Moving DataSketches-Java to Java 17+

Background

Sketches are small state machines used to analyze massive data streams. In the context of large systems that might be managing millions of sketches, achieving high performance requires the ability to act on a memory region of bytes as follows:

- Allocate and immediately deallocate off-heap memory regions.
- Access large memory regions with an index that is larger than the a Java Integer.MAX_VALUE.
- Read and write multibyte primitives and primitive arrays into data structures at high speed;
    - and then read them as byte arrays without any data movement or code loops; and visa versa.
- Provide sub-region views of the original region without any data movement.
- Provide import an export interfaces with ByteBuffer.

Although the ByteBuffer provided a subset of these capabilities, its API is limited, and and it has other drawbacks.

> In March of 2014 Oracle started the Panama project specifically to address the above shortcomings and a whole lot more. Panama released its first LTS version in Java 17 (14 Sep 2021), its second LTS version in Java 21 (19 Sep 2023). Panama, now known as the Foreign Function & Memory API (FFM), is now formally integrated into the Java language as of Java 22 (9 Mar 2024). The first LTS version of Java with fully integrated FFM will be Java 25, expected in September, 2025. See Java Release Table

When the DataSketches project started as an internal project at Yahoo in 2011, Java was at version 7, which, of course, lacked the above capabilites. The only way around the deficiencies in the language was to leverage internal JVM classes, such as Unsafe, VM, Bits, and Cleaner. To do this we created the DataSketches Memory project (26 May 2017), which implemented the features we needed and leveraged the JVM internals.

Java 9 (21 Sep 2017) introduced JPMS and also started restricting access to the JVM internals. By utilizing special configurations of the JVM, we were able to still access what we needed with Java 11 (25 Sep 2018). But with Java 17 (14 Sep 2021), we are no longer able to do this. Meanwhile, Panama introduced its first FFM API with Java 17, which is usable, but not API stable, as more substantial (and breaking) changes came with Java 21 (19 Sep 2023).

Many of our customers are large platforms, which tend to lag in their adoption of newer versions of Java, and we have been able to satisfy most all of our customers with our support of Java 8 and 11. However, now they

are starting to migrate to Java 17 and later. It is also significant that a number of tools that our developers use are also migrating to new versions of Java, and some have stopped supporting earlier versions of Java altogether! E.g., TestNG has stopped supporting its last version (7.5.1) that supports Java 8. Apache Maven 4.0 will require Java 17 (or 21).

There are other major technologies at play here.

- JPMS
    - Provides additional clarity and security of dependencies.
    - Enable partitioning our library down to individual sketches if desired. This would allow users to integrate just the sketches they want without having to integrate the entire library.
    - Provides cleaner separation of our classes into what is publicly available and what is not.
    - Allows user integration of our library into either the user's Classpath or Modulepath.
- MR-JARS & Toolchains
    - This enables the code of multiple releases into one combined Multiple-Release JAR and the Java release selected is configured into a Toolchain.
- Build System
    - Historically we have used Apache Maven for this.
    - Other large platforms like Apache Lucene have chosen Gradle, which appears to be more flexible, but has its own idiosyncrasies and would be a big learning curve for us.

The task at hand is to propose a strategy for how we should transition our DataSketches-Java, with its dependency on DataSketches-Memory, so that we can fully support our users on Java 17 and beyond.

Options

The above technologies are pretty much independent of each other so we have some flexibility in the order in which to execute the following.

1. Convert the DS-Java library from using DS-Memory to FFM (as of Java 17).
    - This version of the library would hava a minimum requirement of Java 17.
    - Users of Java 8 and 11, would still rely on earlier version of DS-Java.
    - We would continue to support bug-fixes on earlier DS-Java versions but not enhancements.
    - This, by itself, would not require the use of MR-JARS and Toolchains, but once we want to support FFM in Java 21, it would make a lot of sense.
    - The benefits include eliminating the dependency on DS-Memory, and improved performance.

    - This should make transitions to newer Java versions much easier.
    - This represents quite a bit of work which cannot easily be divided into smaller chunks.

2. Refactor DS-Java into several Java Modules and at the same time refactor the build system to support them. This would have a minimum requirement of Java 11, but it might make more sense to set the minimum requirement at Java 17 as above. There are seveal benefits to refactoring into Java modules, which include:

    - Users could import only the sections of the library they need.
    - Cleaner library structure with well defined dependencies.
    - Simplifies much of the task of the build system.
    - If this were done before (1) above it would allow breaking up that task into smaller chunks.
    - Whether to structure our releases around the individual modules or not is a separate decision.
    - This represents quite a bit of work.

3. Switch the Build System from Maven to Gradle

    - This would represent a great deal of work and a new learning curve, but would theoretically offer more flexibility and faster build times, and a lot more dependence on scripting.