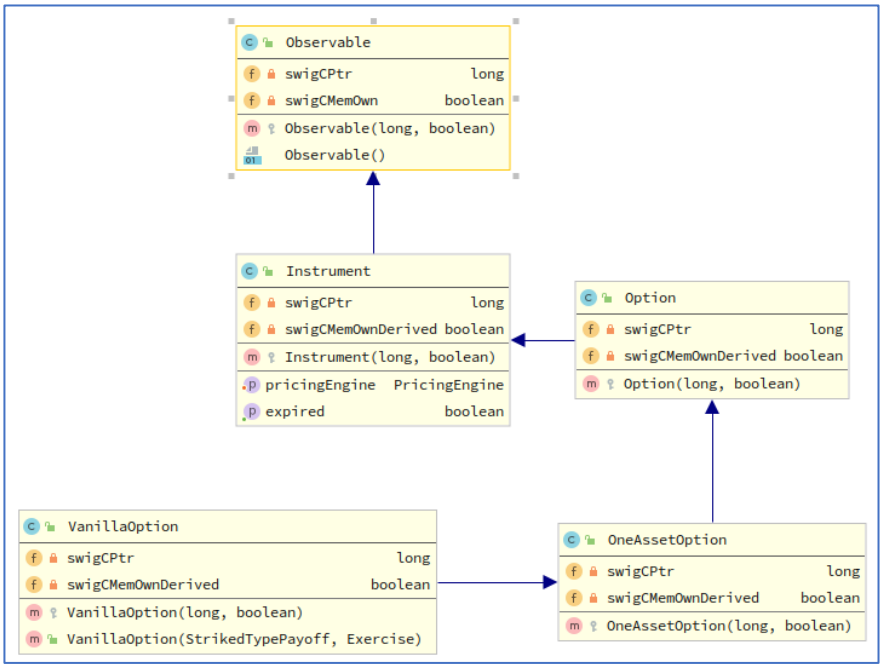

Figure 2-1 - Class Diagram of VanillaOption in QuantLib

2.2 Parameters

Following is the class diagram of BlackVarianceSurface that would be used in our discussion.

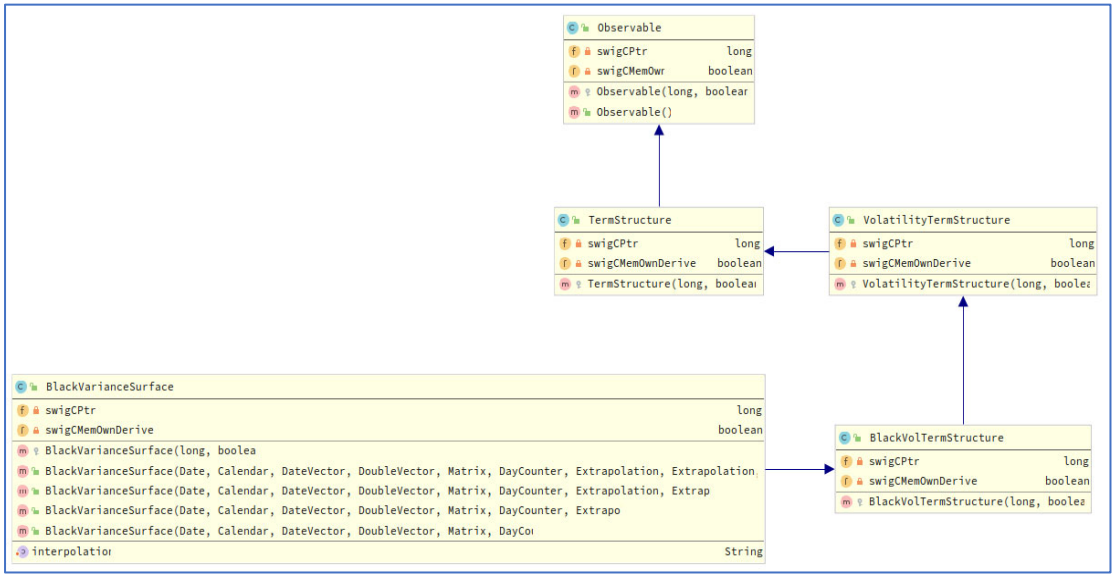

Figure 2-2- Class Diagram of BlackVarianceSurface in QuantLib

Lastly, we would also need the discount curve in the pricing process and following demonstrates the class diagram of DiscountCurve.

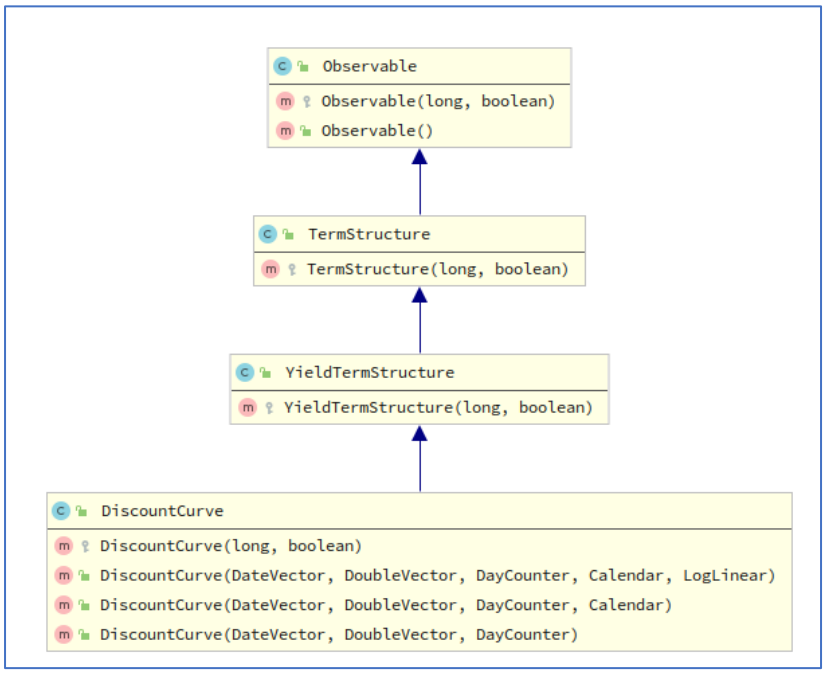

Figure 2-3 Class Diagram of DiscountCurve in QuantLib

2.3 Builder Classes and Differential Updates in RiskServer

2.3.1 Builder Classes

We use the Builder design pattern (Gamma, Helm, Johnson, & John, 1994) to create the instrument and parameter objects required for valuations.

This also gives us the flexibility to alter the parameters at a later stage and helps in extensibility.

Using abstraction, we take out the piece of code that has common functionality. Also, there may be a need to refer to these builder classes by a common reference.

As an example, in the our code both *FlatVolSurfaceBuilder* and *BlackVarianceSurfaceBuilder* extend from *VolatilitySurfaceBuilder*.

The final *PricingEngineProcess* in QuantLib (Ballabio, QuantLib, n.d.) will be the standard *BlackScholesMertonProcess* (Ballabio, QuantLib, n.d.). However, the underlying algorithm for pricing can be *Binomial CRR, Binomial J4, Binomial LR* etc. as designed in (Ballabio, QuantLib, n.d.)

In this case, as the algorithms themselves are the pricing engines by virtue of the JNI QuantLib wrapper, we can have separate instances for each algorithm, however if we have the flexibility to write the logic of the algorithms separately one can go for Decorator pattern to implement the concrete pricing engines (Gamma, Helm, Johnson, & John, 1994), (Bloch, 2016).

Some of the other important Builder classes in our implementation are –

Builder	Responsible for creating
InstrumentBuilder	Instruments e.g., Vanilla Option.
VolatilitySurfaceBuilder	Volatility surfaces like Flat or Black variance surface.
YieldTermStructureBuilder	Dividends and Yield Curves.
BlackVolTermStructureBuilder	BlackVol Term structure which is a wrapper over VolatilitySurfaceBuilder.
BlackScholesMertonProcessBuilder	Composition of builders of dividend, yield curves, volatility surface.

Table 2.1 Builder classes for valuation

2.3.2 Differential Updates

The calculated risk of derivative instruments vary throughout the day (intraday) based on the market movement of the underliers as discussed in *Introduction (Section – 1.1).*

As most of the market data remains same when spot price or a part of the volatility surface changes, transmitting the full entity over wire (serialization/deserialization) or persisting these will lead to storing redundant data. And eventually it impacts the performance.

In Figure 2.2, we have depicted the structure of the volatility surface class that contains a reference to Matrix (Ballabio, QuantLib, n.d.) which is a 2-dimensional array containing the volatility values at various points in time. Matrix can contain more than 500 elements and to store such a huge structure when only a few elements have changed is not a good design.

To implement the "Differential Updates" feature in RiskServer we define the following annotation. All the RiskServer Java classes used in valuation would be annotated with BaseEntity.

```
@BaseEntity
public class VanillaOption extends
Instrument{

    private Double strike ;
    private String underlying ;
    private Double riskFreeRate ;
    private Double dividendYield;
    private Double volatility ;
    private Long settlementDate;
```

Figure 2-4 BaseEntity annotation for VanillaOption

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface BaseEntity {
}
```

Figure 2-5 BaseEntity annotation

We now define a class EntityMetadataRepository that is used to hold all the base entity classes and their relevant return-type information for each setter function.

Logic for this implementation –

```
public class EntityMetadataRepository {

    public static EntityMetadataRepository INSTANCE = new
EntityMetadataRepository();
    private final Map<String, Map<String , Class<?>>> functionMap;

    private EntityMetadataRepository() {
        ClassPathScanningCandidateComponentProvider scanner =
                new ClassPathScanningCandidateComponentProvider(false);
        scanner.addIncludeFilter(newAnnotationTypeFilter(BaseEntity.class));
        Set<BeanDefinition> definitions =
scanner.findCandidateComponents("com.q");
        this.functionMap =

definitions.stream().collect(Collectors.toMap(BeanDefinition::getBeanClassName,
b -> {
            try {
                return
Arrays.stream(Class.forName(b.getBeanClassName()).getDeclaredMethods())
                        .filter(m -> m.getName()
                                .startsWith("get")).collect(Collectors.toMap(k
-> k.getName()
                                .replaceAll("get", "").toLowerCase(),
Method::getReturnType));
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            return Maps.newHashMap();
        }));
    }

    public Map<String, Map<String, Class<?>>> getFunctionMap() {
        return functionMap;
    }
}
```

Figure 2-6 Sample Implementation of EntityMetadataRepository

In addition to this implementation, we also defined the standard contract called the DeltaEntity interface. Following demonstrates the use of this interface in context of *BlackVarianceVolSurface*.

```
public interface DeltaEntity<T extends
Entity> {
    String getEntityName();
    String getFieldName();
    String getFieldValue();
    int[] getFieldIndexes();
    String[] getFieldValues();
    boolean isFieldUpdate();
}
```

Figure 2-7 DeltaEntity interface

We now show a sample implementation of the code that is used to create the modified entity.

```
public <T extends Entity> T getModifiedEntity(T baseEntity, DeltaEntity<T>
deltaEntity)
            throws InvocationTargetException, IllegalAccessException {
        Map<String, Class<?>> stringClassMap =
EntityMetadataRepository.INSTANCE.getFunctionMap().get(deltaEntity.getEntityName(
));
        Class<?> aClass = stringClassMap.get(deltaEntity.getFieldName());
        Object value = getValue(aClass.getCanonicalName(),
deltaEntity.getFieldValue());
        if (deltaEntity.isFieldUpdate()) {
            Optional<Method> method =
Arrays.stream(baseEntity.getClass().getDeclaredMethods()).
                    filter(m -> StringUtils.equalsIgnoreCase("set" +
deltaEntity.getFieldName(), m.getName()))
                    .findFirst();
            if (method.isPresent()) {
                method.get().invoke(baseEntity, value);
            }
        }else {
            Optional<Method> method =
Arrays.stream(baseEntity.getClass().getDeclaredMethods()).
                    filter(m -> StringUtils.equalsIgnoreCase("setArray" +
deltaEntity.getFieldName(), m.getName()))
                    .findFirst();
            if (method.isPresent()) {
                int[] fieldIndexes = deltaEntity.getFieldIndexes();
                if (fieldIndexes.length == 1) {
                    method.get().invoke(baseEntity, fieldIndexes[0] ,value);
                } else if (fieldIndexes.length == 2) {
                    method.get().invoke(baseEntity, fieldIndexes[0],
fieldIndexes[1]  ,value);
                }
            }
        }
        return baseEntity;
```

Figure 2-8 Sample Implementation of getModifiedEntity

It is defined in a class called ReflectionUtils.java. This utility class also has optimization to cache metadata and other frequently used information.

Sample test case to validate the DeltaEntity implementation –

```
@Test
public void testDeltaEntity(){
        java.util.Calendar calendar = java.util.Calendar.getInstance();
        zeroAll(calendar);
        calendar.set(15, java.util.Calendar.MAY, 15);
        long valuationDate = calendar.getTime().getTime();

        Long[] expirations = new Long[5];
        set(calendar, 2013, java.util.Calendar.DECEMBER, 20, expirations, 0);
        set(calendar, 2014, java.util.Calendar.JANUARY, 17, expirations, 1);
        set(calendar, 2014, java.util.Calendar.MARCH, 21, expirations, 2);
        set(calendar, 2014, java.util.Calendar.JUNE, 20, expirations, 3);
        set(calendar, 2014, java.util.Calendar.SEPTEMBER, 19, expirations, 4);

        Double[][] vols = new Double[][]{
                {0.15640, 0.15433, 0.16079 , 0.16394, 0.17383},
                {0.15343, 0.15240, 0.15804 , 0.16255, 0.17303},
                {0.15128, 0.14888, 0.15512 , 0.15944, 0.17038},
                {0.14798, 0.14906, 0.15522 , 0.16171, 0.16156},
                {0.14580, 0.14576, 0.15364 , 0.16037, 0.16042}
        };
        BlackVarianceVolatilitySurface vol = new
BlackVarianceVolatilitySurface(100,
                "QQQ_Black_Variance", 1,
                valuationDate,
                valuationDate,
                USMarketType.NYSE,
                expirations,
                new Double[]{1650.0, 1660.0, 1670.0, 1675.0, 1680.0},
                DayCount.ACTUAL_365_FIXED,
                vols);

        BlackVarianceVolSurfaceDelta delta = new
BlackVarianceVolSurfaceDelta("strikes",
                "1660.3", new int[]{2}, null,false);

        Assert.assertEquals(1670.0, vol.getStrikes()[2], 0);
        try {
            BlackVarianceVolatilitySurface modifiedEntity =
ReflectionUtils.INSTANCE.getModifiedEntity(vol, delta);
            Assert.assertEquals(1660.3, vol.getStrikes()[2], 0);
        } catch (InvocationTargetException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
```

Figure 2-9 Sample DeltaEntity Test case

The metadata repository class holds in-memory, the details about the classes that have BaseEntity annotation. Thereafter the `getModifiedEntity` uses the information to invoke the corresponding setter method. In case the method is having the parameter as an array the same is invoked along with the applicable indices. We leverage the `AnnotationTypeFilter` from Spring framework (Framework, 2014), (Spring Framework, n.d.) and rest of the implementation mainly relies on normal Java reflection (Bloch, 2016).

2.4 Rule Engine Implementation in RiskServer

We define the following nodes based on Rete's algorithm (Rete algorithm, n.d.) -

i. Root Node – consumes the rule universe.
ii. Type Node – segregates the type of objects e.g., instrument and volatility surface.
iii. Join Node – join the flow of instrument source and evaluate the relevant predicates along with the cache generated using Type Nodes.
iv. Terminal Node – return the final key for the instrument for volatility surface.

A root node is initialized using a rule processor listener and whenever a new rule is parsed it invokes the underlying incoming subject from RxJava (Nurkiewicz, Christiansen, & Meijer, 2016). As an example, if the rules we are interested are for volatility surface based on the instrument attributes we will have two type nodes – one for Instrument and the other for Volatility Surface. Both type nodes will then process the updates separately. This is useful as it does not need these processes to be synchronized.

Once done they will update their local cache in the Type Nodes. Each node maintains its own local cache. For the current implementation we are using Guava Cache (Guava, n.d.).

```
@Component @ComponentScan(basePackages = "com.q")
public class RootNode<T extends Entity, U extends Entity> implements Processor {
    private final RuleProcessorListener<T, U> listener;
    private final Pair<TypeNode<T, U>, TypeNode<T, U>> typeNodeTypeNodePair;
    private final Cache<Integer, EntityRule<T, U>> ruleCache;
    private final int level;
    public RootNode(@Autowired RuleProcessorListener<T, U> listener, @Autowired
ConfigProperties configProperties) {
        this.listener = listener;
        this.level = configProperties.getLevel();
        this.typeNodeTypeNodePair = Pair.of(new
TypeNode<>(TypeNode.TypeParam.Instrument, PublishSubject.create(), level),
                new TypeNode<>(TypeNode.TypeParam.Entity, PublishSubject.create(),
level));

        this.ruleCache = CacheBuilder.newBuilder().concurrencyLevel(level).build();
    }
    @Override
    public void exec() {
        PublishSubject<EntityRule<T, U>> incomingSubject =
this.listener.getSubject();
        Observable<EntityRule<T, U>> source = incomingSubject.asObservable();
        source.subscribe(e ->  {this.ruleCache.put(e.getRuleIdentifier(), e); });

source.subscribe(typeNodeTypeNodePair.getLeft().getEntityRulePublishSubject());

source.subscribe(typeNodeTypeNodePair.getRight().getEntityRulePublishSubject());
    }
```

Figure 2-10 Sample Root Node Implementation

```
public class TypeNode<T extends Entity, U extends  Entity> implements
Processor{
    private final TypeParam type;
    private final Cache<Integer, Function<T, Predicate<T>>>
instrumentPredicateCache;
    private final PublishSubject<EntityRule<T, U>> entityRulePublishSubject;
    private final Cache<Integer, Function<Pair<T, U>, Function<T, String>>>
parameterFunctionCache;
    public TypeNode(TypeParam type, PublishSubject<EntityRule<T, U>>
entityRulePublishSubject, int concurrency) {
        this.type = type;
        this.entityRulePublishSubject = entityRulePublishSubject;
        this.instrumentPredicateCache =
CacheBuilder.newBuilder().concurrencyLevel(concurrency).build();
        this.parameterFunctionCache =
CacheBuilder.newBuilder().concurrencyLevel(concurrency).build();
    }
    @Override
    public void exec() {
        switch (type) {
            case Instrument:
                entityRulePublishSubject.map(e ->
Pair.of(e.getRuleIdentifier(),
                        e))
                        .subscribe(e -> {
                            instrumentPredicateCache.put(e.getLeft(), t -> {
                                e.getRight().from(t, null);
                                return e.getRight().getPredicate();
                            });
                        });

                break;
            case Entity:
                entityRulePublishSubject.map(e ->
Pair.of(e.getRuleIdentifier(),
                        e))
                        .subscribe(e -> {
                            parameterFunctionCache.put(e.getLeft(), tuPair -> {
                                e.getRight().from(tuPair.getLeft(),
tuPair.getRight());
                                return e.getRight().getKeyTransform();
                            });
                        });

                break;
        }
    }
```

Figure 2-11 Sample Type Node Implementation

For a given instrument instance we evaluate the rules by using the Join Node as shown below. We should note that more than one rule can satisfy based on the predicates, however as each rule has an associated weight, we sort them and take the one with the highest weight.

The applicable rule is cached for each instrument and a given parameter.

```
public class JoinNode<T extends Entity, U extends Entity> implements
Function<T, Pair<T, Integer>> {
    private final RootNode<T, U> rootNode ;
    private final TypeNode<T, U> typeNode ;

    private final Cache<Integer, EntityRule<T, U>> ruleCache;

    public JoinNode(RootNode<T, U> rootNode, TypeNode<T, U> typeNode, int
concurrency) {
        this.rootNode = rootNode;
        this.typeNode = typeNode;
        this.ruleCache =
CacheBuilder.newBuilder().concurrencyLevel(concurrency).build();
    }
    @Override
    public Pair<T, Integer> apply(T t) {
        List<EntityRule<T, U>> rule = typeNode.getPredicatesAsMap()
                .entrySet()
                .stream()
                .filter(e -> e.getValue().apply(t).test(t))
                .map(e -> rootNode.getRule(e.getKey()))
                .filter(Optional::isPresent)
                .map(Optional::get).sorted((o1, o2) -> o2.getRuleWeight() -
o1.getRuleWeight()).collect(Collectors.toList());
        ruleCache.put(t.getId(), rule.get(0));
        return Pair.of(t, rule.get(0).getRuleIdentifier());
    }

    public Optional<EntityRule<T, U>> getApplicableRule(int id){
        return Optional.ofNullable(ruleCache.getIfPresent(id));
    }
```

Figure 2-12 Sample Join Node Implementation

Finally, for a given Instrument and parameter (e.g., volatility surface) the Terminal Node will return the applicable key for the instrument.

```
public class TerminalNode <T extends Entity, U extends Entity>
implements Function<Pair<Pair<T, U>, Integer>, String> {
    private final TypeNode<T, U> typeNode;

    public TerminalNode(TypeNode<T, U> typeNode) {
        this.typeNode = typeNode;
    }

    @Override
    public String apply(Pair<Pair<T, U>, Integer> pair) {
        Optional<Function<Pair<T, U>, Function<T, String>>>
keyTransform = typeNode.getKeyTransform(pair.getRight());
        return keyTransform.map(func ->
func.apply(pair.getLeft())

.apply(pair.getLeft().getLeft())).orElse(StringUtils.EMPTY);
    }
}
```

Figure 2-13 Sample Terminal Node Implementation

Rule are stored in Redis Cache (Redis, n.d.) and parsed using ANTLR (antlr, n.d.) by using an appropriate Grammar file.0

```
volQQQ_1_1 = ($underlying == "QQQ" && ($tickerSymbol == "OTC123" || $volatility ==
0.36)) ? com.q.util.VolKeygenUtil.getFlatVolKey("QQQ", 20201202, .SPX)

volSPX_1_2 = ($tickerSymbol == "QQQ") ?
com.q.util.VolKeygenUtil.getFlatVolKey("QQQ", 20201202, .SPX)

volAAPL_2_1 = ($tickerSymbol == "AAPL") ?
com.q.util.VolKeygenUtil.getFlatVolKey("QQQ", 20201202, .SPX)
```

Figure 2-14 Sample Rule file

For the following instrument, the underlying is "QQQ" and the associated key should be "FlatVol|QQQ|20201202|.SPX"as per the second condition

The same is asserted in the Junit Test Case.

```
private VanillaOption getVanillaOption(){
        return new VanillaOption(123, "VanillaOptionQQQ",
                34.6, "QQQ",
                0.3, 0.4, 0.36,
                System.currentTimeMillis(),
System.currentTimeMillis(), DayCount.ACTUAL_365_FIXED,
                OptionType.CALL, ExcerciseType.AMERICAN, 1,
        "OTC123", System.currentTimeMillis());
    }
```

Figure 2-15 Underlying instrument for the test case

For the sake of completeness, it would be worthwhile to mention that we are using the *antlr* framework (antlr, n.d.) along with an associated grammar file to parse the rule file.

Some of the generated classes by antlr

```
1. QBaseListener
2. QBaseVisitor
3. QLexer
4. QListener
5. QParser
6. QVisitor
7. RulePredicate
```

Figure 2-16 Generated rule files using antlr framework.

```
@Test
void contextLoads() {
        EntityRuleSet<VanillaOption, BlackVarianceVolatilitySurface> entityRuleSet = new
EntityRuleSet<VanillaOption, BlackVarianceVolatilitySurface>() {
            private final Set<EntityRule<VanillaOption, BlackVarianceVolatilitySurface>> set
= new HashSet<>();
            @Override
            public int ruleSetId() {
                return 1;
            }

            @Override
            public void addEntityRule(EntityRule<VanillaOption,
BlackVarianceVolatilitySurface> entityRule) {
                rootNode.getListener().onRule(entityRule);
                set.add(entityRule);
            }

            @Override
            public Collection<EntityRule<VanillaOption, BlackVarianceVolatilitySurface>>
get() {
                return set;
            }
        };
        RuleVisitor<VanillaOption, BlackVarianceVolatilitySurface> ruleVisitor = new
RuleVisitor<>(entityRuleSet, VanillaOption.class, BlackVarianceVolatilitySurface.class);
        VanillaOption vanillaOption = getVanillaOption();
        TypeNode<VanillaOption, BlackVarianceVolatilitySurface> left =
rootNode.getTypeNodeTypeNodePair().getLeft();
        TypeNode<VanillaOption, BlackVarianceVolatilitySurface> right =
rootNode.getTypeNodeTypeNodePair().getRight();
        JoinNode<VanillaOption, BlackVarianceVolatilitySurface> joinNodeInstrument = new
JoinNode<>(rootNode, left, rootNode.getLevel());
        TerminalNode<VanillaOption, BlackVarianceVolatilitySurface> terminalNode = new
TerminalNode<VanillaOption, BlackVarianceVolatilitySurface>(right);

        rootNode.exec();
        left.exec();
        right.exec();

        ruleRepository.getRuleSet("RULESET1").forEach(s -> {
            com.q.rules.QLexer lexer = new com.q.rules.QLexer(new ANTLRInputStream(s));
            com.q.rules.QParser parser = new com.q.rules.QParser(new
CommonTokenStream(lexer));
            ParseTree tree = parser.parse();
            ruleVisitor.visit(tree);

        });
        Pair<VanillaOption, Integer> retVal = joinNodeInstrument.apply(vanillaOption);
        String key = terminalNode.apply(Pair.of(Pair.of(retVal.getLeft(), getVolSurface()),
retVal.getRight()));
        Assertions.assertEquals("FlatVol|QQQ|20201202|.SPX", key);
        Assertions.assertEquals(3, entityRuleSet.get().size());
    }
```

Figure 2-17 Rule test case

2.5 Conclusion

In this chapter we touched upon the details of the business objects relations and how they fit into the overall valuation framework. We also investigated the concept of differential updates that improves the performance both in terms of space and processing time.

Finally, we went through an implementation of the rule engine with respect to Rete's algorithm.

Before we conclude we would like to revisit the high-level package structure of the project –

Primarily it consists of the following high-level modules -

1. Common - *Plain maven project*
2. Marketdata - *Spring Boot application*
3. Snapshot-orchestrator - *Spring Boot application*
4. Engine - *Spring Boot application*

This contains all the necessary persistent entity, messaging data transfer objects and serializers.

Chapter 1 mostly dealt with some part of *Common* module .

However, this module also contains all the necessary persistent entity, messaging data transfer objects and serializers.

We will be mostly using Kafka (Kafka, n.d.)and Redis (Redis, n.d.) as the messaging and persistent cache frameworks respectively so relevant serializers are implemented accordingly.

In Chapter 3 , will cover the details on Marketdata and Snapshot Orchestrator.

3. Marketdata and Snapshot Orchestrator

3.1 Marketdata

The market data component has implementations for consuming the spot, vol surface and other updates.

e.g., SpotDTO is the data transfer object that arrives over the wire using Kafka layer (Kafka, n.d.) and subsequently its converted into SpotPrice entity and persisted in Redis. (Redis, n.d.)

We are using Kryo (Kryo, n.d.) to serialize/deserialize objects as its efficient and we can control the process at field level.

Each persistent entity implements the interface – NamedTimedEntity. The fundamental design requirement dictates that we should assign a ***timestamp*** and a ***version*** the moment an entity is persisted in the system.

```
public interface NamedTimedEntity
{
    long getSnapshotTime();
    String getName();
    int getVersion();
}
```

Figure 3-1 NamedTimedEntity interface

We would be using the version field to query the Redis cache whenever we want to retrieve a spot or vol surface. This way we can have finer control over the version to use in our calculation so that we can tie/relate all the necessary market data needed accordingly.

The following is the CacheKey class which is being used to serve this purpose.

```
public class CacheKey {
   private int version;
   private String name;
   public CacheKey() {}  // default constructor needed by Kryo Serialization
   public CacheKey(int version, String name) {
      this.version = version;
      this.name = name;
   }
   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      CacheKey cacheKey = (CacheKey) o;
      return version == cacheKey.version && Objects.equals(name, cacheKey.name);
   }
   @Override
   public int hashCode() {
      return Objects.hash(version, name);}
```

Figure 3-2 Cache Key class

The following shows the code snippet which is used to persist and query the entities in Redis hash.

```
@ComponentScan(basePackages = "com.q")
@ConfigurationPropertiesScan("com.q")
@EnableJpaRepositories(basePackages = "com.q")
@Repository
public class SpotPriceDAO implements MarketDataDAO<CacheKey, SpotPrice>{

    private final RedisTemplate<String, SpotPrice> redisTemplate;
    private final MarkerMessageProducer markerMessageProducer;
    private static final Logger LOGGER =
LoggerFactory.getLogger(SpotPriceDAO.class);
    public SpotPriceDAO(@Autowired RedisTemplate<String, SpotPrice>
redisTemplate,
                        @Autowired MarkerMessageProducer markerMessageProducer) {
        this.redisTemplate = redisTemplate;
        this.markerMessageProducer = markerMessageProducer;
    }

    @Override
    public void save(CacheKey key, SpotPrice value) {
        HashOperations<String, CacheKey, SpotPrice> hashOps =
this.redisTemplate.opsForHash();
        hashOps.put("SPOTS", key, value);
        LOGGER.info("Saved Spot {} with key {} ", value, key);
        this.markerMessageProducer.sendMarker(value.getSnapshotTime(),
value.getName(), value.getVersion(), true);

    }
    @Override
    public SpotPrice get(CacheKey key) {
        return this.redisTemplate.opsForValue().get(key);
    }
}
```

Figure 3-3 Sample Implementation of SpotPriceDAO

Typically, intraday updates contain a lot of spot and other market data changes. So, it is beneficial if we process them separately across Kafka partitions and store with corresponding versions stamped. Although not shown here, we can implement custom logic in these partition listeners.

```
@KafkaListener(topicPartitions = @TopicPartition(topic = "${spot.topic.name}" ,
partitions = { "0"} ), containerFactory = "spotPriceKafkaListenerContainerFactory")
public void spotPriceListenerPartition0(SpotPriceDTO spotPriceDTO,
@Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
    logger.info("Received spot message: {} from partition {} ", spotPriceDTO,
partition);
saveSpotPrice(spotPriceDTO);
}
@KafkaListener(topicPartitions = @TopicPartition(topic = "${spot.topic.name}" ,
partitions = { "1"} ), containerFactory = "spotPriceKafkaListenerContainerFactory")
    public void spotPriceListenerPartition1(SpotPriceDTO spotPriceDTO,
@Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
        logger.info("Received spot message: {} from partition {} ", spotPriceDTO,
partition);
        saveSpotPrice(spotPriceDTO);}
```

Figure 3-4 Partitioned Kafka Listeners

We have a provision to register Redis listeners whenever an entry is inserted into the hash as shown below. Necessary action can be taken accordingly. But for our use case we are going to send a marker message (Chandy & Lamport, 1985) as shown above.

```
@Bean
    MessageListenerAdapter messageListener() {
        return new MessageListenerAdapter((MessageListener) (message, bytes)
-> {
            //in redis-cli : config set notify-keyspace-events KEA
          System.out.println("Got message for cache persistence >
"+message);
        });
    }
```

Figure 3-5 Sample Redis Listener

3.2 Design of Snapshot Generation

Each market data subscriber is designated as an independent process in a separate container as shown below.

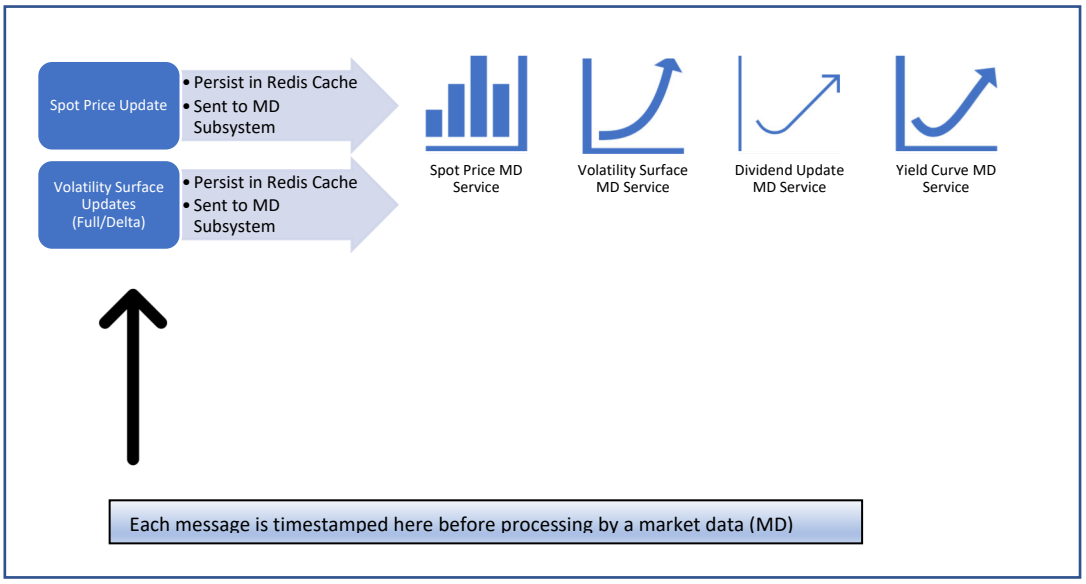

Figure 3-6 Snapshot Generation

Some of the service containers are configured to release a new marker (Chandy & Lamport, 1985) message which is sent to the **Snapshot Orchestrator** based on certain conditions e.g., on receiving a new spot price update.

3.2.1 Snapshot Orchestrator

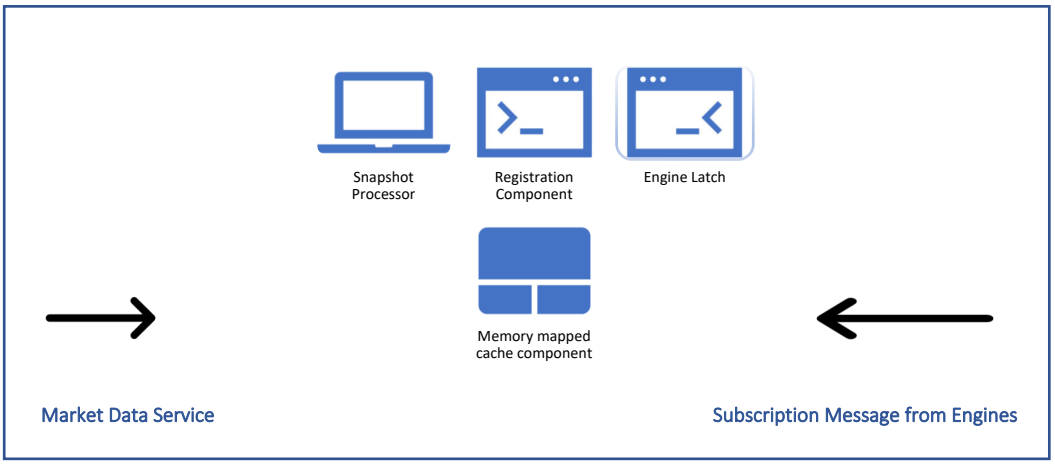

Figure 3-7 Snapshot Orchestrator

3.2.1.1 Snapshot Processor

- The snapshot processor consumes all the market data updates and groups them by symbol.
- It maintains a map of the version and time when it got the updates for each entity.
- Each symbol can have multiple types of spot updates as well as multiple types of parameter updates. E.g., volatility surface can be flat, black variance, etc. (Ballabio, QuantLib, n.d.)
- So, it ties these updates as part of a single update whenever it gets a marker message (Chandy & Lamport, 1985)to create a new snapshot for a symbol.
- For processing the marker, it refers the time-version map to include the versions which are earlier than the incoming update timestamp.
- Once it creates the following structure it removes all the entries in the map which are older than the marker time
- Updates can be also stored based on a window mode and pick the update based on some function over those windows.
- Here the index of each element is fixed and configured at startup.

SYMBOL(sample)

0	Spot Updates	Index 1 for Live Data, Index 2 for End of Day etc.
1	Volatility Surface Updates	Index 1 for Flat Vol, Index 2 for Black Variance etc.
2	Yield Curve Updates	Index 1 for Flat Forward, Index 2 for Curve etc.

- These updates are stored in the memory mapped cache that serves as a persistent store.

3.2.1.2 Memory Mapped Cache

- Implemented using memory mapped file.
- It is divided into pages of configurable size.
- These updates are divided into shards – an ordered collection of symbol updates.
- Each shard has a specific region in the memory map and can be referred by a pointer to the memory region.
- A shard becomes ready for processing by the engine whenever it either reaches a particular size or a signal is generated to release the shard for processing.
- Once it is released for processing it is not mutated.
- Memory buffer from the snapshot processor is flushed to the shard once it becomes available for processing.

3.2.1.3 Registration Component

- All calculation engines will send a Message of Interest to the snapshot orchestrator on startup. E.g., Engine 1 processing AAPL, GOOGL, NFLX and FB options will send the message in the following format (e.g., proto format)

  Message of Interest

Engine Name
Host Name
Process ID
Symbols of interest for spots
Vol Surface keys of interest
Yield curve keys of interest
Dividend curve keys of interest

- On receiving a message of interest this will be stored in a registry.
- This component will send the relevant shard pointer to the engine.

3.2.1.4 Engine Latch

- This component sends an update to all the registered engines for a particular shard.
- It keeps on listening to successful processing from each engine and once it receives updates from all the engines it sends an update to the memory mapped cache component to mark the shard as free for update.

3.3 Markers

Markers are special local updates/signals that are sent from Market Data processes to the snapshot orchestrator (Chandy & Lamport, 1985). The orchestrator holds them up in buckets against each tag. E.g., we can send a marker when we consume the spot update – Spot|AAPL or vol update - Vol|AAPL|BLACKVOL_TEST for the same tag -AAPL.

A bucket close signal is sent to the orchestrator to stop accepting further updates in the bucket for the current tag and whenever the process is ready, we can retrigger a risk calculation for all the instruments depending on this tag.

E.g., *if we plan to send a bucket close marker message when we get the update for Spot|AAPL we are in effect requesting a recalculation of risk numbers for all the instruments that are dependent on AAPL by virtue of volatility or dividend updates*.

However, it should be noted that it is not necessary to send a close signal for every parameter update.

E.g., *for dividend related market data updates for AAPL we may not trigger any recalculation of risk.*

In *Chandy Lamport algorithm* (Chandy & Lamport, 1985) paradigm once the close bucket signal is initiated and a recalculation trigger is raised the orchestrator processes all the buckets for the tag and triggers the calculation.

```
public class MarkerAndAddressReservationMessage {
   private long snapshotTime;
   private String id;
   private int version;
   private boolean closeBucket;
   private boolean reserveAddress;
   private boolean freeAddress;
   private long addressLoc;
.....
}
```

Figure 3-8 MarkerAndAddressReservationMessage

3.4 Snapshot Orchestrator

The orchestrator constitutes of the following components –

1. Buckets
2. Watermark
3. Engine Latch – Free Address Message
4. Memory map manager

3.4.1 Buckets

In terms of the implementation bucket is implemented by the CompositeSnapshotWindow. As mentioned above all local updates are continuously added in the windowed snapshot for the corresponding tags.

A sample implementation of this class is shown below –

```
public class CompositeSnapshotWindow {
    private long snaptime;
    private final Cache<String, Integer> spotsnap;
        <<other market data caches>>
    private final Set<String> spotKeys;
       <<spots keys applicable for next cycle>>
    public CompositeSnapshotWindow() {
        int concurrency = Runtime.getRuntime().availableProcessors();
        this.spotsnap =
CacheBuilder.newBuilder().concurrencyLevel(concurrency).build();
        ……
        this.spotKeys = Sets.newConcurrentHashSet();
        ……
        this.yieldCurveKeys = Sets.newConcurrentHashSet();}
    public void setSpotsnap(String key, int version){
            this.spotKeys.add(key);
          << just serves as the dirty set for the next cycle >>
            this.spotsnap.put(key, version);
}
……

    public void closeWindow(){
        << all done for this window >>
        clearCurrentKeys();
    }
    private void clearCurrentKeys() {
        this.spotKeys.clear();
        this.volKeys.clear();
        this.yieldCurveKeys.clear();}
    ……
```

Figure 3-9 CompositeSnapshotWindow

The corresponding setters are invoked whenever we want to store the local updates. And once the close signal arrives the keys are cleared. The underlying storage of the maps are implemented using Guava cache (Guava, n.d.). The keys that have changed in the current window are tracked using the sets e.g., spotKeys, volsKeys and yieldCurveKeys.

Window Manager is responsible for controlling various states of the Window.

Normal updates are stored in the in-memory maps as shown above. However, when the bucket is closed these updates are written to the memory mapped buffer (explained below).

Care must be taken when these updates are flushed to the memory map. Because these locations may be in-use by the engines that are performing the calculation.

So, first it checked whether an address is free to be written. If it is not, when the next cycle arrives regardless of the address being used or not, the change is flushed to the address. This behavior can be altered depending on the functional requirements.

Window Manager does not send the updates downstream as soon as it closes the bucket. It again depends on the watermark.

Following code snippets demonstrate the implementation–

```
public void closeBucket(String tag) {

processPreviousSnapshot(Objects.requireNonNull(this.prevSnapshot.getIfPresent(ta
g)));
   CompositeSnapshotWindow compositeSnapshotWindow =taggedBucket.get(tag);
        Set<String> rootAddressesFree = Sets.newHashSet();
        compositeSnapshotWindow.getSpotKeys().forEach(s -> {
            memoryIndexRepository.getMemoryAddresses(s).forEach(address -> {
                Integer value = compositeSnapshotWindow.getSpotsnap().get(s);
                processEntry(tag, rootAddressesFree, address, value);
            });
        });
……..

private void processEntry(String tag, Set<String> rootAddressesFree, String
address, Integer value) {
        String cacheId = address.split(":")[0];
        int index = Integer.parseInt(address.split(":")[1]);
        if(memoryMapManager.isFree(cacheId, index)){
            memoryMapManager.putInBuffer(cacheId, index, value);
            LOGGER.info("*****------- DONE storing CacheId {},index {},value{} -
---------******* ", cacheId, index, value);
            rootAddressesFree.add(address);
            SnapshotAllocationMessage snapshotAllocationMessage =

this.snapshotAllocationMessageMapTemp.computeIfAbsent(address, s -> new
SnapshotAllocationMessage());
            if(snapshotAllocationMessage.getMappedFile() == null){
                snapshotAllocationMessage.setCacheId(cacheId);

snapshotAllocationMessage.setMappedFile(memoryMapManager.getMappedFileName(cache
Id));
                snapshotAllocationMessage.setUpdate(true);
            }
            snapshotAllocationMessage.getAddressUpdates().add(index);
        }else {
            List<BackUpSnapshot> backupList = prevSnapshot.getIfPresent(tag);
            Objects.requireNonNull(backupList).add(new BackUpSnapshot(address,
value));
        }
    }
```

Figure 3-10 closeBucket and processEntry operation

It is important to understand the motivation behind the operation to close the bucket. It is to give a consistent picture of the calculation i.e., if a trader has a configuration to recalculate the risk when a spot update arrives, essentially, he wants to use the latest price to understand how it impacts the risk since the last compute cycle (*last spot, volatility update etc.*).

From *Chandy Lamport* (Chandy & Lamport, 1985) algorithm standpoint, this is when a global snapshot is logically ready to be processed.

3.4.2 Watermark

Although marker updates to close the bucket may arrive from market data system it does not mean that we can immediately send the information to the engines once the close bucket operation executes.

E.g. the engine maybe throttled if we send too many updates in one shot.

So, we would have to design and implement a strategy to send the same. *Some implementations may consider the load on the engines or another timer thread.*

For the purpose of our discussion (to keep things simple) we are sending the watermark whenever we are getting the close bucket signal *(ALWAYS_TRUE)*

However, we have the required strategy interface to implement the functionality.

The following code snippet shows a periodic timer thread that keeps on processing the market data updates.

Here it uses the Rx Java (Nurkiewicz, Christiansen, & Meijer, 2016) Publish subject to handover the update and process it in a separate single threaded executor.

```
public interface WaterMarkEmittingStrategy {
    boolean shouldEmit();
    WaterMarkEmittingStrategy ALWAYS_TRUE = () -> true;
}
```

```
scheduledExecutorService.scheduleWithFixedDelay(() -> {
            set.clear();
            try {
                if(!this.enginesInitialized) {
                    this.enginesInitialized =
this.engineRegMessageConsumer.waitTillInitialized();
                }
                Queues.drain(blockingQueue, set,
blockingQueue.size(), 10, TimeUnit.SECONDS);
                set.forEach(publishSubject::onNext);

Arrays.stream(waterMarkEmittingStrategy).forEach(st -> {
                    if(st.shouldEmit()){
                        publishSubject.onNext(Watermark.INSTANCE);
                    }
                });

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, 1, 10, TimeUnit.SECONDS);
```

```
publishSubject.asObservable()
                .observeOn(Schedulers.from(newSingleThreadExecutor))
                .doOnNext(markerMessage ->
                {
                    LOGGER.info("Got marker message in publish
subject {} ", markerMessage);
                    if(markerMessage == Watermark.INSTANCE){
                        snapshotWindowManager.processWatermark();
                    }else {
                        if (markerMessage.isFreeAddress()) {
                            LOGGER.info("*** Got FREE ADDRESS
message {} **** ", markerMessage);

snapshotWindowManager.freeAddress(markerMessage.getId(), (int)
markerMessage.getAddressLoc());
                        } else {
                            String tag =
markerMessage.getId().split("\\|")[1];
                            SnapshotWindowManager.ParameterType
parameterType =

SnapshotWindowManager.ParameterType.valueOf(markerMessage.getId().sp
lit("\\|")[0]);

snapshotWindowManager.addToBucket(parameterType, tag,
markerMessage.getId(), markerMessage.getVersion());
                            if (markerMessage.isCloseBucket()) {

snapshotWindowManager.closeBucket(tag);
                            }
                        }
                    }
                }).doOnError(throwable -> LOGGER.info("Error {} ",
throwable.getMessage())).subscribe();
```

Figure 3-11 Watermark processing over Rx Java subject

3.4.3 Engine Latch – *Free Address* Message

As a sidenote, we should also have a facility to process the messages from the engines which relay the information that they are done with the calculation with the last set of data. So, we can now send the next set when the bucket close signal arrives, and watermark is ready.

This is done using a Boolean field that is set on the incoming message.

The orchestrator keeps a track of the number of address updates it sent per tag per engine and then it decrements the counter whenever it gets the free address message. Once the counter reaches 0, orchestrator is free to send further updates.

```
public void freeAddress(String cacheId, int addressLoc) {
        requestSentToEngines.computeIfPresent(cacheId, (s, integer)
-> integer -1);
        LOGGER.info("  %%%% After free address for {} , size is {}
%%%% ", cacheId, requestSentToEngines.get(cacheId));
        if(requestSentToEngines.get(cacheId) <= 0){ // receive more
than one confirmation message due to any reason
            requestSentToEngines.remove(cacheId);
            memoryMapManager.markFreeInBuffer(cacheId, addressLoc);
        }
    }
```

Figure 3-12 Free Address implementation

3.4.4 Memory map manager

Memory map manager holds the underlying persistent cache using memory mapped file.

```
public interface MemoryMapManager {
   void set(long memoryAddress, int version);
   void putInBuffer(String cacheId, int index, int version);
   String getMappedFileName(String cacheId);
   void markUsedInBuffer(String cacheId, int index);
   void markFreeInBuffer(String cacheId, int index);
   boolean isFree(String cacheId, int idx);
   Long reserveMemory(String cacheId, int spotsCount, int volsCount, int yieldCurveCount);
   int getValue(long address);
   long getStartAddress(long memoryAddress);
   void markUsed(long memoryAddress);
   void markFree(long memoryAddress);
   boolean isFree(long memoryAddress);
   Set<Long> getUsedAddresses();
   Set<Long> getAddressSet();
}
```

Figure 3-13 MemoryMapManager interface

A calculation engine instance sends the request specifying the list of spots, vols or dividend updates which it needs for risk calculation. (Ballabio, QuantLib, n.d.)

Based on this a specific memory mapped region is created –

```
public Long reserveMemory(String cacheId, int spotsCount, int volsCount,
int yieldCurveCount) {
        try {
            double sz = 4 * (spotsCount + volsCount + yieldCurveCount);
            Path path = Paths.get(System.getProperty("user.home")+
File.separator+cacheId+"MappedCache");
            if (!Files.exists(path)) {
                Files.createDirectory(path);
            }
            final String fileName = path.toAbsolutePath().toString()+
File.separator +"version_cache_" + cacheId + FILE_SUFFIX;
            this.cacheFileNameMap.put(cacheId, fileName);
            RandomAccessFile cacheFile = new RandomAccessFile(fileName,
"rw");
            MappedByteBuffer byteBuffer =
cacheFile.getChannel().map(FileChannel.MapMode.READ_WRITE, 0, (long) sz);
            this.mbfMap.put(cacheId, byteBuffer);
            this.usedLocations.put(cacheId, new HashSet<>());
            long rootAddress = ((DirectBuffer) byteBuffer).address();
            for (int i = 0; i < sz; i+=4) {
                long key = rootAddress + i;
                LOGGER.info("Root address set key {}, value {}", key,
rootAddress);
                this.rootAddresses.put(key, rootAddress);
                this.usedAddressMap.put(key, false);
            }
            return (long) sz;

        } catch (Exception e) {
            e.printStackTrace();
        }
        throw new RuntimeException("This should not have occured for
"+cacheId);
    }
```

Figure 3-14 Reserve memory implementation

Implementation-wise we have a single threaded access to the memory map while writing. Operations are short and quick using the Single Writer Principle (Single Writer Principle, n.d.).

At the engine end, the same file is mapped for reading from the desired locations. But as the locations are marked as used until the address free message arrives this does not cause concurrency issues.

A small detail regarding the reservation of memory – the engine sends a list of required tagged parameters (E.g., spot of AAPL, vols of NFLX) and these are allocated and mapped in sequential manner.

However multiple engines may need the same data but via different mapped buffers.

This information is stored in Redis map. When updates need to be sent it queries this map.

All this is maintained in MemoryIndexRepository.

```
@Repository
@ComponentScan(basePackages = "com.q")
public class MemoryIndexRepositoryImpl implements MemoryIndexRepository{

   private final RedisTemplate<String, String> template;
   public MemoryIndexRepositoryImpl(@Autowired RedisTemplate<String, String> template) {
      this.template = template;
   }

   @Override
   public void mapMemoryAddress(String paramKey, String memoryAddress) {
      this.template.opsForList().leftPush(paramKey, memoryAddress);
   }

   @Override
   public void unmapMemoryAddress(String paramKey) {
      this.template.delete(paramKey);
   }

   @Override
   public List<String> getMemoryAddresses(String paramKey) {
      return new ArrayList<>(Objects.requireNonNull(this.template.opsForList().
            range(paramKey, 0, Integer.MAX_VALUE)));
   }
}
```

Figure 3-15 MemoryIndexRepository implementation

3.5 Conclusion

In this discussion we covered the concept of windowing/bucketing, snapshot creation and watermarking .

We also explored the intricacies of using memory map and the concept of single writer principle (Single Writer Principle, n.d.) to effectively manage the valuation execution.

The framework used Kafka partitions (Kafka, n.d.) and Redis hashes (Redis, n.d.) to effectively handle the market data updates and relaying the global state of the processes.

Last but not the least, we are using the SpringBoot (Spring Framework, n.d.) container behind the scenes and hence it effectively handles the containerization and other micro service-based features automatically, including the forward-looking roadmap to migrate all these services to cloud.

4. Engine

The valuation engine is a Spring Boot application that is responsible for executing the actual calculation.

We will now discuss the lifecycle phases.

4.1 Creating the actual valuation process

As an example, if we want to create a BlackScholesMertonProcess (Ballabio, QuantLib, n.d.) we would need a set of certain kind of volatility surface, dividend curves and quotes (spots). Along with the parameters it also needs contract data information.

The actual keys that are mapped to the instrument are stored in Redis (Rule Repository) (Redis, n.d.)as part of *spots_rule, vols_rule* etc. These are stored after processing via the rules engine as discussed in the beginning.

```
private void createValuationRequest() {
Settings.instance().setEvaluationDate(DateUtil.fromEpochMillis(instrum
ent.getSnapshotTime()));
        LOGGER.info("Setting up spot");
        String primarySpotKeyFromRule =
ruleRepo.getPrimarySpotKey(instrument.getName(), null);
        LOGGER.info("Setting up vol");
        String volKeyFromRule =
ruleRepo.getVolSurfaceKey(instrument.getName(), null);

this.engineRegistrationMessage.setEngineSequence(engineConfig.getEngin
eId());

this.engineRegistrationMessage.setSpotids(primarySpotKeyFromRule.split
(",")[0]);

this.engineRegistrationMessage.setVolIds(volKeyFromRule.split(",")[0])
;
       LOGGER.info("Sending registration message {} ",
engineRegistrationMessage);

engineRegistrationMessageProducer.sendEngineRegistrationMessage(engine
RegistrationMessage);
        setSpotParameters(primarySpotKeyFromRule);
        setVolParameters(volKeyFromRule);
        Date settlementDate =
DateUtil.fromEpochMillis(instrument.getSettlementDate());
        prepareBlackScholesMertonProcess(settlementDate);
        PlainVanillaPayoff payoff = new
PlainVanillaPayoff(OptionType.values()[instrument.getOptionType()].typ
e,
                instrument.getStrike());
        Exercise americanExercise = new
AmericanExercise(settlementDate,
                DateUtil.fromEpochMillis(instrument.getMaturity()));
        this.americanOption =
                new VanillaOption(payoff, americanExercise);
    `
```

Figure 4-1 Preparation of valuation process

4.2 Registering for updates

As part of the initialization process of setting up the valuation request, engines send their parameter subscriptions to the snapshot orchestrator.

The engine waits till it gets the confirmation back:

```
public void
sendEngineRegistrationMessage(EngineRegistrationMessage
engineRegistrationMessage){
        kafkaTemplate.send(engineRegistrationMessageTopic,
engineRegistrationMessage);
        this.latch.put(engineRegistrationMessage.getId(), new
CountDownLatch(1));
        LOGGER.info("----- Waiting to hear back from orchestrator
for address confirmation for id {} ----- ",
                engineRegistrationMessage.getId());
        try {

this.latch.get(engineRegistrationMessage.getId()).await();
        } catch (InterruptedException e) {
            e.printStackTrace();}}
```

Figure 4-2 Waiting for confirmation from snapshot orchestrator.

Once the confirmation arrives, the valuation orchestrator sets the mapped byte buffer related mappings as shown below (in ValuationParameterRepository)

```
public void processSnapshotAllocationMessage(SnapshotAllocationMessage message)
{
        if(message.isUpdate()){
            processUpdates(message);
        }else {
            LOGGER.info("Processing Snapshot Allocation Message {} ", message);
            int startMemAddress = 0;
            String[] spotIds = StringUtils.isNotEmpty(message.getSpotIds()) ?
message.getSpotIds().split(",") : null;
            String[] volIds = StringUtils.isNotEmpty(message.getVolIds()) ?
message.getVolIds().split(",") : null;
            String[] yieldCurveIds =
StringUtils.isNotEmpty(message.getYieldCurveIds()) ?
message.getYieldCurveIds().split(",") : null;
            repository.init(message.getMappedFile(), message.getSz(),
message.getCacheId());
            int count = 0;
            if (spotIds != null) {
                for (String spotId : spotIds) {
                    int key = startMemAddress + count;
                    repository.setSpotAddress(message.getCacheId(), spotId,
key);
                    repository.setParameterType(message.getCacheId(), key,
ValuationParameterRepository.ParameterType.SPOT);
                    count += 4;
                    LOGGER.info("Address set for spotId {} at address {} ",
spotId, key);
                }}
```

Figure 4-3 processSnapshotAllocationMessage method

Once the confirmation arrives, the valuation orchestrator sets the mapped byte buffer related mappings as shown below (in ValuationParameterRepository)

```
@Repository
public class ValuationParameterRepository {
    private static Unsafe unsafe;
    private static final ByteOrder byteOrder;
    private final Map<String, MappedByteBuffer> mbfMap;
    public void init(String mappedFile, long sz, String cacheId) {
        RandomAccessFile cacheFile;
        try {
            cacheFile = new RandomAccessFile(mappedFile, "rw");
            MappedByteBuffer byteBuffer =
cacheFile.getChannel().map(FileChannel.MapMode.READ_WRITE, 0, sz);
            this.mbfMap.put(cacheId, byteBuffer);

        } catch (IOException e) {
            e.printStackTrace();
        }

    }
……
public ValuationParameterRepository(@Autowired RedisTemplate<String,
SpotPrice> spotPriceRedisTemplate, @Autowired RedisTemplate<String,
TimedBlackVarianceVolatility> blackVarianceVolatilityRedisTemplate) {
        this.spotPriceRedisTemplate = spotPriceRedisTemplate;
        this.blackVarianceVolatilityRedisTemplate =
blackVarianceVolatilityRedisTemplate;
        this.spotCache = Maps.newHashMap();
…….
    }

    public void setSpotAddress(String cacheId, String key, long
address){
        this.spotCache.computeIfAbsent(cacheId, s -> new
HashMap<>()).put(key, (int) address);
    }
public Optional<SpotPrice> getSpot(String cacheId, String key, String
version){
        if(version != null){
            CacheKey cacheKey = cacheKeyThreadLocal.get();
            cacheKey.setName(key);
            cacheKey.setVersion(Integer.parseInt(version));
            SpotPrice spot = (SpotPrice)
spotPriceRedisTemplate.opsForHash().get("SPOTS", cacheKey);
            return Optional.of(Objects.requireNonNull(spot));
        }
        Integer index;
        if((index = spotCache.get(cacheId).get(key)) != null) {
            CacheKey cacheKey = cacheKeyThreadLocal.get();
            cacheKey.setName(key);
            int ver = mbfMap.get(cacheId).getInt(index);
            cacheKey.setVersion(ver);
            SpotPrice spot = (SpotPrice)
spotPriceRedisTemplate.opsForHash().get("SPOTS", cacheKey);
            LOGGER.info("**********    Extracting value from index {},
spot {} , ver {}************** ", index , spot, ver);
            return Optional.of(Objects.requireNonNull(spot));
        }
        return Optional.empty();
    }
```

Figure 4-4 Valuation Parameter Repository

4.3 Valuation Orchestrator Setup

Valuation orchestrator is responsible for maintaining the overall state with respect to registration of the engine and its updates. It also initializes the parameter repository which the engine is going to use as part of the valuation execution.

For triggering the calculation when updates are received it uses the Rx Java (Samoylov & Nield, 2020) (Nurkiewicz, Christiansen, & Meijer, 2016) subjects as shown below

```
public ValuationOrchestrator(@Autowired ValuationParameterRepository repository,
                             @Autowired EngineRegistrationMessageProducer
engineRegistrationMessageProducer) {
        this.repository = repository;
        this.spotUpdateSubject = PublishSubject.create();
        this.volUpdateSubject = PublishSubject.create();
        ……

        this.volValuatorMap = Maps.newHashMap();
        this.yieldValuatorMap = Maps.newHashMap();
        this.engineRegistrationProducer = engineRegistrationMessageProducer;

        this.spotUpdateSubject
                .observeOn(Schedulers.from(Executors.newSingleThreadExecutor()))
                .doOnNext(triggerRecalcOnSpotUpdate())
                .subscribe();
        this.volUpdateSubject
                .observeOn(Schedulers.from(Executors.newSingleThreadExecutor()))
                .doOnNext(triggerRecalcOnVolUpdate())
                .subscribe();
    }
……

private Action1<? super Pair<String, Integer>> getAction1(Map<Pair<String,
Integer>, Set<String>> valuatorMap, ValuationParameterRepository.ParameterType
parameterType) {
        return (Action1<Pair<String, Integer>>) add -> valuatorMap.get(add)
                .stream()
                .map(valuatorId -> CompletableFuture.supplyAsync(() -> {
                    ValuationExecutor valuationExecutor =
valuatorRegistry.get(valuatorId);
                    switch (parameterType) {
                        case SPOT:
                            LOGGER.info("Triggering recalculation for spot updates
for address {} ", add);
                            valuationExecutor.modifyValuatorSpot(add.getValue());
                            break;
       ……..……..

                    }
                    try {
                        return valuationExecutor.call();
                    } ……
                    return ErroredValuationResponse.INSTANCE;
                })).collect(Collectors.toList())
                .stream()
                .map(CompletableFuture::join)
                .collect(Collectors.toList()).forEach(this::sendResponse);
    }
```

Figure 4-5 ValuationOrchestrator- initialization of subjects

It also needs to map each valuator to the set of parameters which it would be using. This way the ValuationOrchestrator can trigger the recalculation.

```
public void registerValuator(ValuationExecutor valuationExecutor) {
        LOGGER.info("Registering valuator with id {}",
valuationExecutor.getId());
        this.valuatorRegistry.put(valuationExecutor.getId(),
valuationExecutor);
        valuationExecutor.spotInterests().forEach(spotAddress ->

this.spotValuatorMap.computeIfAbsent(Pair.of(valuationExecutor.getCurEngine
Id(),
                        spotAddress),aLong ->
Sets.newHashSet()).add(valuationExecutor.getId()));
        valuationExecutor.volInterests().forEach(volAddress ->

this.volValuatorMap.computeIfAbsent(Pair.of(valuationExecutor.getCurEngineI
d(), volAddress),
                        aLong ->
Sets.newHashSet()).add(valuationExecutor.getId()));
        valuationExecutor.yieldCurveInterests().forEach(yieldCurveAddress -
>

this.yieldValuatorMap.computeIfAbsent(Pair.of(valuationExecutor.getCurEngin
eId(), yieldCurveAddress),
                        aLong ->
Sets.newHashSet()).add(valuationExecutor.getId()));
        LOGGER.info("Registration complete for valuator with id {}",
valuationExecutor.getId());
    }
```

Figure 4-6 ValuationOrchestrator- registration of valuator

We also need to define the action that would be triggered when the updates arrive for each valuator. The same has been shown above in Figure 4.5.

4.4 Valuation Executor

The actual code that would be executed on market data updates is part of the implementation of the following interface.

```
public abstract class ValuationExecutor implements Callable<ValuationResponse> {
   abstract void modifyValuatorSpot(int idx);
   abstract void modifyValuatorVol(int idx);
   abstract void modifyValuatorYieldCurve(int idx);
   abstract void setInstrument(NamedTimedEntity namedTimedEntity);
   abstract Set<Integer> spotInterests();
   abstract Set<Integer> volInterests();
   abstract Set<Integer> yieldCurveInterests();
   abstract String getId();
   abstract String getCurEngineId();
}
```

Figure 4-7 ValuationExecutor interface

The indexes of the mapped buffer are mapped to the functions that would be invoked as part of the specific updates. Following depicts the same -

```
private void setSpotParameters(String primarySpotKeyFromRule) {
        LOGGER.info("Primary Spot Key {}", primarySpotKeyFromRule);
        String primarySpotKey = primarySpotKeyFromRule.split(",")[0];

        String version = primarySpotKeyFromRule.split(",")[1];
        Optional<SpotPrice> spotPrice = repository.getSpot(curEngineId,
primarySpotKey, version);
        Optional<Integer> spotAddress = repository.getSpotAddress(curEngineId,
primarySpotKey);
        if(spotPrice.isPresent() && spotAddress.isPresent()){
            LOGGER.info("Spot Instance {} and address {}", spotPrice.get(),
spotAddress.get());

blackScholesMertonProcessBuilder.withUnderlyingPrice(spotPrice.get().getMid());
            spotTransformMap.put(spotAddress.get(),
blackScholesMertonProcessBuilder -> {
                Optional<SpotPrice> spot = repository.getSpot(curEngineId,
primarySpotKey, null);
                spot.ifPresent(price ->
blackScholesMertonProcessBuilder.withUnderlyingPrice(price.getMid()));
                return null;
            });
        }
    }
……..
@Override
    public void modifyValuatorSpot(int idx) {
        spotTransformMap.get(idx).apply(blackScholesMertonProcessBuilder);
        engineRegistrationMessageProducer.sendFreeAddressMarker(curEngineId,
engineConfig.getMarkerTopic(), idx);
    }
```

Figure 4-8 ValuationExecutor interface implementation

Finally, the call method defined below outputs the valuation response.

```
@Override
    public ValuationResponse call() throws Exception {
        BinomialCRRVanillaEngineBuilder builder = new
BinomialCRRVanillaEngineBuilder();

builder.withProcess(blackScholesMertonProcessBuilder.build()).withSteps(engineC
onfig.getCrrSteps());
        americanOption.setPricingEngine(builder.build());
        valuationResponse.setNpv(americanOption.NPV());
        valuationResponse.setDelta(americanOption.delta());
        valuationResponse.setGamma(americanOption.gamma());
        LOGGER.info("Initial Valuation Result : {}", valuationResponse);
        return valuationResponse;
    }
```

Figure 4-9 ValuationExecutor call()

4.5 Valuation Proxy

The proxy intercepts the first call invocation and in turn invokes the ValuationOrchestrator `registerValuator` method shown above.

```
public class ValuatorProxy implements InvocationHandler {
    private final ValuationExecutor valuationExecutor;
    private final ValuationOrchestrator valuationOrchestrator;
    private static final Logger LOGGER=LoggerFactory.getLogger(ValuatorProxy.class);

    public ValuatorProxy(ValuationExecutor valuationExecutor, ValuationOrchestrator
valuationOrchestrator) {
        this.valuationExecutor = valuationExecutor;
        this.valuationOrchestrator = valuationOrchestrator;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        valuationOrchestrator.registerValuator(valuationExecutor);
        long start = System.nanoTime();
        Object res = method.invoke(valuationExecutor);
        LOGGER.info("Total Valuation Time = {} secs", (System.nanoTime() -
start)/Math.pow(10, 9));
        return res;
    }
}
```

Figure 4-10 Valuator Proxy

4.6 Sample Execution Output

We demonstrate here some sample output of the flow.

Initial Spot Price version: 21, mid-price = 1653.5, npv= 99.10194

Figure 4-11 Sample Valuation Output with Riskserver framework

4.7 Conclusion

In this chapter we demonstrated an actual valuation by using QuantLib (Ballabio, QuantLib, n.d.) and Riskserver that can take place in production. We have shown an implementation of how a pricing engine can register for valuation updates and create a valuation executor along the way together with an orchestrator.

We also looked in the concept of using a valuation proxy by using a standard proxy pattern (Metsker & Wake, 2006) to intercept the valuation requests.

Lastly, we saw how individual pricing processes can send confirmation that the valuations are complete, and they can proceed with the next set of updates.

5. Alternate Implementation Strategies

5.1 Resource Manager, Load Balancer and Service Discovery

As discussed till now all our services are mainly spring boot applications (Spring Boot, n.d.) and as such we do not need to implement any out of the box code for load balancing.

One can easily integrate Spring Cloud Kubernetes Load Balancer (Larsson, 2019) for this purpose.

For service discovery Netflix Eureka (Larsson, 2019) maybe used.

There are easily integrable frameworks like Spring Cloud Sleuth and Zipkin for Distributed Tracing and Grafana for Centralized Monitoring and Alarms (Larsson, 2019).

However, for managing resources we can have custom implementation for the following –

- Reserving memory map size if needed for a cluster of engines. It can just delegate the responsibilities to the MemoryMapManager for this purpose.

  Using JSON / YAML (JSON, n.d.) /(YAML, n.d.) configurations we can make this easily configurable.

- Thread pool configuration and customization can also be done for processing the valuation requests. Again, this can be wrapped around the Executor Services / Thread Pools from Core Java framework (Bloch, 2016).

  Thread pool size, mapping memory map files to thread pools etc. can also be wired using JSON/YAML configurations.

  As these requirements are trivial in nature and mostly encompass configuration-oriented needs we will not delve deeper into the same.

5.2 Streams using Redis updates : an alternative design strategy -

If we want to develop a simple streaming framework using Redis (or any other caching framework) (Redis, n.d.)as the underlying event generator, we can start by consuming the insert events for market data e.g., spot prices, volatility surface and dividends etc.

5.2.1 Windowing

We can use window strategies like Rolling window / Tumbling window which correspond to whether we want to have an overlap between the windows or not.

These windows can fire independently using a timer thread and set the relevant updates index updates in a thread-safe array e.g., AtomicLongArray (Bloch, 2016) , (AtomicLongArray, n.d.)in a ring buffer style (i.e., we wrap around the array using module operation).

For efficiency we can use the id and version of the updates as part of a long using bit shifting.

Sample Window implementation can be as follows –

```
Window(int sz, long millis, WindowType windowType) {
    spots = new AtomicLongArray(sz);
    vols = new AtomicLongArray(sz);
    yieldCurve = new AtomicLongArray(sz);
    this.windowType = windowType;
    this.spotIdx = new AtomicInteger(0);
    ……..
    this.sz = sz;
    this.millis = millis;
    upstreamSources = new ArrayList<>();
}

public void addSpot(int curKey, int curVersion){
    long timestamp = Timer.INSTANCE.getCurrentTime();
    long val = (long) curKey << 48 | (long) curVersion << 32 | timestamp;
    spots.set(curKey % sz, val );
    movePointer(spotIdx);
}
……

private void movePointer(AtomicInteger idx) {
    int curIdx = idx.get();
    switch(windowType){
        case ROLLING:
            while(!idx.compareAndSet(curIdx,
moveRollingPointerAhead(curIdx))){
                curIdx = idx.get();
            }
            break;
        case TUMBLING:
            while(!idx.compareAndSet(curIdx,
moveTumblingPointerAhead(curIdx))){
                curIdx = idx.get();
            }
            break;
    }
}
……


private int moveTumblingPointerAhead(int curIdx) {
    int left = curIdx >> 16;
    int right = curIdx & 0xFFFF;
    right = right + 1;
    if(right == sz -1 ){
        left = (left + 1) %sz;
    }
    right = right %sz;
    return left << 16 | right;
}

private int moveRollingPointerAhead(int curIdx) {
    int left = curIdx >> 16;
    int right = curIdx & 0xFFFF;
```

Figure 5-1 Sample Window Implementation

5.2.2 Stream and Sink Interfaces

For developing a streaming framework, we need a pipeline to consume incoming data and transfer the same to a sink.

A sample interface may look like as shown below. Most of the exposed methods are self-explanatory.

```
public interface IndexedMarketDataSink<T> {
    void begin();
    void accept(T t);
    MarketDataType getUnderlyingType();
    AtomicInteger getProducerIdx();
    AtomicLongArray getDataArray();
    void register(IndexedMarketDataSink<T> other);
    void end();
}
public interface MarketDataStream<T> {
    MarketDataStream<T> merge(MarketDataStream<T>... streams);
    MarketDataStream<T> withWindow(IndexedMarketDataSink<T>
window);
}
```

Figure 5-2 Sample Sink and Stream interface (skeletal representation)

```
public abstract class PeriodicStreamPipeline<T> {
    protected final PeriodicStreamPipeline<T> prev;
    public PeriodicStreamPipeline(PeriodicStreamPipeline<T> prev)
{
        this.prev = prev;
    }
    public abstract void evaluatePipeline();
    public abstract IndexedMarketDataSink<T>
sinkFrom(IndexedMarketDataSink<T> sink);

}
```

Figure 5-3 Pipeline with the capability to evaluate.

To sync up the timings we can expose a separate Timer class and get the current time as integer from the same for all the updates in a streamlined manner rather than using separate timestamps from different initial points.

```
class Timer {
    private final AtomicInteger tick;

    private Timer() {
        tick = new AtomicInteger(0);
        ScheduledExecutorService scheduledExecutorService =
Executors.newSingleThreadScheduledExecutor();

scheduledExecutorService.scheduleAtFixedRate(tick::incrementAndGet
, 1, 1, TimeUnit.MILLISECONDS);
    }

    public static final Timer INSTANCE = new Timer();

    public int getCurrentTime(){
        return tick.get();
    }}
```

Figure 5-4 Separate timer class

5.2.3 *Periodic Calculable Stream Pipeline*

This instance of pipeline merges all the upstream market data streams and add a new sink. In addition, it also registers a Redis listener for all insert and update events. A windowing method is then applied over which the evaluation happens at specific time intervals.

```
public final MarketDataStream<Long> merge(MarketDataStream<Long>... streams) {
    for (MarketDataStream<Long> stream : streams) {
        if(stream instanceof TimedCalculableStreamPipeline) {
            TimedCalculableStreamPipeline curStream =
(TimedCalculableStreamPipeline) stream;
            TimedCalculableStreamPipeline timedCalculableStreamPipeline = new
TimedCalculableStreamPipeline(
                    curStream, curStream.sourceType) {
                @Override
                public IndexedMarketDataSink<Long>
sinkFrom(IndexedMarketDataSink<Long> sink) {
                    return new VanillaIndexedMarketDataSink(32, sink,
curStream.sourceType);
                }
            };
            collection.add(timedCalculableStreamPipeline);
        }
    }
    collection.add(new TimedCalculableStreamPipeline(this, sourceType) {
        @Override
        public IndexedMarketDataSink<Long> sinkFrom(IndexedMarketDataSink<Long>
sink) {
            return new VanillaIndexedMarketDataSink(32, sink, sourceType);
        }
    });

    return this;
}
```

```
public MarketDataStream<Long> withWindow(IndexedMarketDataSink<Long> window)
{
    if(!(window instanceof Window)){
        throw new IllegalStateException("Passed window not instance of
Window");
    }
    this.window = window;
    this.collection.forEach(c -> {
        IndexedMarketDataSink<Long> longIndexedMarketDataSink =
c.sinkFrom(window);
        registerRedisListener(longIndexedMarketDataSink);
        longIndexedMarketDataSink.begin();
    });
    evaluatePipeline();
    return this;
}
```

Figure 5-5 Merging of streams and windowing methods

```
protected void registerRedisListener(IndexedMarketDataSink<Long>
longIndexedMarketDataSink){
    ........................................................................................................................
        JedisPool pool = new JedisPool(new JedisPoolConfig(), "localhost");
        Jedis jedis = pool.getResource();
        jedis.psubscribe(new JedisPubSub() {
            @Override
            public void onMessage(String channel, String message) {
                String[] components = message.split(":");
                if(MarketDataType.valueOf(components[0])
                        == longIndexedMarketDataSink.getUnderlyingType() ) {

longIndexedMarketDataSink.accept(Long.valueOf(jedis.hget(components[0],
components[1])));
                }
            }
        }, "__key*__:*");
    });

    services.add(newSingleThreadExecutor);
}
.......................................................................................................................................................................................

static class RootPipeline extends TimedCalculableStreamPipeline{

    public RootPipeline(TimedCalculableStreamPipeline prev, MarketDataType
type) {
        super(prev, type);
    }

    @Override
    public IndexedMarketDataSink<Long> sinkFrom(IndexedMarketDataSink<Long>
sink) {
        throw new UnsupportedOperationException();
    }
}
```

Figure 5-6 Redis listener registration and RootPipeline class

5.2.4 *OpenCL/GPU based valuation engine -*

We now demonstrate an implementation of a basket pricing model in an OpenCL framework using Java wrapper.
As obvious , this code can be easily invoked from the above pipeline.
Before going ahead with the actual implementation, it is worth noting the actual formulation as illustrated in the (London, 2005)

5.2.4.1 *Background on Monte Carlo pricing for Basket Option with three constituents*

A basket option is a derivative instrument where the underlying asset is a group of securities e.g., simple stocks.

For the purpose of this discussion, we assume S_1, S_2 and S_3 are the three stock components which constitute the basket.

Then it can be proved as cited in (London, 2005)

$$S_1(t) = S_1(t_0)\left(1 + m_1\Delta t + \sigma_1\varepsilon_1\sqrt{\Delta t}\right)$$
$$S_2(t) = S_2(t_0)\left(1 + m_2\Delta t + \sigma_2(\rho_{12}\varepsilon_1 + \sqrt{1-\rho_{12}^2}\,\varepsilon_2)\sqrt{\Delta t}\right)$$
$$S_3(t) = S_3(t_0)\left(1 + m_3\Delta t + \sigma_3(\rho_{13}\varepsilon_1 + (\frac{\rho_{23}-\rho_{12}\rho_{13}}{\sqrt{1-\rho_{12}^2}})\varepsilon_2 + \sqrt{1-\rho_{13}^2-(\frac{\rho_{23}-\rho_{12}\rho_{13}}{\sqrt{1-\rho_{12}^2}})\varepsilon_3}\,)\sqrt{\Delta t}\right)$$

Here, $S_1(t_0), S_2(t_0)$ and $S_3(t_0)$ are the initial stock prices for S_1, S_2 and S_3 respectively.

Moreover ρ_{12} , ρ_{23} and ρ_{13} are the correlation between the stocks 1 and 2, 2 and 3 , 1 and 3, respectively.

And σ_1 , σ_2 and σ_3 are the volatilities of stocks 1, 2 and 3, respectively.

Lastly, $\varepsilon_1, \varepsilon_2,$ and ε_3 are the three normally distributed random variables.

If M and N are the number of simulations and time steps respectively, then after N paths are completed,

The value of the option is $\max(S_1(t) + S_2(t) + S_3(t) - \text{strike}, 0)$ for a call option else $\max(\text{strike} - S_1(t) - S_2(t) - S_3(t), 0)$ for a put. Here strike denotes the strike price for the option.

If we take the rolling sum of the above expression denoted by sum then after M simulations, the basket price $= e^{(-\text{rateOfInterest}*\text{TimePeriod})*(\frac{\text{sum}}{M})}$

5.2.4.2 *OpenCL environment details*

Following code demonstrates the current environment for running the OpenCL code.

```
CLPlatform[] clPlatforms = JavaCL.listPlatforms();
        for (CLPlatform c :
                clPlatforms) {
            CLDevice[] clDevices = c.listAllDevices(true);
            for (CLDevice cl :
                    clDevices) {
                System.out.println(cl);
            }
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz
Intel(R) Iris(TM) Graphics 6100
```

Figure 5-7 Current JavaCL environment

5.2.4.3 Input Parameters

For passing the array of floats to the Kernel code described in the section *5.2.4.1* we are using the NIOUtils.directFloat API (JavaCL, n.d.). We would also need to generate the random variables described above to calculate the basket price.

The code for implementing the same is shown below –

```
CLContext clContext = JavaCL.createBestContext();
        CLQueue queue = clContext.createDefaultQueue();
        FloatBuffer marketData = NIOUtils.directFloats(15, clContext.getByteOrder());
        populate(marketData);
        ByteBuffer type = NIOUtils.directBytes(1, clContext.getByteOrder());
        setType(type);
        IntBuffer simSteps = NIOUtils.directInts(2, clContext.getByteOrder());
        setSimulationParameters(simSteps);
        FloatBuffer randNum = NIOUtils.directFloats(simSteps.get(0) * simSteps.get(1) *
3, clContext.getByteOrder());
        setRandomNumSettings(randNum, (long) simSteps.get(0) * simSteps.get(1) * 3);
        FloatBuffer output = NIOUtils.directFloats(1, clContext.getByteOrder());
        String programText = "";
        try{
            programText =
IOUtils.readText(JCLTest.class.getResource("MonteCarloBasket.cl")); // script name
        }catch (Exception e){
            e.printStackTrace();
        }
        CLBuffer<Float> marketDataBuff = clContext.createFloatBuffer(CLMem.Usage.Input,
marketData, true);
        CLBuffer<Byte> typeBuff = clContext.createByteBuffer(CLMem.Usage.Input, type,
false);
        CLBuffer<Integer> simStepsBuff = clContext.createIntBuffer(CLMem.Usage.Input,
simSteps, true);
        CLBuffer<Float> randNumBuff = clContext.createFloatBuffer(CLMem.Usage.Input,
randNum, true);
        CLBuffer<Float> outputBuff = clContext.createFloatBuffer(CLMem.Usage.Output,
output, false);

        CLProgram program = clContext.createProgram(programText);
        CLKernel kernel = program.createKernel("calcBasketPrice", marketDataBuff,
typeBuff, simStepsBuff, randNumBuff,outputBuff );
         // calcMCBasket is the method name in the kernel script

        CLEvent clEvent = kernel.enqueueNDRange(queue, new int[]{1});
        outputBuff.read(queue, clEvent); // read the output from the script.

        System.out.println("Basket Price = "+output.get());
```

Figure 5-8 Initial setup to call the cl kernel script and the relevant method call.

Below code illustrates the market data and different simulation parameters.

```
private static void setRandomNumSettings(FloatBuffer randNum, long iter) {
        Random random = new Random();
        for (int i = 0; i < iter; i++) {
            randNum.put(random.nextFloat());
        }

    }
```

```
private static void setSimulationParameters(IntBuffer simSteps) {
       simSteps.put(2);
       simSteps.put(3);
    }

    private static void setType(ByteBuffer type) {
        type.put((byte) 'C');
    }

    private static void populate(FloatBuffer marketData) {
        ……
    }
```

Figure 5-9 Passing the relevant parameters to the cl script.

5.2.4.4 Kernel script – vectorization

The kernel script is shown below. Although most of the code is trivial for this use-case as mentioned above (London, 2005), the inherent benefit is using the *dot product of vectors*.

The OpenCL (Scarpino, 2012) framework can also be leveraged to use other data types e.g., *float4* used here.

We can avoid costly loop operations here and optimize the valuation process to a great extent by creating subtasks and parallelizing the jobs.

Details on how to use the Work Sizes and offsets with respect to intricate inner loops can be found in the (Scarpino, 2012)

The interested reader can use the configuration to solve more complicated problems.

Lastly JavaCL (JavaCL, n.d.) gives a direct benefit of integrating the streaming pipeline code with the driver and the kernel script. It leverages the underlying hardware and no separate thread pools are needed to run the valuations.

A more subtle implementation can host a separate web/Spring boot service (Spring Boot, n.d.) to serve the valuation requests and asynchronously return the pricing results by calling the kernel.

```
__kernel void calcBasketPrice(__global const float *marketData, __global const char
*type, __global const int *simSteps, __global const float *randNum,
                          __global float *output )
{
      int numOfSim = simSteps[0];
      int numOfTimeSteps = simSteps[1];
      float stockPrice1, stockPrice2, stockPrice3 = 0.0;
      float rand1, rand2, rand3 = 0.0;
      float dtime = (float)(marketData[11])/(simSteps[1]);
      float sum = 0.0;
      for(int i =0; i < numOfSim; i++)
      {
         stockPrice1 =  marketData[0];
         stockPrice2 =  marketData[1];
         stockPrice3 =  marketData[2];
         for(int j =0 ; j < numOfTimeSteps ; j++)
         {
            rand1 = randNum[(int)(numOfTimeSteps * 3 * i + numOfTimeSteps * j + 0)];
            ……….
            float2 d11 = (float2)((float)stockPrice1, (float)stockPrice1) ;
            float2 d12 = (float2)(1, (float)(marketData[7] - marketData[8])*dtime);
            stockPrice1 = dot(d11, d12) + (float)(stockPrice1 * marketData[4]) +
(float)(sqrt((float)dtime) * rand1);

            float4 d21 = (float4)((float)stockPrice2, (float)stockPrice2,
(float)(stockPrice2 * marketData[5]), (float)(stockPrice2 * marketData[5]));
            float4 d22 = (float4)(1, (float)(marketData[7] - marketData[9])*dtime,
(float)(sqrt((float)dtime)* marketData[12]) * rand1, (float)sqrt((float)dtime)*
sqrt((float)(1- marketData[12]*marketData[12])) * rand2);
            stockPrice2 = dot(d21, d22);             // using the dot product
            float4 d310 = (float4)((float)stockPrice3 ,  (float)stockPrice3,
(float)stockPrice3 * marketData[6], (float)stockPrice3 * marketData[6]);
            float d311 = ((float)stockPrice3 * marketData[6]);
            float v = (float)sqrt((float)(1-marketData[12]*marketData[12]));
            float u = (float)(marketData[14] - marketData[12]*marketData[13]);
            ……….
            float x = (float)(a-b)*z;
            float4 d320 = (float4)(1, x,
(float)sqrt((float)dtime)*marketData[13]*rand1, sqrt((float)dtime)*(u/v)*rand2);
            float d321 = ((float)sqrt((float)dtime)* sqrt((float)(1 -
marketData[12]*marketData[12] - ((marketData[14] -
marketData[12]*marketData[13])/sqrt((float)(1-
marketData[12]*marketData[12])))))*rand3);

            stockPrice3  = dot(d310, d320) + d311 * d321 ; // using the dot product
         }

        ……….

        value  = fmax(stockPrice1 + stockPrice2 + stockPrice3 - marketData[3], 0 );
         }
          sum = sum + value;
          value = exp(-marketData[7] * marketData[11]) * sum;
          output[0] = value;
      }
}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Basket Price = 118.37463
```

Figure 5-10 Kernel script.

5.3 Designing a Cloud based solution-

In this section we quickly touch upon the usage of Amazon Web Services (AWS) as a cloud platform in the context of a risk and pricing application.

AWS Kinesis (Kinesis, n.d.) is a framework that supports streaming data processing and storage. Kinesis stores data in shards. This data can be later processed separately. Shards have specific data capacities, and the overall capacity of Kinesis stream is proportional to the number of shards in the stream.

On the other hand, AWS Firehose (Kinesis, n.d.) has no storage facility and is used directly in processing.

As such Firehose can use a Lambda (AWS Lambda, n.d.) function. Lambda function is used in AWS specifically for implementing event driven processing.

However, one cannot just use AWS Firehose for implementing the derivative pricing application. This is because we have separate market data producers, and it must be multiplexed and demultiplexed to retrigger the valuation after applying differential updates as described in Section *2.3.2*

We can use both AWS Kinesis and Firehose along with a multiplexer/demultiplexer component and a conflated pulsing mechanism for periodic processing of market data depending on other factors like User Interface. A gate helps to restrict sending the next batch of requests.

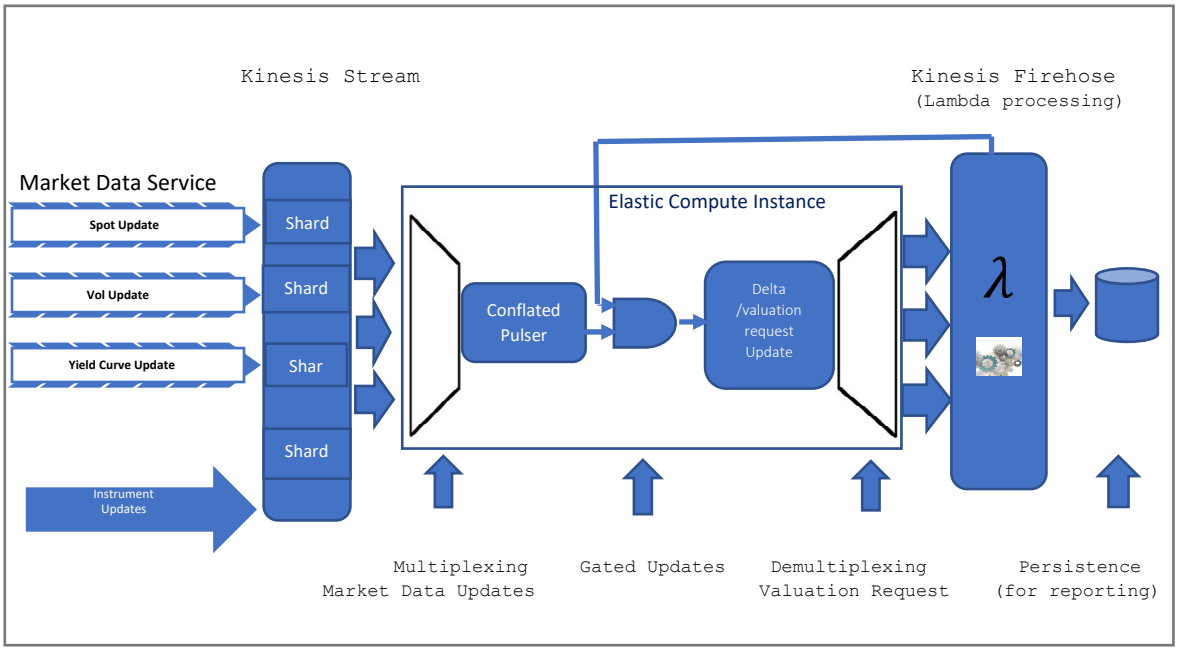

Figure 5-11 AWS based design diagram for derivative risk platform.

5.4 Conclusion

This chapter was dedicated towards implementation strategies other than the conventional ones e.g., custom streaming data updates, OpenCL implementation and cloud-based implementation. Most of them guarantee better performance and robustness and is a cut above most of the other implementation principles.

6. Scenario Generation

In section 1.6.3.3 we suggested using the following set of equations as the drivers behind the discriminator

$$\begin{cases} g(x) = e^{\frac{-1}{2}(\frac{2r}{\sigma^2}-1)x} & Eq\ (5) \\ u(x,\tau) = e^{-\tau\left[\frac{1}{4}\left(\frac{2r}{\sigma^2}+1\right)^2+\left(\frac{\pi}{T}\right)^2\right]}\sin\frac{\pi x}{T}\,g(x) & Eq\ (6) \end{cases}$$

6.1 Black Scholes Partial Differential Equation (PDE) and Heat Equation

Now we investigate if $u(x,\tau)$ satisfies the PDE for a *simple/vanilla call option*.

$$Let\ k_1 = \frac{2r}{\sigma^2},\ then\ Eq\ 5 \implies g(x) = e^{\frac{-1}{2}(k_1-1)x}\ and$$

$$Eq\ 6 \implies u(x,\tau) = e^{-\tau\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]}\sin\frac{\pi x}{T}\,g(x)$$

$$\implies v = e^{-\left(\tau\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]+\frac{1}{2}(k_1-1)x\right)}\sin\frac{\pi x}{T}\ \ (here, v = u(x,\tau))$$

By applying the necessary transformations from PDE to the heat equation and as mentioned in (Wilmott, Dewynne, & Howison, 1994) we must show that $Eq\ 6$ satisfies the following.

Note that, as k_1 is a ratio, we can easily control the values of σ and r or even change k_1

$$\frac{\partial v}{\partial \tau} = \frac{\partial^2 v}{\partial x^2} + (k_1 - 1)\frac{\partial v}{\partial x} - k_1 v$$

Now,

$$\frac{\partial v}{\partial \tau} = -e^{-\left(\tau\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]+\frac{1}{2}(k_1-1)x\right)}\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]sin\frac{\pi x}{T}$$

$$\frac{\partial v}{\partial x} = e^{-\left(\tau\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]+\frac{1}{2}(k_1-1)x\right)}\left\{\left(-\frac{1}{2}(k_1-1)\right)sin\frac{\pi x}{T} + \frac{\pi}{T}cos\frac{\pi x}{T}\right\}$$

$$\frac{\partial^2 v}{\partial x^2} = e^{-\left(\tau\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]+\frac{1}{2}(k_1-1)x\right)}\left(-\frac{1}{2}(k_1-1)\right)\left\{\left(-\frac{1}{2}(k_1-1)\right)sin\frac{\pi x}{T} + \frac{\pi}{T}cos\frac{\pi x}{T}\right\}$$
$$+ e^{-\left(\tau\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]+\frac{1}{2}(k_1-1)x\right)}\left\{\left(-\frac{\pi}{2T}(k_1-1)\right)cos\frac{\pi x}{T} - \left(\frac{\pi}{T}\right)^2 sin\frac{\pi x}{T}\right\}$$

Substituting $E = e^{-\left(\tau\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]+\frac{1}{2}(k_1-1)x\right)}$

$$\frac{\partial v}{\partial \tau} = -E\left[\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right]sin\frac{\pi x}{T}$$

$$\frac{\partial v}{\partial x} = E\left\{\left(-\frac{1}{2}(k_1-1)\right)sin\frac{\pi x}{T}+\frac{\pi}{T}cos\frac{\pi x}{T}\right\}$$

$$\frac{\partial^2 v}{\partial x^2} = E\left(-\frac{1}{2}(k_1-1)\right)\left\{\left(-\frac{1}{2}(k_1-1)\right)sin\frac{\pi x}{T}+\frac{\pi}{T}cos\frac{\pi x}{T}\right\}$$
$$+E\left\{\left(-\frac{\pi}{2T}(k_1-1)\right)cos\frac{\pi x}{T}-\left(\frac{\pi}{T}\right)^2 sin\frac{\pi x}{T}\right\}$$

$$\therefore \frac{\partial^2 v}{\partial x^2}+(k_1-1)\frac{\partial v}{\partial x}-k_1 v$$

$$= -E\frac{1}{2}(k_1-1)\left\{\frac{\pi}{T}cos\frac{\pi x}{T}-\frac{1}{2}(k_1-1)\,sin\frac{\pi x}{T}\right\}$$

$$-E\left\{\left(\frac{\pi}{2T}(k_1-1)\right)cos\frac{\pi x}{T}+\left(\frac{\pi}{T}\right)^2 sin\frac{\pi x}{T}\right\}$$

$$+E\,(k_1-1)\left\{\frac{\pi}{T}cos\frac{\pi x}{T}-\frac{1}{2}(k_1-1)\,sin\frac{\pi x}{T}\right\}$$

$$-E*k_1*\sin\frac{\pi x}{T}$$

$$= -E\frac{\pi}{2T}(k_1-1)\,cos\frac{\pi x}{T}-E\frac{\pi}{2T}(k_1-1)\,cos\frac{\pi x}{T}+E\,(k_1-1)\frac{\pi}{T}cos\frac{\pi x}{T}$$

$$+E\frac{1}{4}(k_1-1)^2\,sin\frac{\pi x}{T}-E\left(\frac{\pi}{T}\right)^2 sin\frac{\pi x}{T}-E\frac{1}{2}(k_1-1)^2\,sin\frac{\pi x}{T}-E*k_1*\sin\frac{\pi x}{T}$$

$$= -E\frac{1}{4}(k_1-1)^2\,sin\frac{\pi x}{T}-E*k_1*\sin\frac{\pi x}{T}-E\left(\frac{\pi}{T}\right)^2 sin\frac{\pi x}{T}$$

$$= -E\left(\frac{1}{4}(k_1-1)^2+k_1+\left(\frac{\pi}{T}\right)^2\right)sin\frac{\pi x}{T}$$

$$= -E\left(\frac{1}{4}(k_1+1)^2+\left(\frac{\pi}{T}\right)^2\right)sin\frac{\pi x}{T}$$

$$= \frac{\partial v}{\partial \tau}$$

6.2 Forward Backward Stochastic Neural Network - FBSNN

In FBSNN (Raissi, FBSNNs, 2018), we modify the exact equation for the discriminator by $u(x, \tau)$

```
import numpy as np
import tensorflow as tf
import math
from FBSNNs import FBSNN
……..

def u_exact(t, X):

if bool(model.generator):
    ……..
else:


return tf.math.exp(tf.multiply((-0.5 * (T - t) * model.sigma ** 2) * (
            (0.25 * (2 * model.r / model.sigma ** 2) + 1) ** 2 + (math.pi / T) **
2),model.g_tf_discrminator(X)))
# using u(x, tau) as the discriminator equation
```

```
def g_tf_discrminator(self, X):
    return tf.math.exp(tf.multiply(X, -0.5 * (2 * (self.r / self.sigma * self.sigma) -
1)))
```

Figure 6-1 Code snippet showing example of discriminator function.

The assumptions used in the derivation shown above are trivial and strictly relate to a call option. However, in a real-world context, any PDE to Heat Equation transformation will be of help to set the discriminator function. These are based on the boundary conditions of the asset type in question. Most of the models differ based on the underlying assumptions.

6.3 Generational Adversarial Network (GAN) Design

Before designing the actual GAN, we should understand some basic concepts for using a dense network in Keras as described in the (Keras, n.d.).

6.3.1 Regularizers

Regularizers apply penalties on layer parameters or layer activity during optimization and are added in the loss function. This is needed to prevent overfitting the network. Overfitting is a phenomenon that occurs when a function is tightly coupled to some data points that are used in the training phase.

Kernel regularizers apply penalty on the layer's kernel whereas bias and activity regularizers apply penalties on the bias and output, respectively.

In FBSNN (Raissi, FBSNNs, 2018), most of the input are generated by the model itself using random number generation and there is a high probability of overfitting. So , we apply all the three regularizers in the design.

6.3.2 Leaky ReLU

As defined in (Goodfellow, et al., 2014) In a deep learning context an activation function is some transform that is applied on the input before sending to the next layer or neurons.

Rectified Linear Unit (ReLU) is an activation function defined as –

$$\begin{cases} f(x) = & 0, \quad when\ x < 0 \\ & x, \quad when\ x \geq 0 \end{cases}$$

In ReLU if the activation unit is not active the signal is lost.

Leaky ReLU handles this case by allowing a small gradient when the unit is not active based on a parameter $alpha$. For Keras its defined as

$$\begin{cases} f(x) = & alpha * x, \quad when\ x < 0 \\ & x, \quad when\ x \geq 0 \end{cases}$$

6.3.3 Batch Normalization

As defined in Keras documentation, batch normalization applies a transformation that maintains the mean output close to 0 and the standard deviation close to 1. There are separate techniques used during training and testing.

As the input to the discriminator is mostly noise its imperative that we use batch normalization.

6.3.4 Sequential and Dense network

In Keras, a sequential network is one where we have a stack of layers where each layer has exactly one input tensor and one output tensor. Tensors are multi-dimensional arrays with a uniform type.

Similarly, Keras defines a Dense layer as the regular deeply connected neural network.

6.4 Network Design

We now define the actual network design.

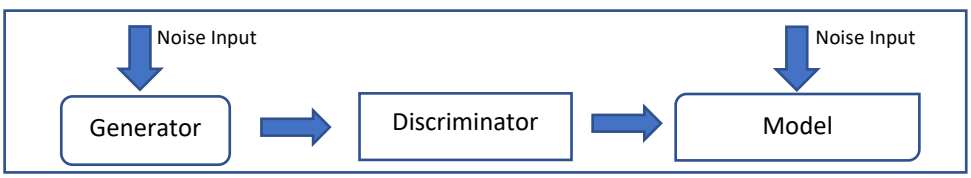

Figure 6-2 High level design of GAN

Following code demonstrates the generator and discriminator –

```
def generator(self):
    model = Sequential()
    model.add(Flatten(input_shape=(100, 51, 100)))
    model.add(Dense(256,
                    kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
                    bias_regularizer=regularizers.l2(1e-4),
                    activity_regularizer=regularizers.l2(1e-5)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512,
                    kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
                    bias_regularizer=regularizers.l2(1e-4),
                    activity_regularizer=regularizers.l2(1e-5)
                    ))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(1024,
                    kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
                    bias_regularizer=regularizers.l2(1e-4),
                    activity_regularizer=regularizers.l2(1e-5)
                    ))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(np.prod(self.img_shape), activation='tanh'))
    model.add(Reshape(self.img_shape))
    model.summary()
    noise = Input(shape=(100, 51, 100))
    img = model(noise)

    return Model(noise, img)
```

```
def discriminator(self):
    model = Sequential()
    model.add(Flatten(input_shape=(100, 51, 1)))
    model.add(Dense(512,
                    kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
                    bias_regularizer=regularizers.l2(1e-4),
                    activity_regularizer=regularizers.l2(1e-5)
                    ))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(256,
                    kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
                    bias_regularizer=regularizers.l2(1e-4),
                    activity_regularizer=regularizers.l2(1e-5)
                    ))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()

    img = Input(shape=(100, 51, 1))
    validity = model(img)
    return Model(img, validity)
```

Figure 6-3 Generator and Discriminator

6.4.1 Training and Inference

For training we create two instances of BlackScholesBarenblatt (Raissi, FBSNNs, 2018),– one as a generator and the other as a discriminator. We are using 10 iterations with a learning rate of 0.0001 as an example.

```
def train(self, epochs, batch_size=128):
    ……..
    self.Xi = np.array([1.0, 0.5] * int(D / 2))[None, :]
T = 1.0

self.realModel = BlackScholesBarenblatt(self.Xi, T, M, N, D, layers, False)
self.realModel.train(N_Iter=10, learning_rate=1e-4)

self.fakeModel = BlackScholesBarenblatt(self.Xi, T, M, N, D, layers, True)
self.fakeModel.train(N_Iter=10, learning_rate=1e-4)
    ……
real_data_list = numpy.zeros(shape=(batch_size, 100, 51, 1))
fake_data_list = numpy.zeros(shape=(batch_size, 100, 51, 1))

real_data_combined = numpy.zeros(shape=(batch_size, 100, 51, 100))
for i in range(batch_size):
    t, w = self.realModel.fetch_minibatch()
    x_pred_real, y_pred_real = self.realModel.predict(self.Xi, t, w)
    x_pred_fake, y_pred_fake = self.fakeModel.predict(self.Xi, t, w)
……
    fake_data = self.generator.predict(tf.expand_dims(x_pred_fake, 0), steps=1)

    mmx = MinMaxScaler()
    y_real_pred_transfromed = mmx.fit_transform(y_pred_real.reshape(100, 51))
    y_real_pred_transfromed = y_real_pred_transfromed.reshape(100, 51, 1)

    real_data_list[i - 1] = y_real_pred_transfromed

    fake data transfromed = mmx.fit transform(fake data.reshape(100, 51))
    fake_data_transfromed = fake_data_transfromed.reshape(1, 100, 51, 1)

    fake_data_list[i - 1] = fake_data_transfromed

    t, w = self.realModel.fetch_minibatch()
    x_pred_real, y_pred_real = self.realModel.predict(self.Xi, t, w)

    y_real_pred_transfromed = mmx.fit_transform(y_pred_real.reshape(100, 51))
    y_real_pred_transfromed = y_real_pred_transfromed.reshape(100, 51, 1)

    real_data_combined[i - 1] = y_real_pred_transfromed

# Train the discriminator
d_loss_real = self.discriminator.fit(real_data_list, valid, epochs=epochs,
batch_size=batch_size, validation_split=0.2)
d_loss_fake = self.discriminator.fit(fake_data_list, fake, epochs=epochs,
batch_size=batch_size, validation_split=0.2)
……

def sample_scenario(self):
    t, w = self.fakeModel.fetch_minibatch()
    x_pred_fake, y_pred_fake = self.fakeModel.predict(self.Xi, t, w)
    real_data = self.generator.predict(tf.expand_dims(x_pred_fake, 0), steps=1)
    print("Generated scenario data for ", real_data)
```

Figure 6-4 Training and sample scenario – code snippet

6.4.2 Sample Output and Training Plots

Sample Output and some plot showing the learning rate and relevant data are shown as samples.

```
>>> y_pred_fake.shape
(100, 51, 1)
>>> y_pred_fake.reshape(100, 51)
array([[4.1756487, 4.1822476, 4.1896415, ..., 4.1614842, 4.167003 ,
        4.1665416],
       [4.1756487, 4.178204 , 4.161512 , ..., 4.2558894, 4.20992  ,
        4.195654 ],
       [4.1756487, 4.162927 , 4.1672354, ..., 4.001324 , 3.9739842,
        3.915312 ],
       ...,
       [4.1756487, 4.1734705, 4.185445 , ..., 3.9521353, 3.9292128,
        3.9545057],
       [4.1756487, 4.161788 , 4.1895785, ..., 3.914036 , 3.9240046,
        3.9391062],
       [4.1756487, 4.172854 , 4.138704 , ..., 4.140717 , 4.1629996,
        4.1607533]], dtype=float32)
>>> y_pred_fake.reshape(100, 51)[0]
array([4.1756487, 4.1822476, 4.1896415, 4.207899 , 4.2237496, 4.225791 ,
       4.194943 , 4.1862016, 4.1857038, 4.164434 , 4.124925 , 4.1227036,
       4.1731963, 4.130196 , 4.0985336, 4.089562 , 4.113567 , 4.14272  ,
       4.104445 , 4.1232104, 4.1579633, 4.187429 , 4.181387 , 4.1631513,
       4.1789784, 4.1791716, 4.1407356, 4.108681 , 4.1300626, 4.1652837,
       4.2138953, 4.208172 , 4.175188 , 4.203029 , 4.181392 , 4.172046 ,
       4.165961 , 4.122643 , 4.105021 , 4.116936 , 4.127142 , 4.128814 ,
       4.1319723, 4.110916 , 4.1533113, 4.1435804, 4.1631074, 4.1430464,
       4.1614842, 4.167003 , 4.1665416], dtype=float32)
>>> y_pred_fake.reshape(100, 51)[0][0]
4.1756487
```

Figure 6-5 Sample Output

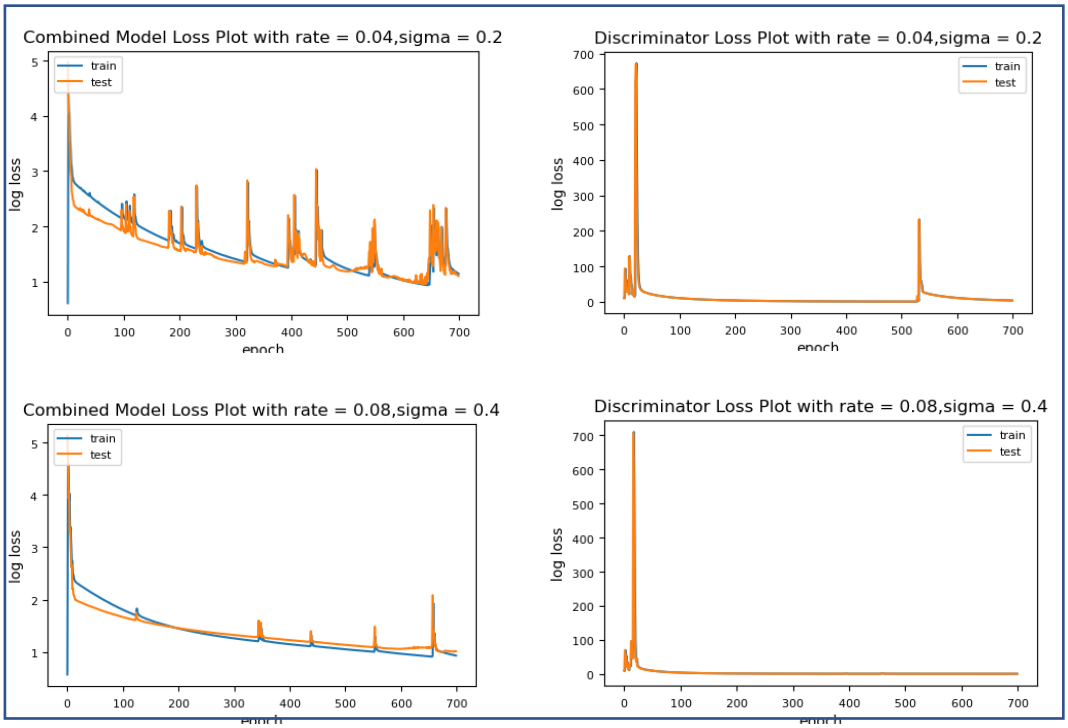

Figure 6-6 Sample Plots

6.5 Conclusion

In this chapter the focus was more around generating new scenarios using the Forward Backward Stochastic Neural Network – FBSNN (Raissi, FBSNNs, 2018),. To design the Generational Adversarial Network (GAN) (Goodfellow, et al., 2014), we have designed a pair of generator and discriminator network and chosen a specific function pertaining to a Black Scholes PDE simple call option model (Wilmott, Dewynne, & Howison, 1994) that satisfy the heat equation after transformation. These equations are used in the FBSNN instances and passed through the training and testing phases to generate new scenarios.

7. Conclusion and Future Scope of Work

This work touches upon the design and implementation of various facets of a derivative risk application. To make the application robust, scalable, extensible, fault tolerant, reliable and with somewhat low latency we need to consider several notions of distributed event driven systems.

We investigated the theoretical basis of distributed systems and built the implementation on top of these ideas by carefully choosing the reactive design pattern. We also used some robust frameworks like Spring Boot, Kafka and Redis to validate our design. These frameworks are highly flexible and even cloud ready.

Future scope of work can have more emphasis on implementing various strategies of resource allocation like thread pools, memory maps and making these more configurable. A production ready implementation demands a more configuration driven approach where the behavior of the systems can be modified without code deployment. As mentioned in the introduction a front office setup technically consists of a variety of desks and each of them can have various compute requirement. So, it is imperative that we consider performance monitoring and based on the results come up with separate configurations for each desk.

One important characteristic of the system is designing a performing User Interface (UI). For an application like this, we can use React JS (React, n.d.) as a possible UI solution. Most of the end users perform various operations like grouping, sorting along with average, max, min etc. As we are talking about various data types, real time streaming and cases of simple or nested joining of data we might have to consider transferring some part of the lightweight calculations to the user interface thin client.

Future work can also aim at enhancing the custom streaming framework by carefully considering the performance implications and integrating GPU (GPU, n.d.) related implementations. Streaming frameworks suffer from multi-threading issues like usage of locks and indiscriminate use of new objects on the heap. Although on the surface things may look to be working fine but when doing a performance analysis, we can easily see the glitch.

A resounding technique in these cases is to limit creation of threads where not needed and move to an off-heap implementation (Lawrey, n.d.)in Java. GPU implementation would also need a carefully designed model library that can take advantage of techniques like vectorization in the algorithms.

Most of the business objects used in the implementations should be made immutable to the extent possible and strategies like differential updates discussed here may be used.

Also, a caching framework should not be used as a messaging layer by indiscriminately depending on event creations. This causes a bottleneck in the server that can be extremely hard to debug and resolve as most of the times the threading model behavior of a third-party caching implementation is not known or cannot be changed.

To keep track of already processed stages of valuation request one can go for the use of checkpointing (Akidau, Chernyak, & Lax, 2018), (Kleppman, 2017) in the orchestrator. So, if an

engine is not responding in the middle of a valuation, another one can take over from the next stage by referring the last known good position in the valuation state. Moreover, we can go for replicating the persistent memory mapped files to another host in a hot – hot state i.e., another engine can be on standby where the state has been replicated and it can easily start the valuation from that point onwards.

Generational Adversarial Networks (Goodfellow, et al., 2014) are a breakthrough in the deep learning framework (Goodfellow, Bengio, & Courville, 2016) that can easily create somewhat true copies of data. Although what we showed here is more of a toy implementation future work can use the valuation models used by the desk and easily map each asset type to a discriminator function and create scenarios by changing the parameters. This way one is not bound to a handful of scenarios for monitoring the health of the bank.

Bibliography

Akidau, T., Chernyak, S., & Lax, R. (2018). *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing.* O'Reilly Media.

Allen, S. (2012). *Financial Risk Management - A Practitioner's Guide to Managing Market and Credit Risk.* Wiley.

antlr. (n.d.). Retrieved from antlr: https://www.antlr.org/

AtomicLongArray. (n.d.). Retrieved from AtomicLongArray: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLongArray.html

AWS Lambda. (n.d.). Retrieved from AWS Lambda: https://aws.amazon.com/lambda/

Ballabio, L. (2020). *Implementing QuantLib: Quantitative Finance in C++: an Inside Look at the Architecture of the QuantLib Library.* Autopubblicato.

Ballabio, L. (n.d.). *QuantLib*. Retrieved from QuantLib: https://www.quantlib.org/

Bloch, J. (2016). *Effective Java.* Pearson Education.

Chandy, K. M., & Lamport, L. (1985, February). Distributed Snapshots: Determining Global Snapshots of Distributed Systems. *ACM Transactions on Computer Systems, 3*(1), 63-75.

Dempster, M. (2002). *Risk Management Value at Risk and Beyond.* Cambridge University Press.

Evans, L. C. (2010). *Partial Differential Equations* (Second ed.). American Mathematical Society.

Framework, I. S. (2014). *Gutierrez, Felipe.* Apress.

Gamma, E., Helm, R., Johnson, R. ,., & John. (1994). *Design Patterns -Elements of Reusable Object-Oriented Software.* Pearson Education.

Gao, T., Qiu, x., & He, L. (2013). Improved RETE Algorithm in Context Reasoning for Web of Things Environments. *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, (pp. 1044-1049). doi:10.1109/GreenCom-iThings-CPSCom.2013.177

generative-adversarial-network. (n.d.). Retrieved from Deep Learning for Java: https://mgubaidullin.github.io/deeplearning4j-docs/generative-adversarial-network

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., . . . Bengio, Y. (2014). Generative Adversarial Nets. *Proceedings of the 27th International Conference on Neural Information Processing Systems. 2*, pp. 2672–2680. Montreal, Canada: MIT Press.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.

GPU. (n.d.). Retrieved from GPU: https://en.wikipedia.org/wiki/Graphics_processing_unit

Guava. (n.d.). Retrieved from Guava: https://github.com/google/guava/wiki/CachesExplained

Hull, J., & Basu, S. (2016). *Options, futures and other derivatives.* Pearson.

JavaCL. (n.d.). Retrieved from JavaCL: https://github.com/nativelibs4java/JavaCL

JSON. (n.d.). Retrieved from JSON: https://www.json.org/json-en.html

Kafka. (n.d.). Retrieved from Kafka: https://kafka.apache.org/

Keras. (n.d.). Retrieved from Keras: https://keras.io/

Kinesis. (n.d.). Retrieved from Amazon Web Services: https://aws.amazon.com/kinesis/

Kleppman, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly .

Kryo. (n.d.). Retrieved from Kryo: https://github.com/EsotericSoftware/kryo

Laignelet, A. (2019). *Deep Learning of High dimensional Partial Differential Equations.* Imperial College, London.

Larsson, M. (2019). *Hands-On Microservices with Spring Boot and Spring Cloud.* Packt Publishing Ltd.

Lawrey, P. (n.d.). *On Heap vs Off Heap Memory Usage*. Retrieved from On Heap vs Off Heap Memory Usage: https://dzone.com/articles/heap-vs-heap-memory-usage

Limited, E. C. (2006). *Business Knowledge for IT in Investment Banking -A Complete Handbook for IT Professionals.* Essvale Corporation.

London, J. (2005). *Modeling Derivatives in C++.* Hoboken,New Jersey: John Wiley & Sons, Inc.

Metsker, S. J., & Wake, W. C. (2006). *Design Patterns in Java.* Pearson Education.

Narkhede, N., Shapira, G., & Palino, T. (2017). *Kafka The Definitive Guide.* O'Reilly Media Inc.

Nurkiewicz, T., Christiansen, B., & Meijer, E. (2016). *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications.* O'Reilly Media, Inc.

NYSE. (n.d.). Retrieved from NYSE: https://www.nyse.com/index

Patterson, S. (2010). *The Quants.* Crown.

Raissi, M. (2018). *FBSNNs*. Retrieved from https://github.com/maziarraissi/FBSNNs

Raissi, M. (2018). Forward-Backward Stochastic Neural Networks: Deep Learning of High-dimensional Partial Differential Equations. *arXiv preprint arXiv:1804.07010*, 1, 15, 16, 18.

React. (n.d.). Retrieved from React: https://reactjs.org/

Redis. (n.d.). Retrieved from Redis: https://redis.io/

Rete algorithm. (n.d.). Retrieved from Rete algorithm: https://en.wikipedia.org/wiki/Rete_algorithm

S&P 500. (n.d.). Retrieved from S&P 500: https://en.wikipedia.org/wiki/S%26P_500

Samoylov, N., & Nield, T. (2020). *Learning RxJava* (Second ed.). Packt Publishing.

Scarpino, M. (2012). *OpenCL in Action.* Manning Publications Co.

Single Writer Principle. (n.d.). Retrieved from Single Writer Principle: https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.html#:~:text=The%20Single%20Writer%20Principle%20is,execution%20context%20for%20all%20mutations.

Spring Boot. (n.d.). Retrieved from Spring Boot: https://spring.io/projects/spring-boot

Spring Framework. (n.d.). Retrieved from Spring Framework: https://spring.io/projects/spring-framework

Szylar, C. (2013). *Handbook of Market Risk.* Wiley Publishers.

Tensorflow. (n.d.). Retrieved from Tensorflow: https://www.tensorflow.org/

The Heat Equation. (n.d.). Retrieved from Paul's Online Notes: https://tutorial.math.lamar.edu/classes/de/theheatequation.aspx#:~:text=If%20Q(x%2Ct),at%20that%20location%20and%20time.&text=While%20this%20is%20a%20nice,actually%20something%20we%20can%20solve.

Tulchinsky, I. (2015). *Finding Alphas A Quantitative Approach to Building Trading Strategies.* Wiley.

Wilmott, P., Dewynne, J., & Howison, S. (1994). *Option Pricing Mathematical models and computation.*

YAML. (n.d.). Retrieved from YAML: https://yaml.org/

Table of Figures