June 2012

LOKAD

# WHITEPAPER: PROCESSING A LARGE RETAIL NETWORK ON A SMARTPHONE

www.lokad.com | Joannes Vermorel, Founder

Long before 'big data' became the technology buzzword of 2012, retail networks have been among the pioneers in dealing with the large amounts of data that is produced by their supply chain and their point of sale systems. Recognizing the richness and immense value of their data, heavy IT infrastructure investments have, in many cases, been made.

A compact binary representation with about 5 bytes per receipt line on average, that is 60% less than GZip over TSV. It requires also 7x less CPU to write receipts, and 11x less CPU to read receipts.

However, to date, the limitations and cost of the required infrastructure has left the reality far behind ambition and promise. This is particularly true for the richest retail data source, which also **dwarfs all others in size: receipts generated by point of sale systems**. Collecting and processing receipts of hundreds or even thousands of stores has remained a daunting, and very expensive, task.

Each receipt contains the list of items purchased, including information such as day, time, price, discounts etc. This data for example provides the basis for inventory and supply chain optimization, on-shelf availability improvements, customer loyalty marketing, consumer analysis and staffing optimization. In some cases, retail networks are even selling this information to their suppliers in multi-million yearly contracts.

As of 2012, from the vantage point of Lokad, we observe that many retailers have invested millions of Euros or Dollars for example in high-end servers and/or server clusters and bear annual operating costs in the hundreds of thousands in order to collect and process receipts. Yet, their agility is often limited, data access remains costly and as a consequence the exploitation of this data remains far below its potential.

**About the Author:** A recognized expert on cloud computing and statistical learning, **Joannès Vermorel** holds a Master of Science degree from the 'École Normale Supérieure de Paris' (ENS Ulm). In 2010, Joannès won the worldwide Microsoft Windows Azure Partner of the Year Award in recognition for his pioneering work in the cloud.

### *What about running a large retail network on a smartphone instead?*

While this question is provocative both from a technical and commercial point of view, we explain in this whitepaper how fundamental operations such as collecting and processing receipts for retail networks of up to 1000 stores can be done on a smartphone.

The source code used by Lokad to produce the results exposed in this white paper has been made available as open source under a very liberal license (BSD) on GitHub[1].

When Lokad migrated toward the cloud in 2010, Windows Azure gave us access to an unprecedented amount of processing power (roughly 100x more!). Interestingly however, while we learned to 'harness' the suddenly abundant computing resources provided by the cloud, a rather unexpected skill emerged in parallel: **we also learned to be 100x more efficient when processing retail data**.

The combination of these two radical evolutions has made a reality what would have been dismissed as science-fiction only a few years ago. By sharing some insights on big data for retail, we hope to further fuel the advances in retail data exploitation.

## DIMENSIONING THE RETAIL NETWORK

Lokad has been working on large datasets, obtained from several of the largest retail networks worldwide. First, let's define what we consider to be a typical large (grocery) retail network:

- 1000 stores.
- 15 billion Euros of turnover.
- 1.5€ per item on average.
- 37 items purchased per receipt on average (i.e. 50€ shopping carts).

Those assumptions imply collecting, processing and persisting about 10 billion receipt lines per year.

Here, we will assume that each **receipt header** hold the following information:

- A date, with second precision.
- An optional loyalty card number.
- A store identifier.

Then, each **receipt line** contains:

- GTIN (Global Trade Identification Number), a barcode identifier, up to 14 digits.
- The purchased quantity.
- The unit price.
- An optional discount.
- The VAT rate for the item.

[1] See https://github.com/Lokad/lokad-receiptstream

For the sake of simplicity, we have restricted the scope to a relatively narrow number of fields. However, our experience indicates that those fields cover the overwhelming majority of applications that could be envisioned while leveraging receipts.

Also, the methodology described below could be extended with extra fields without fundamentally changing the primary claim: *a smartphone should be enough*.

## DIMENSIONING THE SMARTPHONE

From a practical viewpoint, it does not make sense to even try to process receipts over a smartphone. However, we believe that looking at **a smartphone as a piece of dead cheap unit of computing resources** is enlightening.

In the following, we assume that the smartphone has:

- 1Ghz CPU
- 512MB of RAM
- 64GB of solid state drive[2] with I/O at 30MB/s

Compared to nowadays high-end servers, the smartphone computing power is thin, and yet, from an absolute perspective, it's enormous: the text of the Wikipedia fits several times on such a device.

## WRITING AND READING RECEIPTS

The most elementary operation to be done with receipts consists of writing them to disk as they are streamed in; and then to make receipts available for read later on.

In order to do that, we propose *ReceiptStream* a tiny piece of software, weighting **less than 500 lines of code**.

To perform those operations, we observe that retailers nowadays quasi exclusively rely on SQL databases. However, we will compare *ReceiptStream* to two alternative approaches:

- Relational database (SQL).
- Flat text file compressed with GZip.

Indeed, while retail systems in production are primarily relying on SQL databases, GZIP-compressed flat text files are surprisingly efficient, and consequently worth being benchmarked against *ReceiptStream*.

In order to measure the respective performance of those approaches, we believe there are three key metrics:

[2] As of 2012-05-31, a 64GB MicroSD card from SanDisk is sold $64.99 on Amazon.com

1. The number of bytes consumed on average per receipt line for persistence.
2. The number of receipts that can be written per second.
3. The number of receipts that can be read per second.

# RECEIPTSTREAM, A DATA COMPRESSION ALGORITHM TAILORED FOR RECEIPTS

The central idea behind *ReceiptStream* is a compact binary representation of receipts. This binary format relies on simple statistical properties of the receipts to achieve both a very high compression ratio, but also a very low CPU consumption.

With 37 lines per receipt in this study, receipt lines vastly outweigh receipt headers. Thus, *ReceiptStream* is not even doing anything special for receipt headers: it relies on a plain binary representation. Subtleties lie in the representation of *receipt lines*.

## Combining 7-bits integers with lookups

A receipt line includes 5 tokens; each one has specific statistical properties:

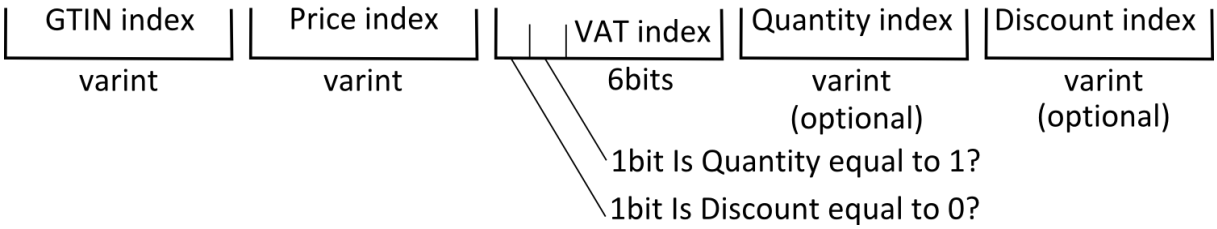
- GTIN: 12 to 14 digits, however, there are hardly more than 100k distinct barcodes being sold on any given day, even for the largest retail networks.
- The purchased quantity, which frequently equals 1. Our measurements indicate that for grocery retail, about 85% of the receipt lines have a quantity of 1.
- Unit prices typically follow a *price grid*: 49cts is much more frequent than 51cts. Hence, frequent prices can be leveraged for compression.
- The optional discount follows a pattern quasi-identical to prices; however the most frequent discount value is *zero*.
- VAT rates have a very limited diversity (dozens at most).

Representing GTINs with 10+ bytes is a waste of storage, because the total number of distinct GTINs in existence is only a tiny fraction of the potential number of combinations.

Thus, instead, *ReceiptStream* replaces GTINs with 7-bit encoded integers, called *varint*, i.e. integers of varying size. The principle behind those varints is simple: the $8^{th}$ bit indicates if the next byte should be read as part of the current integer. As a result, values between 0 and 127 need only a single byte to be represented.

Intuitively, the varint format assumes that small values are much more frequent than large values. *ReceiptStream* leverages this property by making sure that, indeed, small values are much more frequent than large ones.

The illustration below describes our binary representation of a receipt line:




In particular, the GTIN and the price are replaced by two indexes written as varints. Naturally, this representation implies that *ReceiptStream* maintains two lookups[3] mapping between indexes toward the actual GTIN and price values.

Then, the 3rd block has a size of exactly 1 byte. It contains 2 Boolean flags indicating if *quantity is not one* and if *discount is not zero*. If the quantity equals one then the 4th block is omitted. Similarly if the discount equals zero, then the 5th block is omitted. This pattern leaves room for 64 distinct VAT rates which is vastly sufficient in practice.

Quantities are represented by varints directly (quantities larger than 127 are extremely infrequent), while VAT rates, and discounts get represented through indexes, and thus involve 2 more lookups. In practice, the size of the GTIN lookup dwarfs the size of the 3 other lookups put together. Yet, the GTIN lookup remains below 50MB in memory.

The content of a receipt is represented by the concatenation of all the receipt lines. Before writing down the first receipt line, *ReceiptStream* writes down a varint indicating how many lines should be read as part of the current receipt.

## RoughSort as an approximate bubble sort

In order to achieve a very compact representation of the receipt lines with the binary representation outlined in the previous section, we need to allocate the most frequent GTINs to the lower index values.

A similar process needs to be applied to other fields; however for the sake of concision, we only describe the GTIN case in the following: *ReceiptStream* treats prices, discounts and VAT rates in an identical manner.

The naïve way to perform this index allocation consists in a 2-pass process over the data:

1. Read all receipts, and build the lookup by sorting GTINs by decreasing frequency.
2. Read all receipts again, and leverage the lookup established in (1) to encode.

However, any multi-pass algorithm would be rather impractical for production purposes. *ReceiptStream* is, as the name suggest, an *online* algorithm that processes the receipts in a strict streaming manner (1-pass only).

[3] See http://en.wikipedia.org/wiki/Lookup_table

It is possible to keep a sorted dictionary of the GTINs (sorted against their respective frequency) and to update this dictionary after reading each individual receipt. However, we found out that such a process is relatively CPU intensive.

Instead, *ReceiptStream* relies on an *approximate* sorting data structure that we name a *RoughSort*. It is an approximate variant of the Bubble Sort[4]. An array containing all the respective frequencies (aka number of observations so far) of the GTIN is maintained. When a new GTIN of index A is observed, its observation count is increased of 1. If the observation count of A is greater than the observation count of the GTIN of index B = A / 2, then, the two GTINs swaps their indexes A and B.

The *RoughSort* maintains only an approximately sorted array, however:

- Each update (of the table that counts observations) triggers at most a single swap.
- The frequency of swaps decrease rapidly as the number of receipts increases.

As a result, the CPU consumption of *ReceiptStream* at write-time is maintained extremely low in comparison of maintaining a more refined binary tree *ala* GZip.

## Alternating lookup updates and receipts

At write-time (resp. read-time) *ReceiptStream* processes one receipt a time, however, before writing (resp. reading) the content of the actual receipts, *ReceiptStream* writes (resp. reads) *lookup updates* that contain, as the name suggest, changes that need to be applied to the lookup tables *before* writing (resp. reading) the actual receipt.

There are two types of lookup update:

- **Adding a new value to one of the lookup tables**. By convention the value is added at the end of the lookup. In practice, *ReceiptStream* uses dynamic arrays[5] to keep this process efficient.
- **Swapping two indexes in one of the lookup tables**. Contrary to GZip, swaps are explicit in the data stream. By following this pattern, at read-time, *ReceiptStream* does not need to maintain any hash table[6] or binary tree, but only plain lookups.

For the sake of concision, we will not give here the full detail of the header binary representation, i.e. the section that contains all the lookup changes. We suggest referring to the code on GitHub instead, as there are no clever tricks being used, only straightforward binary formatting.

[4] See http://en.wikipedia.org/wiki/Bubble_sort
[5] See http://en.wikipedia.org/wiki/Dynamic_array
[6] See http://en.wikipedia.org/wiki/Hash_table

# EXPERIMENTAL RESULTS

The table below has been obtained on an Intel Core i5 at 2.4 GHz using only a single CPU processing over a million receipts (real world data).

| | **Compression** | **Write (per sec)** | **Read (per sec)** |
|---|---|---|---|
| ReceiptStream | 5.21 byte / line | 33478 receipts<br>6356 kB | 64984 receipts<br>12338 kB |
| TSV+Gzip | 12.83 byte / line | 4845 receipts<br>2264 kB | 5714 receipts<br>2264 kB |
| Microsoft SQL | 83.04 byte / line | *not tested* | *not tested* |

We observe that:

- *ReceiptStream* data footprint is only 40% of TSV+Gzip.
- Both *ReceiptStream* and TSV+Gzip are **CPU bound**: the bottleneck is CPU, not I/O.
- *ReceiptStream* writes receipts about 7x faster than TSV+GZip.
- *ReceiptStream* reads receipts about 11x faster than TSV+GZip.
- The data footprint of the SQL representation is 16x worse than *ReceiptStream*.

## More about SQL

In order to collect the SQL numbers, we used the *plain canonical* schema for receipts, with two tables for respectively *receipts* and *receipt_lines*. We did not even try to produce accurate I/O figures for SQL, however, our experience indicates that it is even worse in comparison to the alternative approaches than it is for raw storage.

Furthermore, with SQL, one needs to **store indexes in memory**. Here, we cannot avoid an index on the *receipt_lines* table, which alone represent about 25% of the overall space being consumed by the SQL database. Denormalizing the database is possible, but it further increases storage inefficiencies.

In practice, processing receipts through SQL requires machines with a large amount of RAM, and while your mileage may vary, we believe that typical I/O performance for such SQL setups are about 100x times lower than those of *ReceiptStream*.

*This conclusion is not specific of Microsoft SQL Server, but applies to all relational databases.*

While it is possible to fine-tune the SQL schema to improve performance (ex: storing heavily compressed blobs in a table), we believe that such tuning defeats the purpose of SQL in the first place.

## Back to our smartphone

*ReceiptStream* results indicate that storing 1 year of receipts for the 1000 stores retail

network introduced here above takes about 50GB of storage. Also, a 1 GHz smartphone CPU would write more than 5000 receipts per second (resp. read more than 10000 receipts per second); assuming that smartphone CPU performance is no more than 6x slower than a 2.4 Ghz notebook CPU. This implies that:

- Writing 1 year worth of receipts takes less than 2h10min.
- Reading 1 year worth of receipts take less than 1h05min.

*Clearly*, **a smartphone-sized processing capacity is sufficient to deal with the data of an entire very large retail network**.

## LOOKING AHEAD

While being capable of processing receipts of a very large retail network over a smartphone might look as a tough challenge: **it is not**. In particular, *ReceiptStream* is not even really optimized: our primary goal was to keep it extremely *simple*, not extremely efficient. With less than 500 lines of code, *ReceiptStream* is a *tiny* piece of software. We believe that **achieving a further 2x compression factor while halving the CPU consumption is possible**, but maybe not worth the effort.

Then, it's not because basic retail data processing can happen over a smartphone, that massive cloud computing resources do not come handy. By migrating toward Windows Azure back in 2009, at Lokad, we discovered that combining both *lean* data infrastructure (with a typical 100x speed-up over SQL) and cloud auto-scaling capabilities (with a 100x scale-out factor, available within minutes) was the best approach to get the most out big data in retail.

We believe that **processing all receipts of the largest retail networks is a non-issue**. We believe that vendors selling tools in 6 digit figures to do it are just doing it wrong and/or doing a disservice to their clients.

## ABOUT LOKAD

Lokad is a technology company focusing on big data analytics software for retail, wholesale and eCommerce. Software solutions include inventory optimization, loyalty card data analysis and out-of-shelf monitoring. The company is the winner of the 2010 Microsoft Windows Azure Worldwide Partner of the Year Award.

For large retail networks, we also routinely perform **big data consulting** helping local teams to get the most of their data while maintaining IT budget in control. Learn more about our services at http://www.lokad.com/big-data-consulting.ashx