springone 2GX

DALLAS

2014

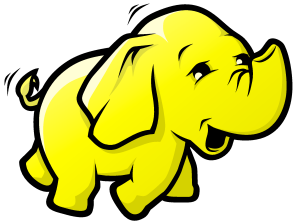
Big Data in Memory

John Davies

springone 2GX

Memory and Big Data

- The path to or from Big Data is memory
- Whatever you write into your Big Data gets there through, at some point, memory
  - At some point a CPU reads it, does something and writes it
- If you want to process it (say Hadoop) then you need memory again



- We take memory for granted but with Big Data we really need a lot of the stuff

Why Big Data in memory?

- The main use is real-time processing
  - If you've had time to store the data then it's not real-time
- Another is performance, if you can query your data in memory then you're not hitting the disk
- It's also the more logical way to work with data
  - If you were working with "normal data" you'd be working in memory
  - You wouldn't be writing everything to disk

A repeat of the old days?

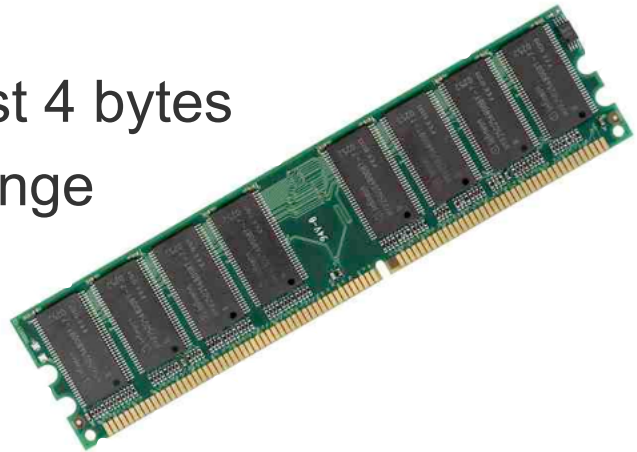
- Go back a few years and we used to write everything to disk and then query it
- Those were the “hay day” of the relational database where the DBA was God (or next to God if you’re religious)
- Then we started to see in-memory caches and finally in-memory databases

- We can’t let Big Data drive away the programmers and JVM
  - We need to fight back!

Java IS the problem!

- Java is very inefficient at storing data in memory

- Take the string “ABC”, typically it needs just 4 bytes
  - Three if you know the length won’t change



- Java takes 48 bytes to store “ABC” as a String
  - It optimises only by using 24 byte references for reuse of the same String

Start Simple…

- Simple data we're going to be referring to for the next few slides…

ID	TradeDate	BuySell	Currency1	Amount1	Exchange Rate	Currency2	Amount2	Settlement Date
1	21/07/2014	Buy	EUR	50,000,000.00	1.344	USD	67,200,000.00	28/07/2014
2	21/07/2014	Sell	USD	35,000,000.00	0.7441	EUR	26,043,500.00	20/08/2014
3	22/07/2014	Buy	GBP	7,000,000.00	172.99	JPY	1,210,930,000.00	05/08/2014
4	23/07/2014	Sell	AUD	13,500,000.00	0.9408	USD	12,700,800.00	22/08/2014
5	24/07/2014	Buy	EUR	11,000,000.00	1.2148	CHF	13,362,800.00	31/07/2014
6	24/07/2014	Buy	CHF	6,000,000.00	0.6513	GBP	3,907,800.00	31/07/2014
7	25/07/2014	Sell	JPY	150,000,000.00	0.6513	EUR	97,695,000.00	08/08/2014
8	25/07/2014	Sell	CAD	17,500,000.00	0.9025	USD	15,793,750.00	01/08/2014
9	28/07/2014	Buy	GBP	7,000,000.00	1.8366	CAD	12,856,200.00	27/08/2014
10	28/07/2014	Buy	EUR	13,500,000.00	0.7911	GBP	10,679,850.00	11/08/2014

Why is Java one of the problems?

- A simple CSV can grow by over 4 times…

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

- From roughly 70 bytes per line as CSV to around 328 in Java
- That means you get over 4 times less data when stored in Java
- Or need over 4 times more RAM, network capacity and disk

Java bloats your data

- You probably thought XML was bad imagine what happens when you take XML and bind it to Java!
- Anything you put into Java objects get horribly bloated in memory
- Effectively you are paying the price of memory and hardware abstraction

Fat Java Objects

- These fat Java Objects are a hardware vendor's wet dream
  - Think about it, Java came from Sun, it was free but they made money selling hardware, well they tried at least

- Fat objects need more memory, more CPU, more network capacity, more machines

- And everything just runs slower because you're busy collecting all the memory you're not using

- Java for programmers was like free shots for AA members

This isn't just Java

- Think this is just a Java problem?

- It's all the same, every time you create objects you're blasting huge holes all over your machine's RAM

- And someone's got to clean all the garbage up too!

In-memory caches

- If you use an in-memory cache then you're most likely suffering from the same problem…

- Many of them provide and use compression or "clever" memory optimisation
  - But this usually slows things down, introduces restrictions and only goes so far to resolve the issue

Consumer-driven Innovation

- Back in 1990 I had a recording of “Good Morning Viet Nam”, it took up an entire floppy disk
  - Just the sound of Robin William (RIP), not the movie!

- Today we can get the entire movie on a phone

- This is thanks to consumer-driven innovation in data compression and compaction

Improving on Java's Objects

- So how can we improve this…

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

```
public class ObjectTrade {
    private long id;
    private Date tradeDate;
    private String buySell;
    private String currency1;
    private BigDecimal amount1;
    private double exchangeRate;
    private String currency2;
    private BigDecimal amount2;
    private Date settlementDate;
}
```

- On the up-side it's fast to retrieve / search / query data

Improving on Java's Objects

```
public class ObjectTrade {
    private long id;
    private Date tradeDate;
    private String buySell;
    private String currency1;
    private BigDecimal amount1;
    private double exchangeRate;
    private String currency2;
    private BigDecimal amount2;
    private Date settlementDate;
}
```

- Every time we allocate a new ObjectTrade we create loads of new Objects

- When we de-serialize it's the same, it's very inefficient

Just store the original data?

- We could just store each row

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

```
public class StringTrade {
    private String row;
}
```

- But every time we wanted a date or amount we'd need to parse it and that would slow down analysis

- If the data was XML it would be even worse
  - We'd need a SAX (or other) parser every time

Just store the original data?

```
public class StringTrade {
    private String row;
}
```

- Allocation of new StringTrades are faster as we allocate just one Object
  - Well two actually one for StringTrade one for the String
- Serialization and De-Serialization are improved for the same reason
- But over all we lose out when we're accessing the data

Compression or Compaction?

- OK, everyone asks, why don't we just use compression?
- Well there are many reasons, mainly that it's slow, slow to compress and slow to de-compress, the better it is the slower it is
- Compression is the lazy person's tool, a good protocol or codec doesn't compress well, try compressing your MP3s or videos
- It has it's place but we're looking for more, we want compaction not compression, then we get performance too

Store it as binary

- What if we just use binary?

```
ID,TradeDate,BuySell,Currency1,Amount1,Exchange Rate,Currency2,Amount2,Settlement Date
1,21/07/2014,Buy,EUR,50000000.00,1.344,USD,67200000.00,28/07/2014
2,21/07/2014,Sell,USD,35000000.00,0.7441,EUR,26043500.00,20/08/2014
3,22/07/2014,Buy,GBP,7000000.00,172.99,JPY,1210930000,05/08/2014
```

```
public class ObjectTrade {
    private byte[] data;
}
```

- Just one object again so fast to allocate
- If we can encode the data in the binary then it's fast to query too
- And serialisation is just writing out the byte[]

Same API, just binary

- Classic getter and setter vs. binary implementation
- Identical API

```
@Override
public Date getTradeDate() {
    return tradeDate;
}

@Override
public void setTradeDate(Date tradeDate) {
    this.tradeDate = tradeDate;
}
```

```
@Override
public Date getTradeDate() {
    long date = wordFromBytesFromOffset(8);
    date *= 86_400_000L; // milliseconds in a day
    return new Date(date);
}

@Override
public void setTradeDate(Date tradeDate) {
    long date = tradeDate.getTime();
    date /= 86_400_000L; // milliseconds in a day

    data[8] = (byte)(date >>> 8);
    data[9] = (byte)(date);
}
```

Did I mention - The Same API

- This is a key point, we're changing the implementation not the API
- This means that Spring, in-memory caches and other tools work exactly as they did before
- Let's look at some code and a little demo of this…

```
@Override
public Date getTradeDate() {
    long date = wordFromBytesFromOffset(8);
    date *= 86_400_000L; // milliseconds in a day
    return new Date(date);
}

@Override
public void setTradeDate(Date tradeDate) {
    long date = tradeDate.getTime();
    date /= 86_400_000L; // milliseconds in a day

    data[8] = (byte)(date >>> 8);
    data[9] = (byte)(date);
}
```

Time to see some code

- A quick demo, I've created a Trade interface and two implementations, one "classic" and the other binary
    - We'll create a List of a few million trades (randomly but quite cleverly generated)
    - We'll run a quick Java 8 filter and sort on them
    - We'll serialize and de-serialize them to create a new List
- Finally for the binary version we'll write out the entire list via NIO and read it in again to a new List

The code

- The code in the following five slides was demonstrated during the conference but is included in these slides for your convenience

- If you'd like copies or more explanation please watch the live recording on the Spring web site (http://springone2gx.com/) or contact John at C24 (John dot Davies at C24 dot biz)

Interfaces and abstract base class…

```
package biz.c24.io.trade;

import java.math.BigDecimal;
import java.util.Date;

public interface ImmutableTrade {
    public long getId();
    public Date getTradeDate();
    public String getBuySell();
    public String getCurrency1();
    public BigDecimal getAmount1();
    public double getExchangeRate();
    public String getCurrency2();
    public BigDecimal getAmount2();
    public Date getSettlementDate();
}
```

```
package biz.c24.io.trade;

import java.math.BigDecimal;
import java.text.ParseException;
import java.util.Date;

public interface MutableTrade extends ImmutableTrade {
    public void setId(long id);
    public void setTradeDate(Date tradeDate);
    public void setBuySell(String buySell);
    public void setCurrency1(String currency1);
    public void setAmount1(BigDecimal amount1);
    public void setExchangeRate(double exchangeRate);
    public void setCurrency2(String currency2);
    public void setAmount2(BigDecimal amount2);
    public void setSettlementDate(Date settlementDate);

    public BasicTrade parse(String line) throws ParseException;
}
```

```
package biz.c24.io.trade;

import java.math.BigDecimal;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.TimeZone;

public abstract class BasicTrade implements MutableTrade {
    private static SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");

    @Override
    public String toString() {
        sdf.setTimeZone(TimeZone.getTimeZone("GMT"));

        StringBuffer sb = new StringBuffer();

        sb.append(getId()).append(',');
        sb.append(sdf.format(getTradeDate())).append(',');
        sb.append(getBuySell()).append(',');
        sb.append(getCurrency1()).append(',');
        sb.append(getAmount1().setScale(2, BigDecimal.ROUND_HALF_DOWN)).append(',');
        sb.append(getExchangeRate()).append(',');
        sb.append(getCurrency2()).append(',');
        sb.append(getAmount2().setScale(2, BigDecimal.ROUND_HALF_DOWN)).append(',');
        sb.append(sdf.format(getSettlementDate()));

        return sb.toString();
    }

    public BasicTrade parse( String line ) throws ParseException {
        sdf.setTimeZone(TimeZone.getTimeZone("GMT"));

        String[] fields = line.split(",");
        setId(Long.parseLong(fields[0]));
        setTradeDate(sdf.parse(fields[1]));
        setBuySell(fields[2]);
        setCurrency1(fields[3]);
        setAmount1(new BigDecimal(fields[4]));
        setExchangeRate(Double.parseDouble(fields[5]));
        setCurrency2(fields[6]);
        setAmount2(new BigDecimal(fields[7]));
        setSettlementDate(sdf.parse(fields[8]));

        return this;
    }
}
```

The “classic” Object Trade

```
package biz.c24.io.trade;

import java.io.*;
import java.math.BigDecimal;
import java.util.*;


public class ObjectTrade extends BasicTrade implements
Serializable {
    private long id;
    private Date tradeDate;
    private String buySell;
    private String currency1;
    private BigDecimal amount1;
    private double exchangeRate;
    private String currency2;
    private BigDecimal amount2;
    private Date settlementDate;

    @Override
    public long getId() {
        return id;
    }

    @Override
    public void setId(long id) {
        this.id = id;
    }

    @Override
    public Date getTradeDate() {
        return tradeDate;
    }

    @Override
    public void setTradeDate(Date tradeDate) {
        this.tradeDate = tradeDate;
    }

    @Override
    public String getBuySell() {
        return buySell;
    }

    @Override
    public void setBuySell(String buySell) {
        this.buySell = buySell;
    }

    @Override
    public String getCurrency1() {
        return currency1;
    }

    @Override
    public void setCurrency1(String currency1) {
        this.currency1 = currency1;
    }

    @Override
    public BigDecimal getAmount1() {
        return amount1;
    }

    @Override
    public void setAmount1(BigDecimal amount1) {
        this.amount1 = amount1;
    }

    @Override
    public double getExchangeRate() {
        return exchangeRate;
    }

    @Override
    public void setExchangeRate(double exchangeRate) {
        this.exchangeRate = exchangeRate;
    }

    @Override
    public String getCurrency2() {
        return currency2;
    }

    @Override
    public void setCurrency2(String currency2) {
        this.currency2 = currency2;
    }

    @Override
    public BigDecimal getAmount2() {
        return amount2;
    }

    @Override
    public void setAmount2(BigDecimal amount2) {
        this.amount2 = amount2;
    }

    @Override
    public Date getSettlementDate() {
        return settlementDate;
    }

    @Override
    public void setSettlementDate(Date settlementDate) {
        this.settlementDate = settlementDate;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof ObjectTrade)) return false;

        ObjectTrade that = (ObjectTrade) o;

        if (Double.compare(that.exchangeRate,
exchangeRate) != 0) return false;
        if (id != that.id) return false;
        if (amount1 != null ? !
amount1.equals(that.amount1) : that.amount1 != null)
return false;
        if (amount2 != null ? !
amount2.equals(that.amount2) : that.amount2 != null)
return false;
        if (buySell != null ? !
buySell.equals(that.buySell) : that.buySell != null)
return false;
        if (currency1 != null ? !
currency1.equals(that.currency1) : that.currency1 != null)
return false;
        if (currency2 != null ? !
currency2.equals(that.currency2) : that.currency2 != null)
return false;
        if (settlementDate != null ? !
settlementDate.equals(that.settlementDate) :
that.settlementDate != null)
            return false;
        if (tradeDate != null ? !
tradeDate.equals(that.tradeDate) : that.tradeDate != null)
return false;

        return true;
    }
}
```

The binary Trade

```
package biz.c24.io.trade;

import java.io.*;
import java.math.BigDecimal;
import java.util.*;


public class BinaryTrade extends BasicTrade implements Externalizable {

//     private long id;                 -->  8
//     private Date tradeDate;          -->  2
//     private String buySell;          -->  1
//     private String currency1;        -->  1
//     private BigDecimal amount1;      -->  8
//     private double exchangeRate;     -->  8
//     private String currency2;        -->  1
//     private BigDecimal amount2;      -->  8
//     private Date settlementDate;     -->  2
//     Total                            --> 39

    private byte[] data;

    private static String currencies[] = { "", "AUD", "CAD", "CHF", "EUR",
"GBP", "JPY", "USD" };

    public BinaryTrade() {
        data = new byte[39];
    }

    public byte[] getData() {
        return data;
    }

    public BinaryTrade( byte[] data ) {
        this.data = Arrays.copyOfRange(data, 0, 39);
    }

    @Override
    public void setId( long id) {
        long2BytesFromOffset(id, 0);
    }

    @Override
    public long getId() {
        return longFromBytesFromOffset(0);
    }

    @Override
    public Date getTradeDate() {
        long date = wordFromBytesFromOffset(8);
        date *= 86_400_000L; // milliseconds in a day
        return new Date(date);
    }

    @Override
    public void setTradeDate(Date tradeDate) {
        long date = tradeDate.getTime();
        date /= 86_400_000L; // milliseconds in a day

        data[8] = (byte)(date >>> 8);
        data[9] = (byte)(date);
    }

    @Override
    public String getBuySell() {
        return data[10] == 0 ? "Buy" : "Sell";
    }

    @Override
    public void setBuySell(String buySell) {
        // Not the safest but good for a test
        data[10] = buySell.charAt(0) == 'B' ? (byte) 0 : 1;
    }
```

```
    @Override
    public String getCurrency1() {
        return currencies[data[11]];
    }

    @Override
    public void setCurrency1(String currency1) {
        data[11] = 0;
        for(int i = 0; i < currencies.length; i++)
            if( currency1.equals(currencies[i])) {
                data[11] = (byte) i;
                return;
            }
    }

    @Override
    public BigDecimal getAmount1() {
        long value = longFromBytesFromOffset(12);
        BigDecimal bigDecimal = new BigDecimal(value);
        return bigDecimal.divide(new BigDecimal(100));
    }

    @Override
    public void setAmount1(BigDecimal amount1) {
        BigDecimal times100 = amount1.multiply(new BigDecimal(100));
        long value = times100.longValue();
        long2BytesFromOffset(value, 12);
    }

    @Override
    public double getExchangeRate() {

        long value = longFromBytesFromOffset(20);
        return Double.longBitsToDouble(value);
    }

    @Override
    public void setExchangeRate(double exchangeRate) {
        long value = Double.doubleToLongBits(exchangeRate);
        long2BytesFromOffset(value, 20);
    }

    @Override
    public String getCurrency2() {
        return currencies[data[28]];
    }

    @Override
    public void setCurrency2(String currency1) {
        data[28] = 0;
        for(int i = 0; i < currencies.length; i++)
            if( currency1.equals(currencies[i])) {
                data[28] = (byte) i;
                return;
            }
    }

    @Override
    public BigDecimal getAmount2() {
        long value = longFromBytesFromOffset(29);
        BigDecimal bigDecimal = new BigDecimal(value);
        return bigDecimal.divide(new BigDecimal(100));
    }

    @Override
    public void setAmount2(BigDecimal amount2) {
        BigDecimal times100 = amount2.multiply(new BigDecimal(100));
        long value = times100.longValue();
        long2BytesFromOffset(value, 29);
    }
```

```
    @Override
    public Date getSettlementDate() {
        long date = wordFromBytesFromOffset(37);
        date *= 86_400_000L; // milliseconds in a day
        return new Date(date);
    }

    @Override
    public void setSettlementDate(Date settlementDate) {
        long date = settlementDate.getTime();
        date /= 86_400_000L; // milliseconds in a day

        data[37] = (byte)(date >>> 8);
        data[38] = (byte)(date);
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.write(data);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        in.read(data);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof BinaryTrade)) return false;

        BinaryTrade that = (BinaryTrade) o;

        if (!Arrays.equals(data, that.data)) return false;

        return true;
    }

    private void long2BytesFromOffset(long value, int offset) {
        data[offset] = (byte)(value >>> 56);
        data[offset+1] = (byte)(value >>> 48);
        data[offset+2] = (byte)(value >>> 40);
        data[offset+3] = (byte)(value >>> 32);
        data[offset+4] = (byte)(value >>> 24);
        data[offset+5] = (byte)(value >>> 16);
        data[offset+6] = (byte)(value >>  8);
        data[offset+7] = (byte)(value);
    }

    private long longFromBytesFromOffset( int offset ) {
        long val = 0;
        for (int i = 0; i < 8; i++)
            val  = (val << 8) + (data[offset+i] & 0xff);
        return val;
    }

    private long wordFromBytesFromOffset( int offset ) {
        long val = 0;
        for (int i = 0; i < 2; i++)
            val  = (val << 8) + (data[offset+i] & 0xff);
        return val;
    }
}
```

“Classic” Serialization and query

```
private static byte[] serialize(Object obj) throws IOException {
    ByteArrayOutputStream b = new ByteArrayOutputStream();
    ObjectOutputStream o = new ObjectOutputStream(b);
    o.writeObject(obj);
    return b.toByteArray();
}

private static Object deserialize(byte[] bytes) throws IOException, ClassNotFoundException {
    ByteArrayInputStream b = new ByteArrayInputStream(bytes);
    ObjectInputStream o = new ObjectInputStream(b);
    return o.readObject();
}

private static void testSerialization(List<ImmutableTrade> immutableTrades) throws IOException, ClassNotFoundException {
    long start;
    System.out.println("Testing Serialization...");
    List<ImmutableTrade> trades2 = new ArrayList<>(ARRAY_SIZE);
    byte[] bytes = null;

    start = System.nanoTime();
    for( ImmutableTrade immutableTrade : immutableTrades) {
        bytes = serialize(immutableTrade);
        trades2.add((ImmutableTrade) deserialize(bytes));
    }
    System.out.printf("Time to serialize/deserialize %,.1f million: %,.2f seconds%n", ARRAY_SIZE / 1e6, (System.nanoTime() - start) / 1e9);
    System.out.printf("Serialized size: %,d%n", bytes.length);

    checkResults(immutableTrades, trades2);
    trades2.clear();
}
```

```
private static void runSimpleQuery(List<ImmutableTrade> immutableTrades) {
    long start = System.nanoTime();

    System.out.println("Running a filter and sort on the trades (3 times)");
    for( int i = 0; i < 3; i++) {
        start = System.nanoTime();
        immutableTrades.stream()
                .filter(t -> t.getCurrency1().matches("GBP") && t.getCurrency2().matches("USD") && t.getBuySell().matches("Buy"))
                .sorted(Comparator.comparing(ImmutableTrade::getExchangeRate))
                .limit(l)
                .forEach(System.out::println);
    }
    System.out.printf("Last query time = %.3f seconds%n%n", (System.nanoTime() - start) / 1e9);
}
```

Customised (batched) Serialization

```
private static List<ImmutableTrade> readFromFile(String filename) throws IOException {
    long start;
    FileChannel ch;
    ByteBuffer bb;
    List<ImmutableTrade> trades2 = new ArrayList<>(ARRAY_SIZE);

    System.out.println("Reading the binary data from the raw file...");
    start = System.nanoTime();
    FileInputStream fis = new FileInputStream(new File(filename));
    ch = fis.getChannel();
    bb = ByteBuffer.allocateDirect(400_000); // 10k at a time
    int nRead;
    while ((nRead = ch.read(bb)) != -1) {
        if (nRead == 0)
            continue;
        bb.position(0);
        bb.limit(nRead);
        while (bb.hasRemaining()) {
            byte[] barray = new byte[40];      // This would probably work better with a ring-buffer to reuse the memory
            bb.get(barray, 0, 40);
            trades2.add(new BinaryTrade(barray));
        }
        bb.clear();
    }
    System.out.printf("Time to read %,.1f million: %,.2f seconds%n%n", ARRAY_SIZE / 1e6, (System.nanoTime() - start) / 1e9);
    return trades2;
}

private static void writeToFile(List<ImmutableTrade> immutableTrades, String filename) throws IOException {
    long start;

    System.out.println("\nWriting binary data to raw file...");
    start = System.nanoTime();

    FileDescriptor fd = new RandomAccessFile(new File(filename), "rw").getFD();
    FileOutputStream fos = new FileOutputStream(fd);
    FileChannel ch = fos.getChannel();
    ByteBuffer bb = ByteBuffer.allocateDirect(400_000); // 10k at a time
    final byte[] zeros = new byte[1];

    for (ImmutableTrade immutableTrade : immutableTrades) {
        byte[] data = ((BinaryTrade) immutableTrade).getData();
        bb.put(data);
        bb.put(zeros);  // Padding

        if( ! bb.hasRemaining() ) {
            bb.flip();
            while(bb.hasRemaining()) {
                ch.write(bb);
            }
            bb.clear();
        }
    }
    fos.close();
    System.out.printf("Time to write %,.1f million: %,.2f seconds%n%n", ARRAY_SIZE / 1e6, (System.nanoTime() - start) / 1e9);
}
```

How does it perform?

- On our little test here are a few results…

	Classic Java version	Binary Java version	Improvement
Bytes used	328	39	840%
Bytes per instance	328	48	683%
Serialization size	668	85	786%
Custom Serialization*	668	40	1,670%

NB: All timing are single thread on a Retina MacBook Pro and should just be indicative only

How does it perform?

- And performance? (single thread on Java 8)

	Classic Java version	*Binary Java version*	*Improvement*
Time to query	*323nS*	*219nS*	*147%*
Time to Serialize / Deserialize	*41.1µS*	*4.17µS*	*988%*
*Time to read raw data from disk**		*44nS**	*N/A*

NB: All timing are single thread on a Retina MacBook Pro and should just be indicative only

Let me explain the *

- Since the entire Trade object (and all its elements) are in a byte[] we can read/write the byte[] to/from disk or network
- We don’t need to use Serialization and the data is already available to query via the getters
- So we can read 40 MB containing 1 million trades into memory and it’s immediately query-able since we don’t need to parse it
- This is a HUGE gain in performance - several hundred times!

Batching & Mechanical sympathy

- You probably noticed that the actual byte[] size was 39 but Java used 48 bytes per instance

- By batching and creating batch classes that handle the large numbers of instances not as a List or Array but more tightly we can get further improvements in memory and performance

- 1 million messages or 39 bytes should be exactly 39,000,000 bytes
  - As things get more complex the message size varies and we often have to compromise with a larger batch quanta
  - We usually have to stick to 8 byte chunks too

Batching & Mechanical sympathy

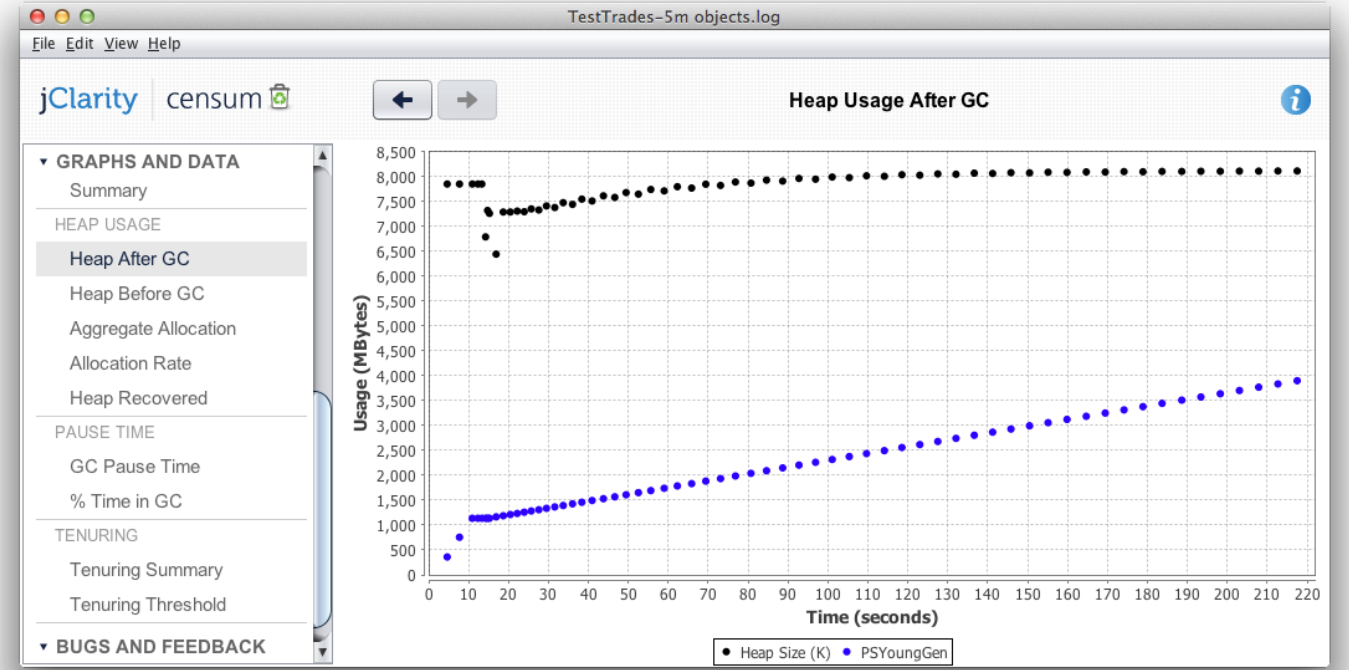
- Knowing how your disk (HDD or SSD) works, knowing how you network works means we can make further optimisations

- A typical network packet it about 1.5k in size, if we can avoid going over that size we see considerable network performance improvements

- What Java set out to do was to abstract the programmer from the hardware, the memory, CPU architecture and network, that abstraction has cost us dearly

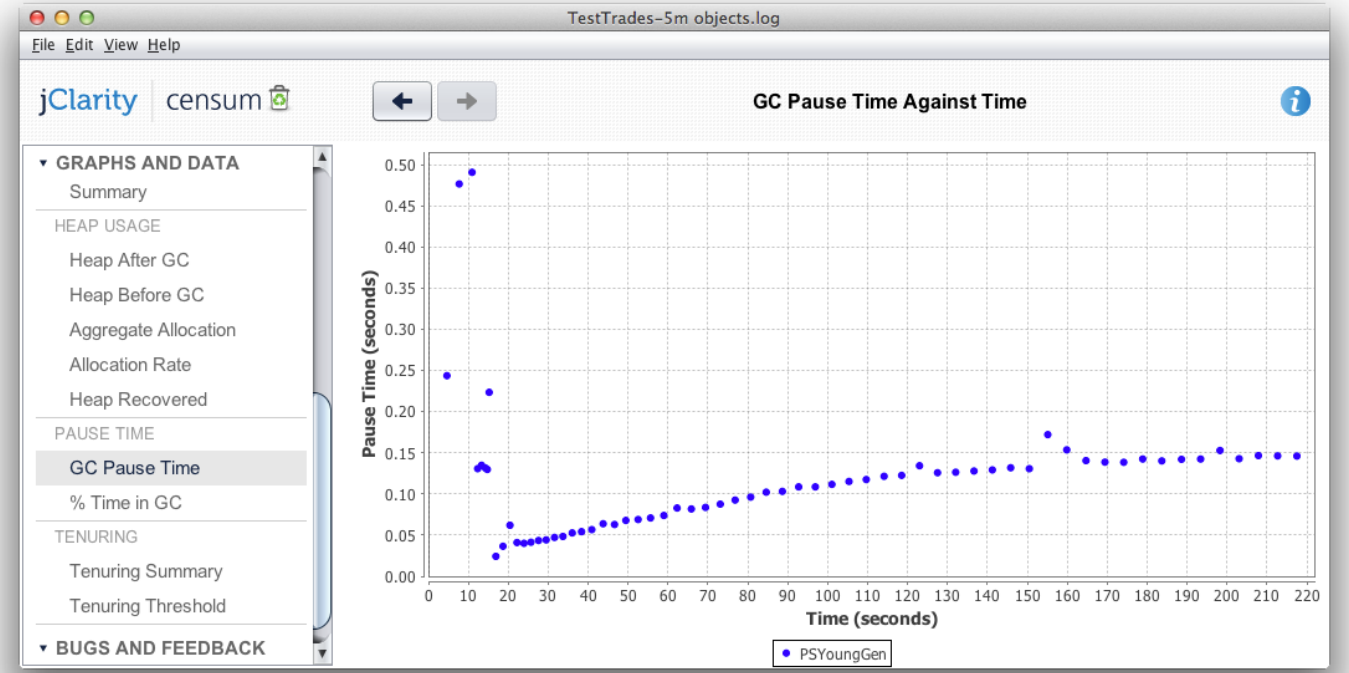
Memory heap usage (Object version)

- 200 Seconds for serialization and 4GB



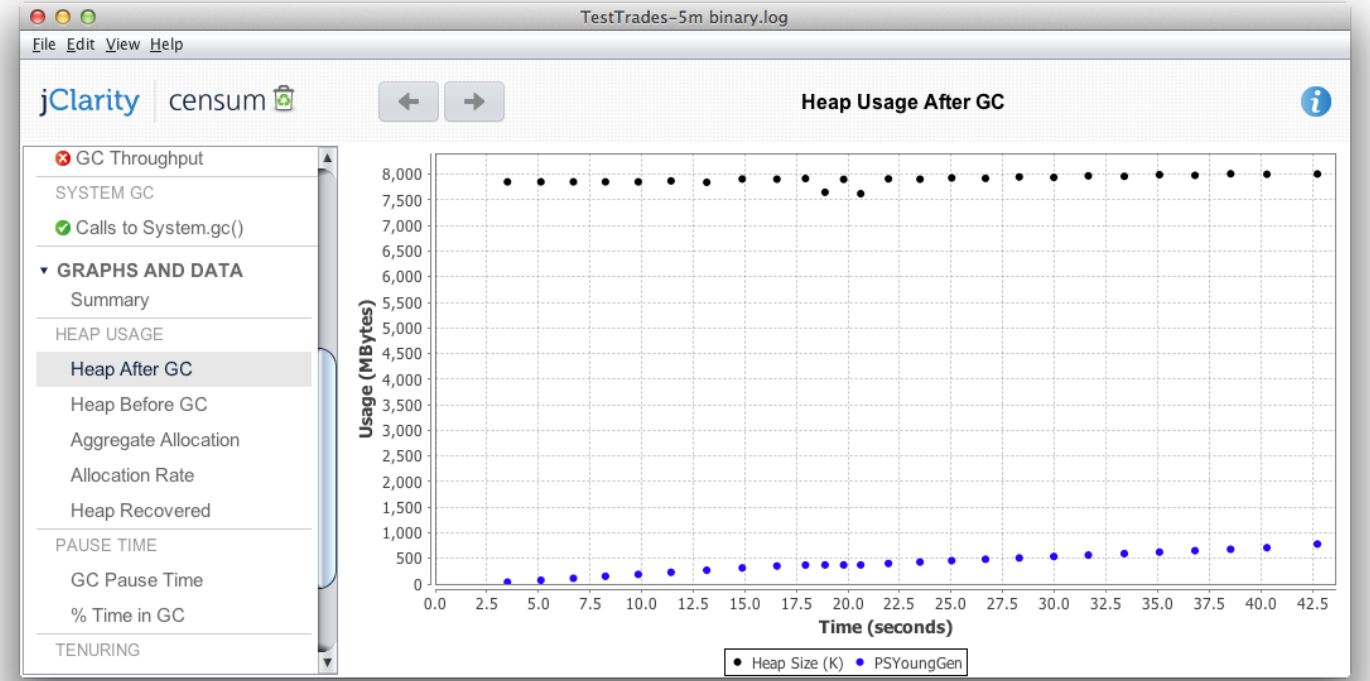
Memory heap usage (Object version)

- GC pause up to 500mS, averaging around 150mS



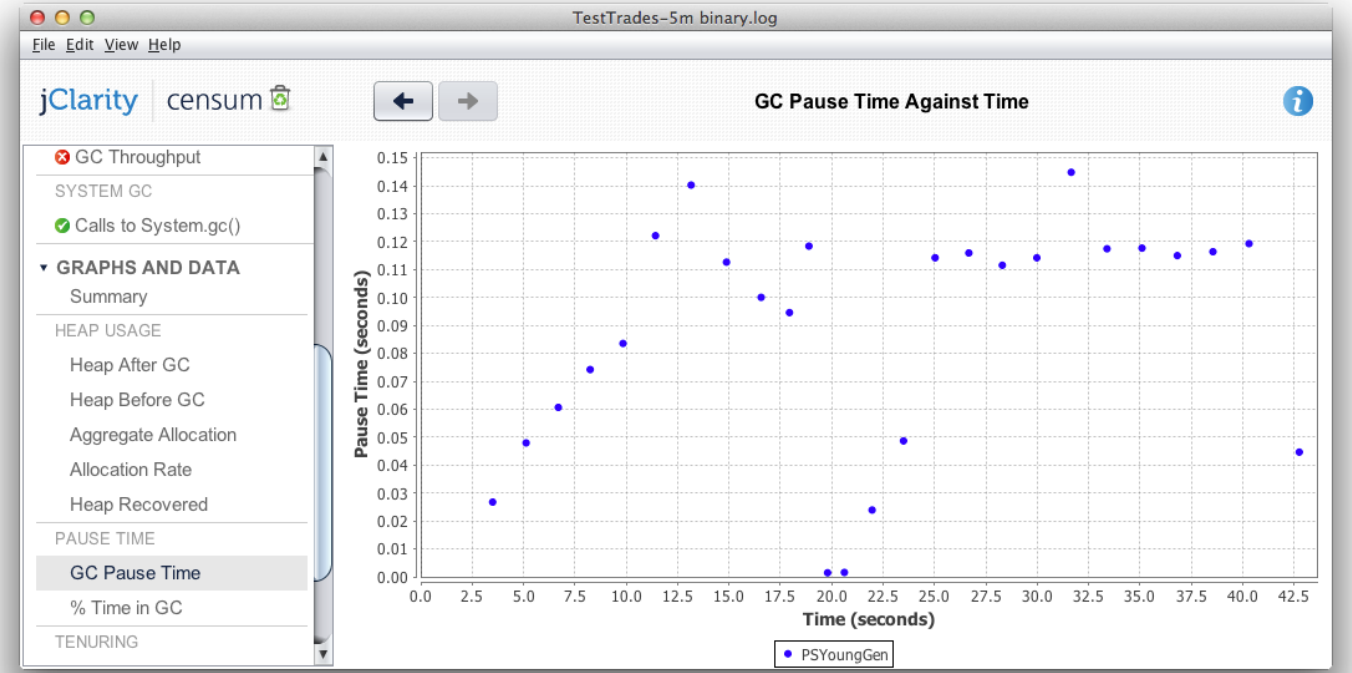
Memory heap usage (Binary version)

- 40 Seconds for serialization and 700MB



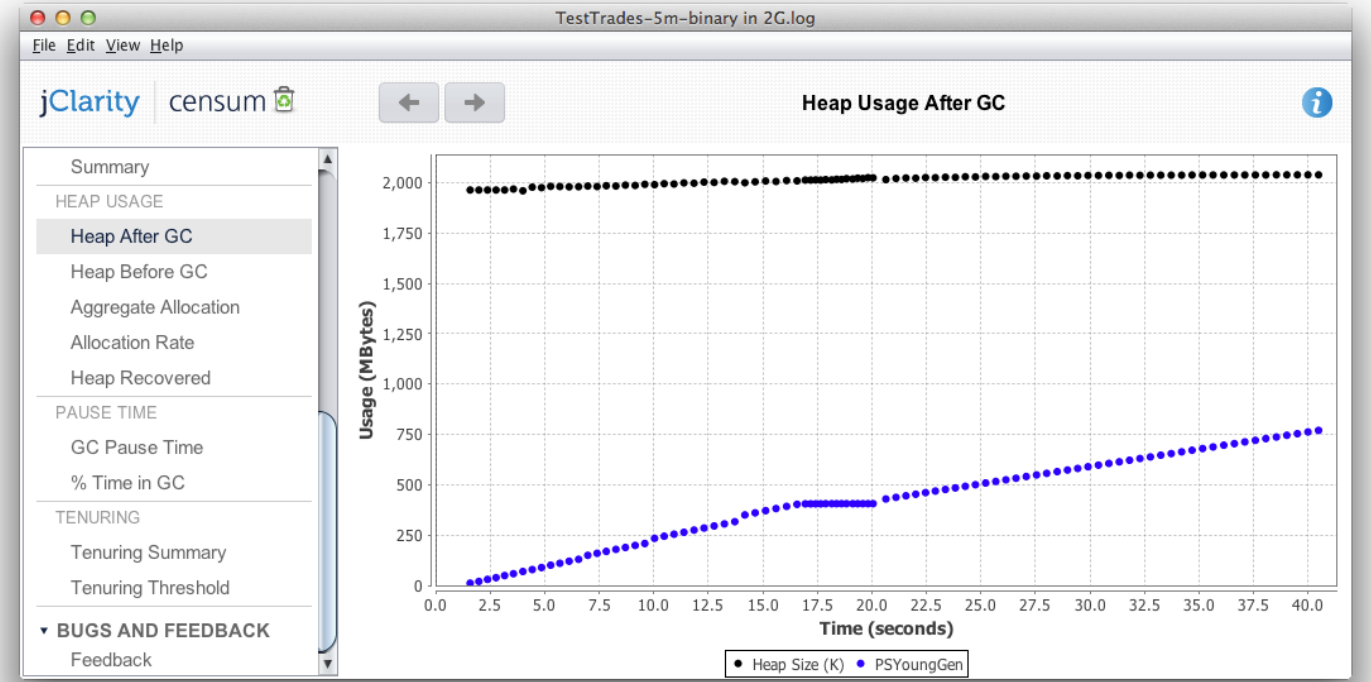
Memory heap usage (Binary version)

- GC pause up to 150mS, averaging around 100mS but a lot less



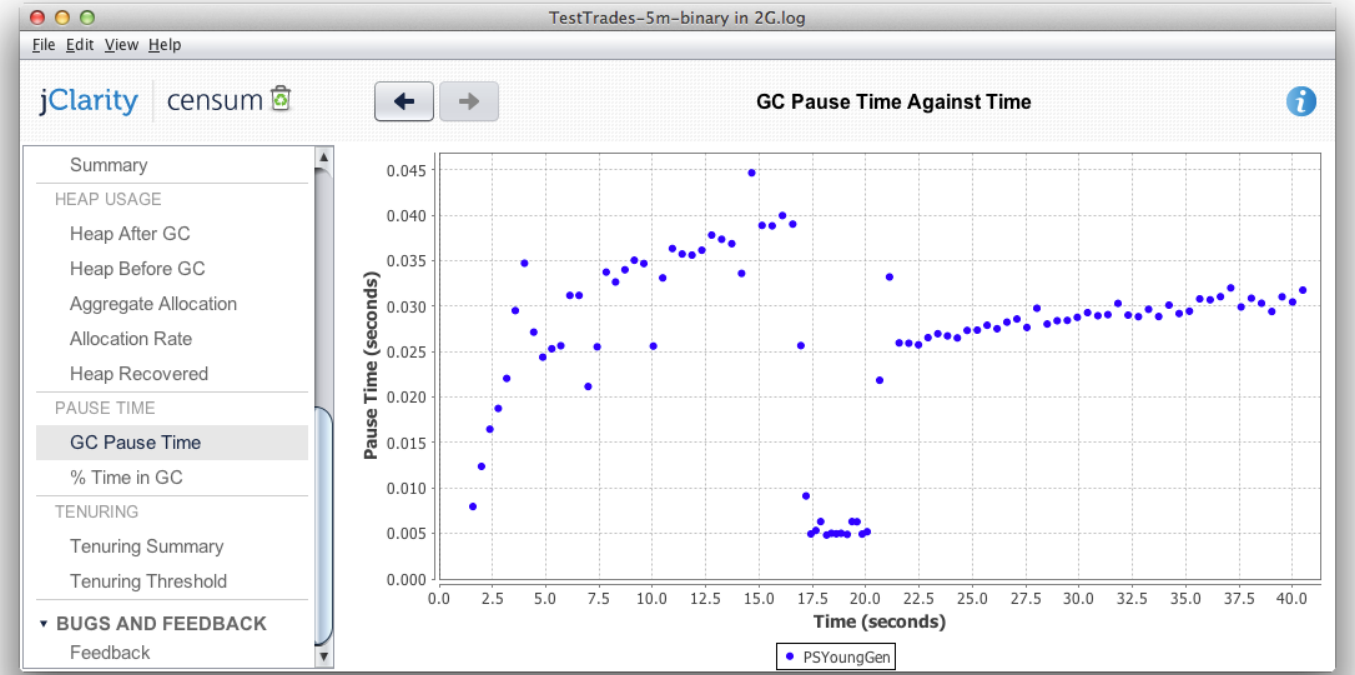
Memory heap usage (Binary version)

- The same but in a 2GB heap



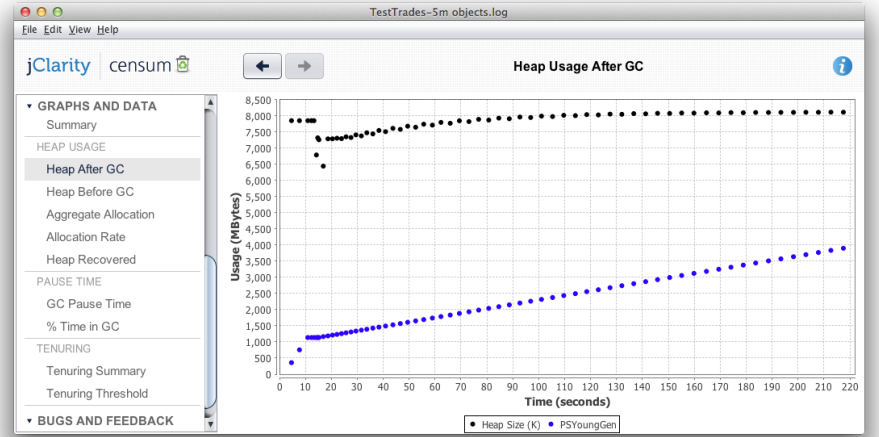
Memory heap usage (Binary version)

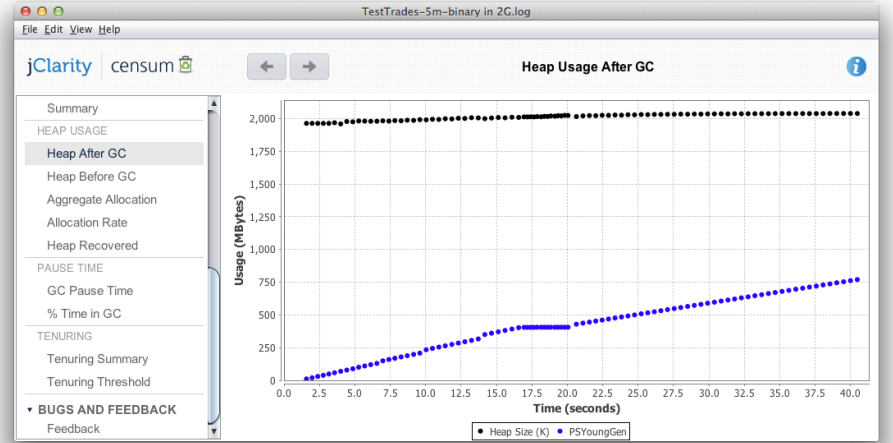
- GC pause up to 45mS, averaging around 30mS



Comparing…

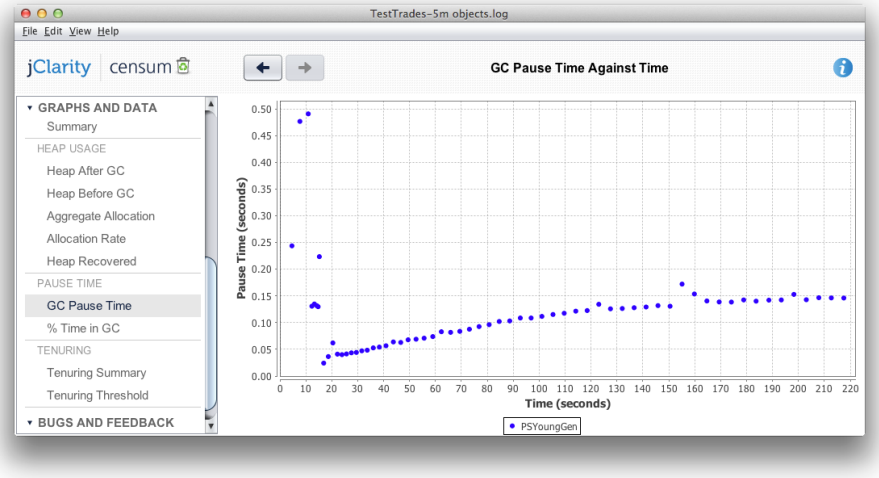
- While the shapes look the same you can clearly see the differences on both axis
  - The Object version is a lot slower
  - And the Object version uses significantly more memory

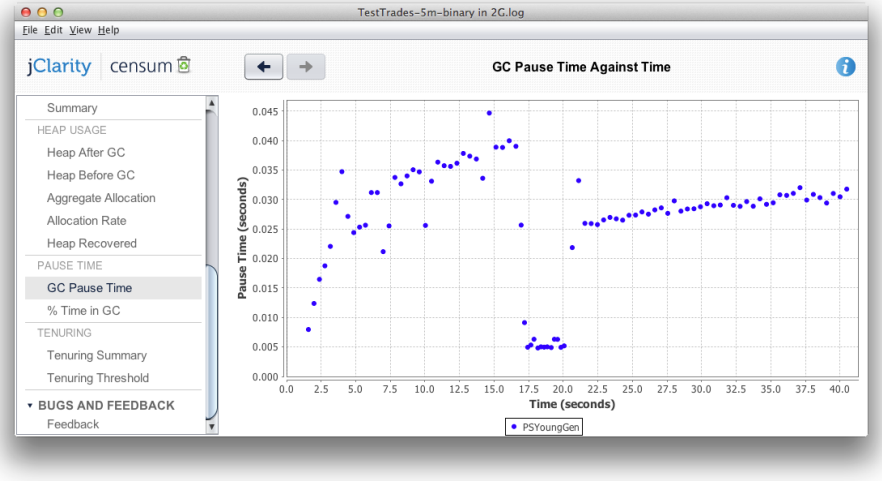




Comparing…

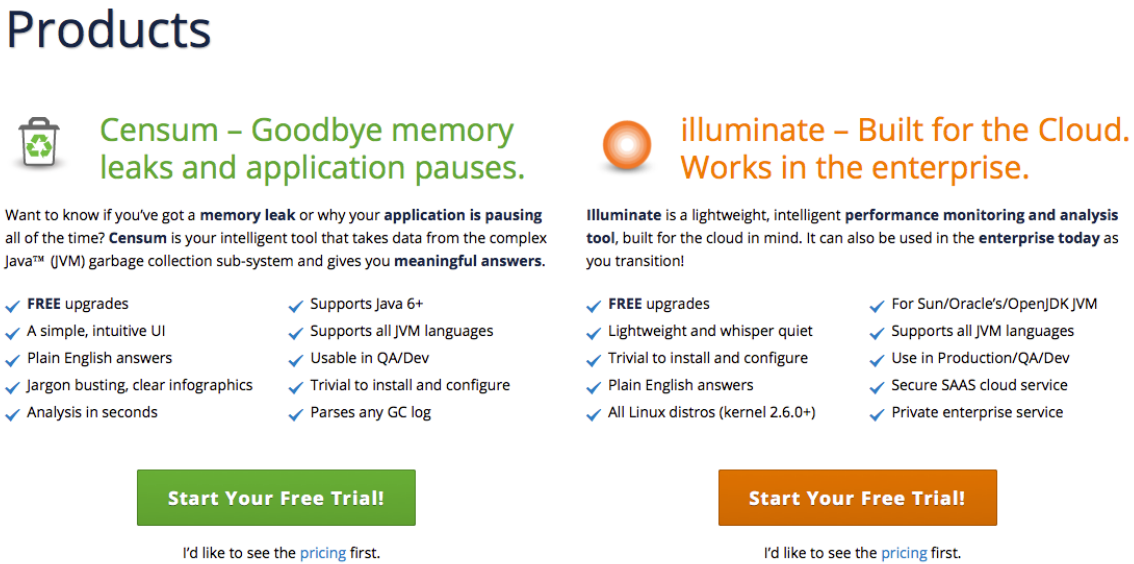
- Again look at the y-axis (GC Pause time)
  - The top of the right-hand graph (binary) is lower then the first line of the left-hand graph (objects)





jClarity

- The memory usage graphs were created using jClarity's Censum
  - Many thanks to Martijn and especially Kirk for their help



Beyond the CSV file…

- It worked for a simple CSV how about more complex models like XML or JSON?
- Once we have a mapping of types to binary and code we can extend this to any type of model
- But it gets a lot more complex as we have optional elements, repeating elements and a vast array of different types
- But we did it!

Turbo-charge Spring Integration

- All these numbers apply directly to Spring since we're just using the standard Java API to our objects

- `<filter input-channel="filter-message-channel" output- channel="process-message-channel" ref="payload" method="isValid"/>`

- `<filter input-channel="filter-message-channel" output- channel="process-message-channel" expression="payload.currency1 == GBP"/>`

SpEL compilation - Spring 4.1

- As we get into nano-second performance every little optimisation counts
- Spring 4.1 introduces SpEL compilation so expressions can be compiled “on the fly”
- The results are impressive a single expression goes from (typically) 1.5µs to 60ns, an improvement of 25 times
- This means we can do several expressions and still achieve 1 million per second throughput

SpEL compilation - Spring 4.1

- Without these improvements in Spring we couldn't have gotten anywhere near the 1 million messages per second we're seeing with RTI

- The SpEL compilation was as a direct result of customer requirements and field engineers pushing the limits
  - Coded by Andy Clement

- SpEL compilation is now available in the latest Spring release

Code Generation

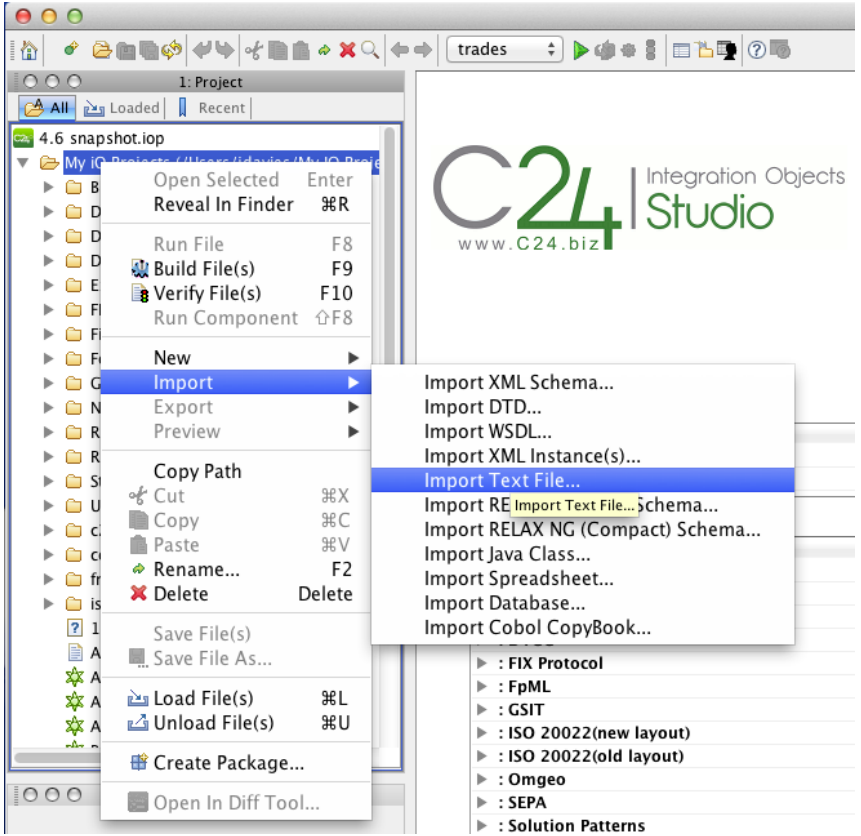
- The first implementations of binary were written by hand
  - The performance improvements were so dramatic we decided to take it to the next level and write a Java-Binding engine for binary

- We can now generate binary implementations for almost any model, from…
  - Java code (via reflection), XML Schema (including complex standards like FpML and ISO-20022), JSON, CSVs
  - Models can be created graphically too

- We use C24's binary Java Binding tool, iO Studio (**www.C24.biz**)

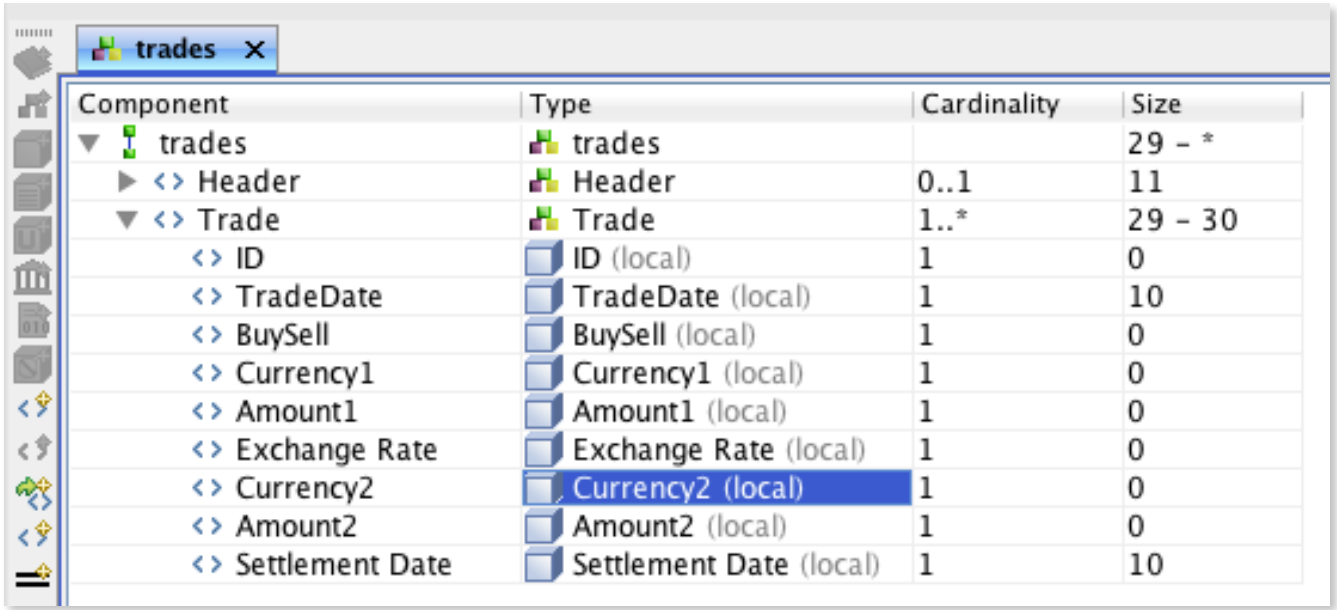
Code Generation - Importing the model

- Importing the model…
  - XML Schema
  - DTD & Instant
  - WSDL
  - Text files
  - Java
  - Spreadsheet
  - Database
  - even COBOL copybook



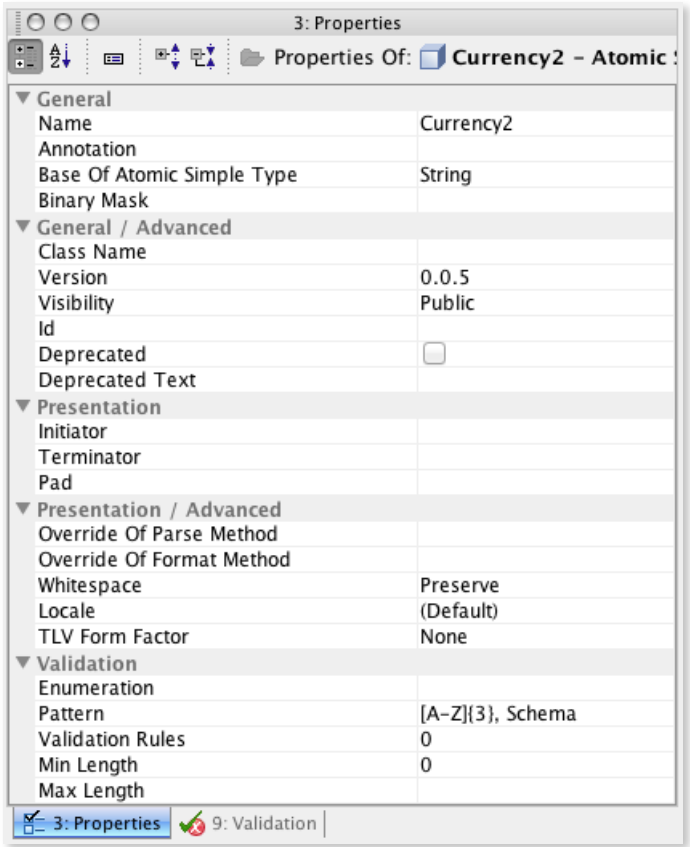
Code Generation - The model

- Once imported the model can be “tweaked”



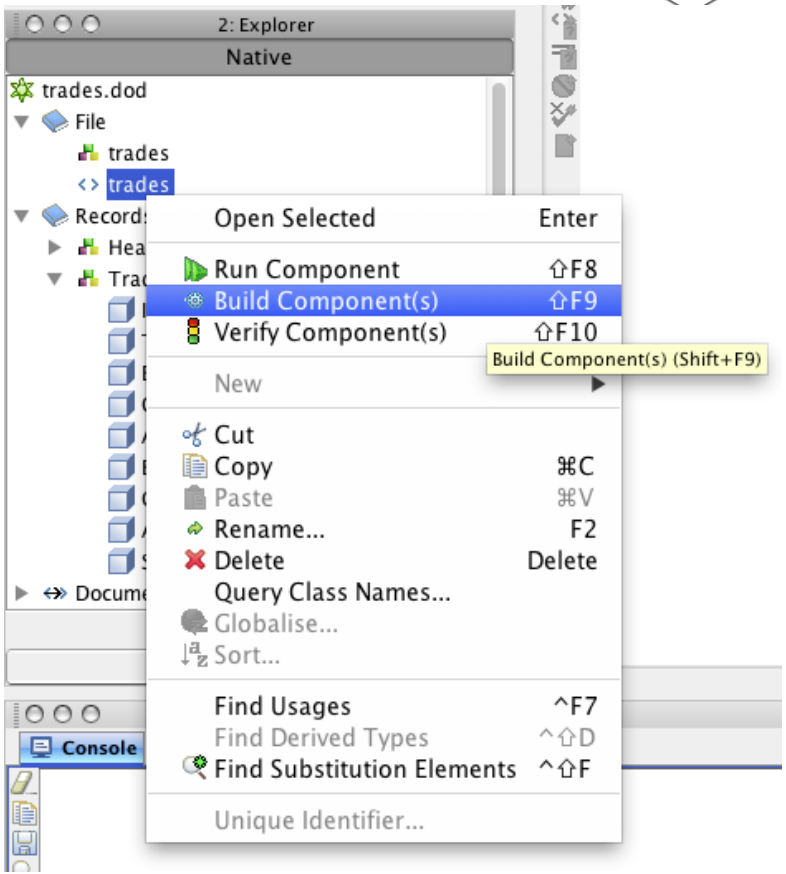
Code Generation - Element properties

- Every aspect of each element can be changed
  - Name
  - Type
  - Binary implementation
  - Rules, regex patterns
  - Initiators, terminators, padding
- We can even create “virtual” methods for fields that are calculated



Code Generation - Deploying the code

- Once the model is complete it can be built
- This will create two versions
  - The full (classic) version
  - The binary version (if the option is set)
- The is MDA so any changes are made to the model which is then re-deployed



C24 generated code

- The C24 generated code for the Trade example is actually smaller, it averages around 33 bytes
- It uses run-length encoding so we can represent the 64 bit long by as little as a single byte for small numbers
- This means the size of the binary message varies slightly but for more complex models/message these small optimisations result in considerable improvements

Back to Big Data

- With a small Spring Integration workflow we can
  - Read in the data
  - Filter, route, enrich (as required)
  - Write to GemFire / Spring XD (or whatever takes your fancy)

- The Spring config does not change but we see a dramatic (10-50 plus times) increase in performance

- As well as a dramatic (10-50 times) increase in what can be stored in memory

Pivotal's "RTI"

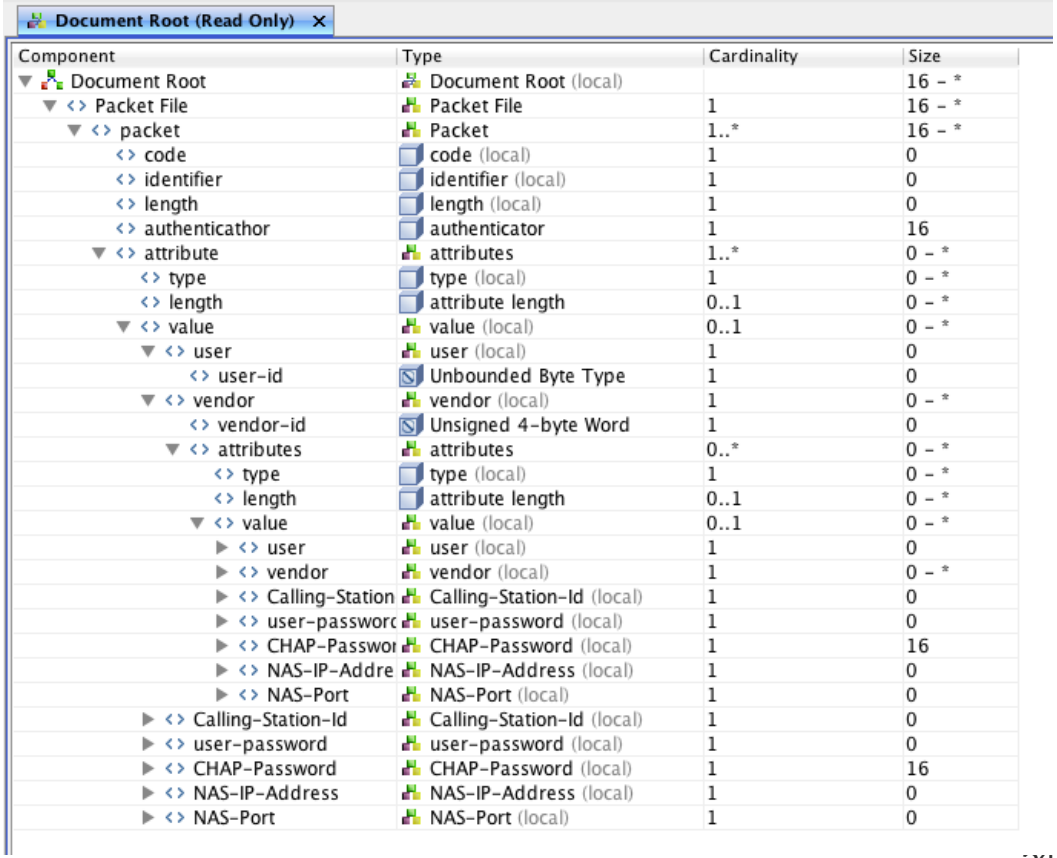
- RTI is "Real Time Intelligence", it uses all these tools and concepts to achieve incredible real-time performance

- A major customer is Vodafone

- The binary 2G, 3G & 4G messages averaging around 100k/sec are handled in binary all the way through
  - C24, Spring, Spring Integration, Batch, SpEL compilation
  - Reactor, Netty, GemFire etc.

Telco data models

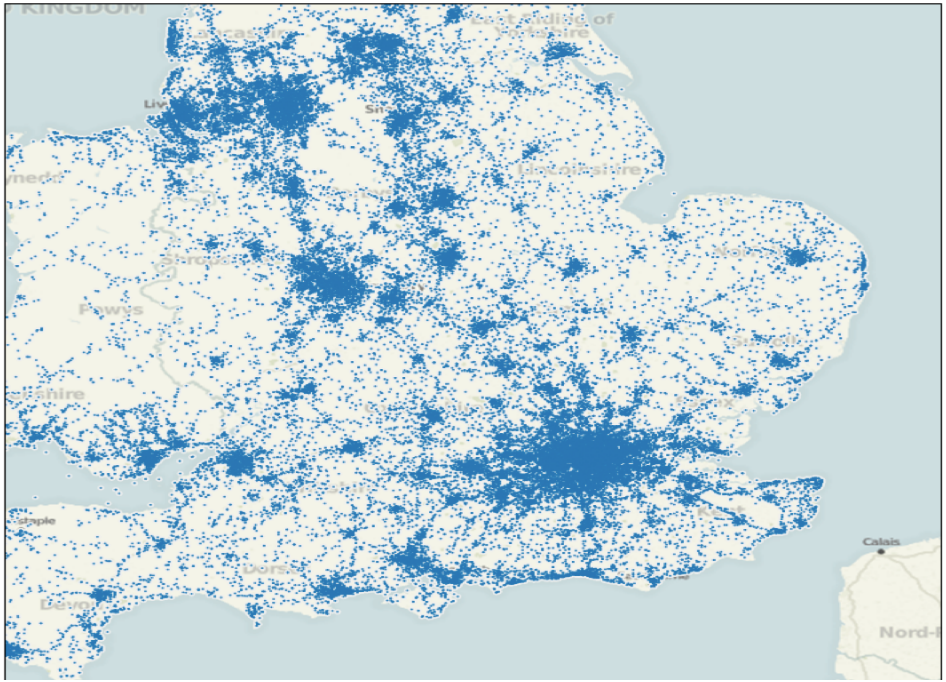
- They are 100% binary
- So we modelled them and generated the binary code
- This was a LOT of work but still less than hand-coding each standard



Component	Type	Cardinality	Size
Document Root	Document Root (local)		16 - *
Packet File	Packet File	1	16 - *
packet	Packet	1..*	16 - *
code	code (local)	1	0
identifier	identifier (local)	1	0
length	length (local)	1	0
authenticathor	authenticator	1	16
attribute	attributes	1..*	0 - *
type	type (local)	1	0 - *
length	attribute length	0..1	0 - *
value	value (local)	0..1	0 - *
user	user (local)	1	0
user-id	Unbounded Byte Type	1	0
vendor	vendor (local)	1	0 - *
vendor-id	Unsigned 4-byte Word	1	0
attributes	attributes	0..*	0 - *
type	type (local)	1	0 - *
length	attribute length	0..1	0 - *
value	value (local)	0..1	0 - *
user	user (local)	1	0
vendor	vendor (local)	1	0 - *
Calling-Station	Calling-Station-Id (local)	1	0
user-passwor	user-password (local)	1	0
CHAP-Passwo	CHAP-Password (local)	1	16
NAS-IP-Addre	NAS-IP-Address (local)	1	0
NAS-Port	NAS-Port (local)	1	0
Calling-Station-Id	Calling-Station-Id (local)	1	0
user-password	user-password (local)	1	0
CHAP-Password	CHAP-Password (local)	1	16
NAS-IP-Address	NAS-IP-Address (local)	1	0
NAS-Port	NAS-Port (local)	1	0

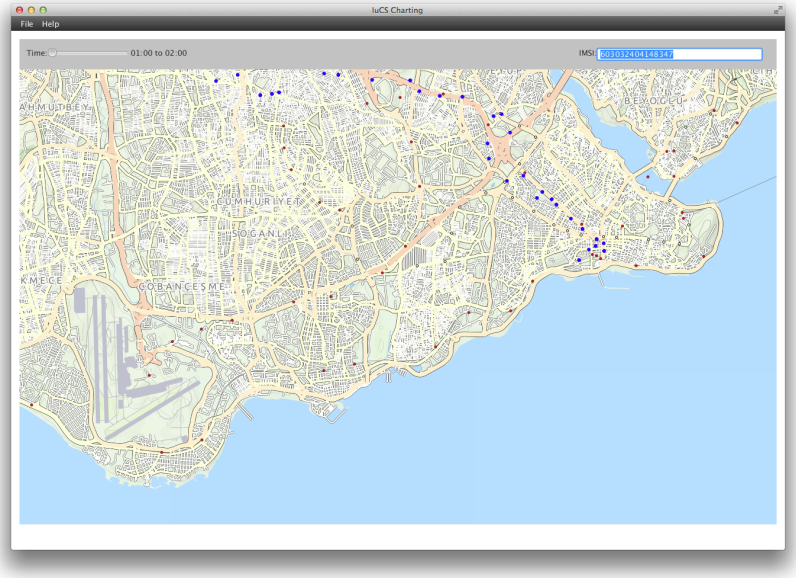
Network performance

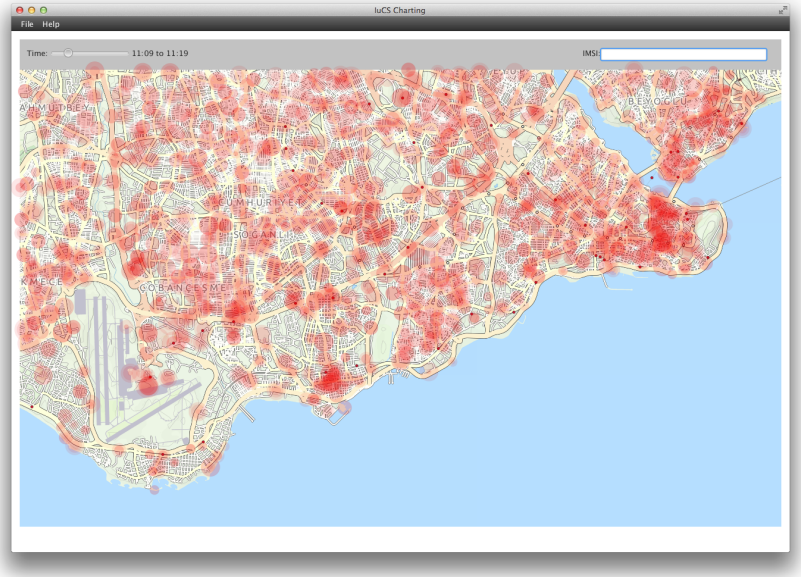
- It’s no use knowing what went wrong afterwards
- With real-time we can see issues as they happen
- The user can be pro-active
- Imagine AT&T using it
  - They’d know you’re call’s been cut off in real time!



Testing can be fun…

- Istanbul in just 10 minutes, 21 million events processed at 137,594/sec on a single thread on a my laptop





NB: These are personal tests and GUI's not Pivotal's

So it works

- Key points from the slides…

- If you want performance, scalability and ease of maintenance then you need…
  - Using binary data instead of objects
  - Start coding like we used to in the 90s
  - Move to Spring 4.1

- Or just buy much bigger, much faster machines, more RAM, bigger networks and more DnD programmers

Thank you…

- For more white-papers on binary codecs for Java

  http://SDO.C24.biz

- Twitter: @jtdavies