**UE Software Engineering**

# Software Engineering Project 2025

## Bit Packing:
## compressing for sped up transmission

Prepared By:
**Abdelbaki Kacem**

2025

# Contents

# 1 Introduction

This document presents integer compression techniques using bit packing. The project demonstrates three approaches to compress integer arrays by storing values in fewer bits than the standard 32-bit representation.

Applications include:

- Large-scale data processing
- Database indexing
- Cache-efficient computations
- Real-time data streaming

# 2 Project Overview

## 2.1 Motivation

Standard Java integers consume 32 bits. Bit packing optimizes storage by:

1. Determining minimum bits required
2. Packing multiple values into fewer integers
3. Providing efficient element retrieval

## 2.2 Classes

- BitPacking.java: Main driver
- BaseCompression.java: Utilities
- NoSpillCompression.java: No boundary crossing
- SpillCompression.java: Boundary crossing
- OverflowCompression.java: Hybrid approach

# 3 Base Compression

Listing 1: BaseCompression.java

```java
package compression;

import java.util.ArrayList;

public class BaseCompression {
    public String convertToBinary(int i) {
        return i > 0 ? "0".concat(Integer.toBinaryString(i)) : Integer.toBinaryString(i);
    }

    protected Integer convertToInteger(String b) {
        return Integer.parseUnsignedInt(b, 2);
    }

    public ArrayList<String> convertArrayToBinary(ArrayList<Integer> decompressedList, int bits) {
        ArrayList<String> binaryDecompressedList = new ArrayList<String>();
        decompressedList.forEach(i -> {
            String binary = Integer.toBinaryString(i);
            if (binary.length() < bits)
                binary = "0".repeat(bits - binary.length()) + binary;
            else if (binary.length() > bits)
                binary = binary.substring(binary.length() - bits);
            binaryDecompressedList.add(binary);
        });
        return binaryDecompressedList;
    }

    public ArrayList<Integer> convertArrayToInteger(ArrayList<String> binaryCompressedArray) {
        ArrayList<Integer> integerCompressedArray = new ArrayList<>();
        for (String binary : binaryCompressedArray)
            integerCompressedArray.add(convertToInteger(binary));
        return integerCompressedArray;
    }
}
```

# 4 Compression Algorithms

## 4.1 No-Spill Compression

Values packed without crossing 32-bit boundaries.

Listing 2: NoSpillCompression.java

```java
package compression;

import java.util.ArrayList;

```

```java
public class NoSpillCompression extends BaseCompression {
    public ArrayList<String> compress(ArrayList<String> binaryDecompressedArray, int bits) {
        long startTime = System.nanoTime();
        int maxBits = 32, numberOfValuesPerInt = maxBits / bits;
        ArrayList<String> binaryCompressedArray = new ArrayList<>();
        String binary;
        for (int i = 0; i < binaryDecompressedArray.size(); i += numberOfValuesPerInt) {
            binary = "";
            for (int j = i; j < Math.min(i + numberOfValuesPerInt, binaryDecompressedArray.size()); j++)
                binary = binary.concat(binaryDecompressedArray.get(j));
            binaryCompressedArray.add(binary);
        }
        long endTime = System.nanoTime();
        long duration = endTime - startTime;
        System.out.println("Execution time (No Spill): " + duration + " nano seconds");
        return binaryCompressedArray;
    }

    public int get(ArrayList<String> binaryCompressedArray, int bits, int i) {
        int maxBits = 32, numberOfValuesPerInt = maxBits / bits, BCi = i / numberOfValuesPerInt, startFromBit = (i % numberOfValuesPerInt) * bits;
        String binary = binaryCompressedArray.get(BCi).substring(startFromBit, startFromBit + bits);
        return convertToInteger(binary);
    }

    public ArrayList<Integer> decompress(ArrayList<String> compressedArray, int originalSize, int bits) {
        ArrayList<Integer> decompressedList = new ArrayList<>();
        int maxBits = 32, numberOfValuesPerInt = maxBits / bits;

        for (int i = 0; i < originalSize; i++) {
            int maxBitsTemp = 32, numberOfValuesPerIntTemp = maxBitsTemp / bits, BCi = i / numberOfValuesPerIntTemp, startFromBit = (i % numberOfValuesPerIntTemp) * bits;
            String binary = compressedArray.get(BCi).substring(startFromBit, startFromBit + bits);
            decompressedList.add(convertToInteger(binary));
        }

        return decompressedList;
    }
}
```

## 4.2 Spill Compression

Values allowed to cross 32-bit boundaries for better compression.

Listing 3: SpillCompression.java

```
package compression;

import java.util.ArrayList;

public class SpillCompression extends BaseCompression {
    public ArrayList<String> compress(ArrayList<String>
       binaryDecompressedArray) {
        long startTime = System.nanoTime();
        int maxBits = 32;
        ArrayList<String> binaryCompressedArray = new ArrayList<>();
        String binary = "";
        for (String b : binaryDecompressedArray)
            binary = binary.concat(b);
        while (!binary.isEmpty()) {
            binaryCompressedArray.add(binary.substring(0, Math.min(
               maxBits, binary.length())));
            binary = binary.substring(Math.min(maxBits, binary.
               length()));
        }
        long endTime = System.nanoTime();
        long duration = endTime - startTime;
        System.out.println("Execution time (Spill): " + duration + "
            nano seconds");
        return binaryCompressedArray;
    }

    public int get(ArrayList<String> binaryCompressedArray, int bits
       , int i) {
        int maxBits = 32, startFromBitGlobally = i * bits, BCi =
           startFromBitGlobally / maxBits, startFromBitLocally =
           startFromBitGlobally % maxBits;
        String compressed = binaryCompressedArray.get(BCi);
        if (startFromBitLocally + bits <= compressed.length()) {
            String binary = compressed.substring(startFromBitLocally
               , startFromBitLocally + bits);
            return convertToInteger(binary);
        } else {
            String firstPart = compressed.substring(
               startFromBitLocally);
            int neededFromNext = bits - (compressed.length() -
               startFromBitLocally);
            String secondPart = binaryCompressedArray.get(BCi + 1).
               substring(0, neededFromNext);
            String binary = firstPart + secondPart;
            return convertToInteger(binary);
        }
    }

    public ArrayList<Integer> decompress(ArrayList<String>
       compressedArray, int originalSize, int bits) {
        ArrayList<Integer> decompressedList = new ArrayList<>();
        int maxBits = 32;

        for (int i = 0; i < originalSize; i++) {
            int startFromBitGlobally = i * bits, BCi =
               startFromBitGlobally / maxBits, startFromBitLocally =
                startFromBitGlobally % maxBits;
            String compressed = compressedArray.get(BCi);
            String binary;

            if (startFromBitLocally + bits <= compressed.length()) {
                binary = compressed.substring(startFromBitLocally,
                   startFromBitLocally + bits);
            } else {
                String firstPart = compressed.substring(
                   startFromBitLocally);
                int neededFromNext = bits - (compressed.length() -
                   startFromBitLocally);
                String secondPart = compressedArray.get(BCi + 1).
                   substring(0, neededFromNext);
                binary = firstPart + secondPart;
            }
            decompressedList.add(convertToInteger(binary));
        }

        return decompressedList;
    }
}
```

## 4.3 Overflow Compression

Hybrid approach using threshold-based mechanism.

# 5 Main Driver

Listing 4: BitPacking.java

```java
package compression;

import org.openjdk.jol.info.*;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Random;

public class BitPacking {

    enum CompressionType {
        NO_SPILL, SPILL, OVERFLOW, ALL
    }

    public static void main(String[] args) {
        System.out.println("STARTING EXECUTION");
        long startTime = System.currentTimeMillis();

        if (args.length == 0) {
            System.err.println("Please provide a compression type: NO_SPILL, SPILL, OVERFLOW, or ALL");
            return;
        }

        CompressionType compressionType = null;
        try {
            compressionType = CompressionType.valueOf(args[0].toUpperCase());
        } catch (IllegalArgumentException e) {
            System.err.println("Invalid compression type. Use: NO_SPILL, SPILL, OVERFLOW, or ALL");
            return;
        }

        int x = Integer.MIN_VALUE;
        ArrayList<Integer> decompressedList = new ArrayList<>();
        Random rand = new Random();
        int listSize = 1000;
        for (int i = 0; i < listSize; i++)
            decompressedList.add(rand.nextInt(0, 1000));

        BaseCompression baseCompression = new BaseCompression();
        int max = Collections.max(decompressedList, Comparator.comparingInt(i -> Integer.toBinaryString(i).length()));
        String binaryMax = baseCompression.convertToBinary(max);
        int bits = binaryMax.length();
    }
}
```

# 6 Use Cases

- No-Spill: Aligned bit widths
- Spill: Maximum compression
- Overflow: Mixed-range data

# 7 Usage

## 7.1 Compile

```
javac -cp jol-core.jar -d out
    src/compression/*.java
```

## 7.2 Run

```
java -cp out:jol-core.jar
    compression.BitPacking <TYPE>
```

where `<TYPE>` can be:

- `NO_SPILL` - Run only the No-Spill compression algorithm
- `SPILL` - Run only the Spill compression algorithm
- `OVERFLOW` - Run only the Overflow compression algorithm
- `ALL` - Run and compare all compression algorithms

Example:

```
java -cp out:jol-core.jar
    compression.BitPacking ALL
```

# 8 Conclusion

This project demonstrates practical integer compression:

- No-Spill: Simple and predictable
- Spill: Maximum compression
- Overflow: Adaptive approach

Benefits:

- Correct compression/decompression
- Efficient random access
- Benchmarking support
- Modular design