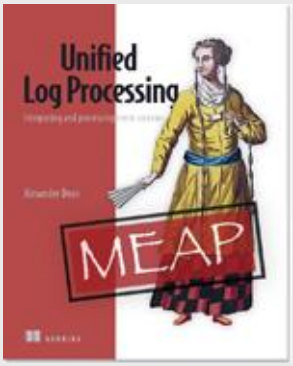


Stream processing in e-commerce

By Alexander Dean

In this article, in order to illustrate cases for stream processing, we use a practical example based around online shopping.

This article is excerpted from Unified Log Processing by Alexander Dean. Save 39% on Unified Log Processing with code 15dzamia at manning.com.

Imagine that we work for a sells-everything e-commerce website, let's call it Nile. The management team at Nile wants the company to become much more dynamic and responsive: Nile's analysts should have access to up-to-the-minute sales data, and Nile's systems should react to customer behavior in a timely fashion. As we will see, we can meet their requirements by implementing a unified log and some *multiple event processing* applications.

Identifying our key events

Online shoppers browse products on the Nile website, sometimes adding products to their shopping cart, and sometimes then going on to buy those products through the online checkout. Of course there are plenty of other things that visitors can do on the website, but Nile's executives and analysts care most about this "viewing through to buying" workflow. Figure 1 shows a typical shopper (albeit with somewhat eclectic shopping habits) going through this workflow.

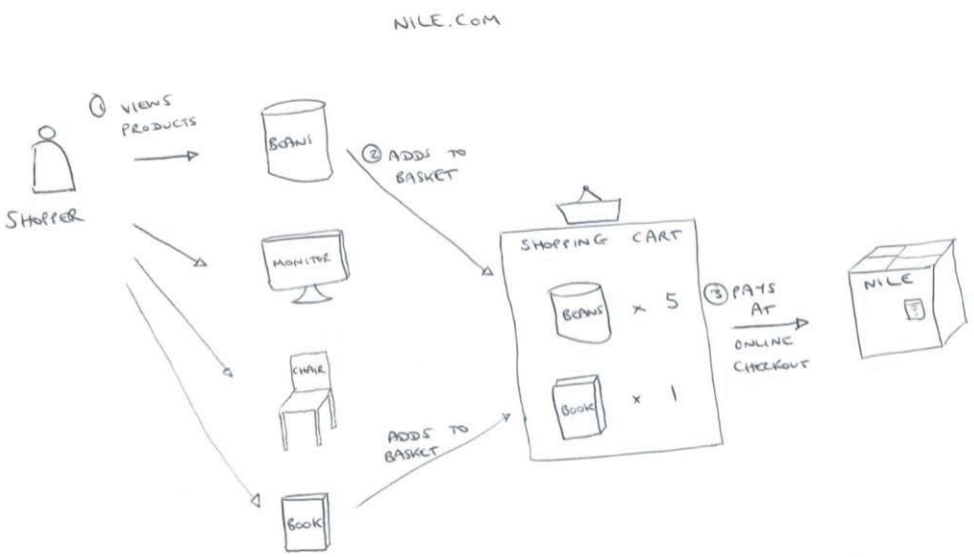


Figure 1. A shopper views four different products on the Nile website, before adding two of those products to her shopping cart. Finally she checks out and pays for those items.

Drawing on our standard definition of an event as *Subject verb object*, we can identify three discrete events in this "viewing through to buying" workflow:

- *Shopper views product* – which occurs every time the shopper views a product, whether on a product's dedicated "detail" page, or on a general "catalog" page which happens to include the product
- *Shopper adds item to cart* – whenever the shopper adds one of those products to his or her shopping basket. A product is added to the basket with a quantity of one or more attached
- *Shopper places order* – where the shopper checks out, paying for the items in his or her shopping basket

To avoid complicating our lives later in this chapter we have kept this workflow deliberately simple, steering clear of more complex interactions, such as the shopper adjusting the quantity of an item in his shopping basket, or removing an item from his basket at checkout. But no matter: the above three events represent the essence of the shopping experience at Nile.

What management wants

As part of their goal of creating a more dynamic and responsive business, the management team at Nile has identified three key business requirements centered on the "viewing through to buying" workflow.

HOURLY SALES FIGURES

Nile's analysts want to know the total value of orders placed on the website, aggregated to an hourly level. All Nile's sales are in dollars, and all of Nile's business s reporting is done using UTC (Coordinated Universal Time), which makes things simpler. For the current hour, the Nile team is happy to see a rolling figure for the sales so far in that hour. Putting all this together, the data should look something like Figure 2.

Figure 2. The Nile management team wants a report showing sales figures as in this figure. This report was probably captured at around 2:15 in the morning of September 6 2014, based on the sales reported so far for that hour.

Hourly sales figures would be calculated simply by summing the values of all *Shopper placed order* events which occurred within the given hour.

LIFETIME PRODUCT LOOK-TO-BOOK RATIOS

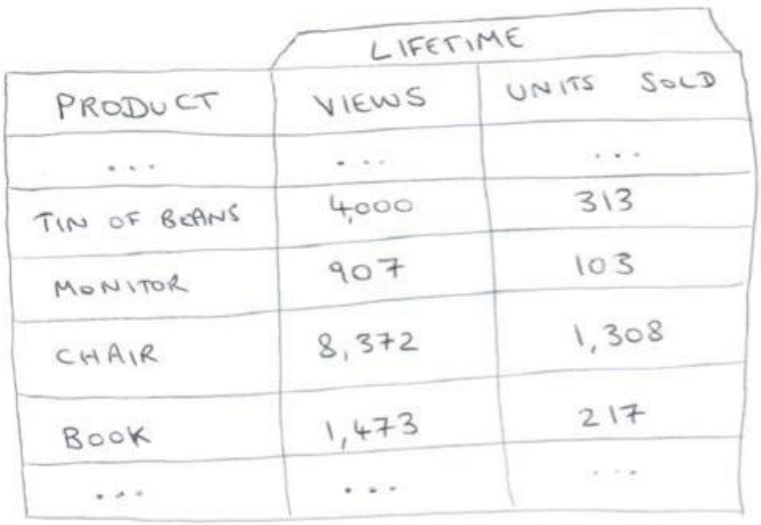
Look-to-book is hotel industry terminology for comparing how many people *look* at a hotel room to how many people actually *book* a hotel room. At Nile, the team is interested in comparing the total number of views that a product receives online to how many units of that product actually get sold. This ratio is helpful to understand which products are converting better online and is thus easy for the team to act on:

- *Strong look-to-book ratio* – if a product is shifting a lot of units for comparatively few product views, then it is converting well and Nile's marketing team should consider

driving more traffic to this product page

- *Weak look-to-book ratio* – if a product is failing to turn lots of product views into sales, then Nile's merchandising team should look to improve the product's pricing, description, photos and so on

Nile is only interested in each product's look-to-book ratio across that product's entire lifetime, which should keep things simple for us. If we store each product's lifetime views and lifetime units sold, then we can easily calculate the ratio whenever we want, by dividing one number by the other. Figure 3 shows some example rows for the product look-to-book report.



	LIFETIME	
PRODUCT	VIEWS	UNITS SOLD
...	...	...
TIN OF BEANS	4,000	313
MONITOR	907	103
CHAIR	8,372	1,308
BOOK	1,473	217
...	...	...

Figure 3 showing the lifetime views and units sold for the four products introduced in Figure 1. By storing lifetime views and units sold on a per product basis, we can quickly calculate any product's lifetime look-tobook ratio when needed.

ABANDONED SHOPPING CART DETECTION

Now we come to Nile's third and most challenging requirement: identifying and responding to abandoned shopping carts. A shopping cart is defined as abandoned when a shopper adds products to their shopping cart but doesn't continue through to checkout. For online retailers, it is worthwhile contacting shoppers who have abandoned their carts and asking if they would like to complete their orders. Timing is everything here: it's important to identify and react to abandoned shopping carts quickly, but not so quickly that a shopper feels pestered or rushed;

UK handbag company Radley released a study which showed that 30 minutes after abandonment is the optimal time to get in touch.[1]

Typically an online retailer will respond to an abandoned shopping cart by emailing the shopper, or by showing the shopper "re-targeting" ads on other websites, but we don't need to worry about the exact response mechanism. The important thing is to define a new event, *Shopper abandons cart*, and generate one of these new events whenever we detect an abandoned cart.

How do we detect an abandoned shopping cart? For the purposes of this article, let's use a simple algorithm:

- Our shopper adds a product to cart
- Derive a Shopper abandons cart event if 30 minutes passes without either: ○ Our shopper adding any further products to cart, or: ○ Our shopper placing an order
- If our shopper adds a further product to cart during the 30 minutes, restart the timer
- If our shopper places an order during the 30 minutes, clear the timer

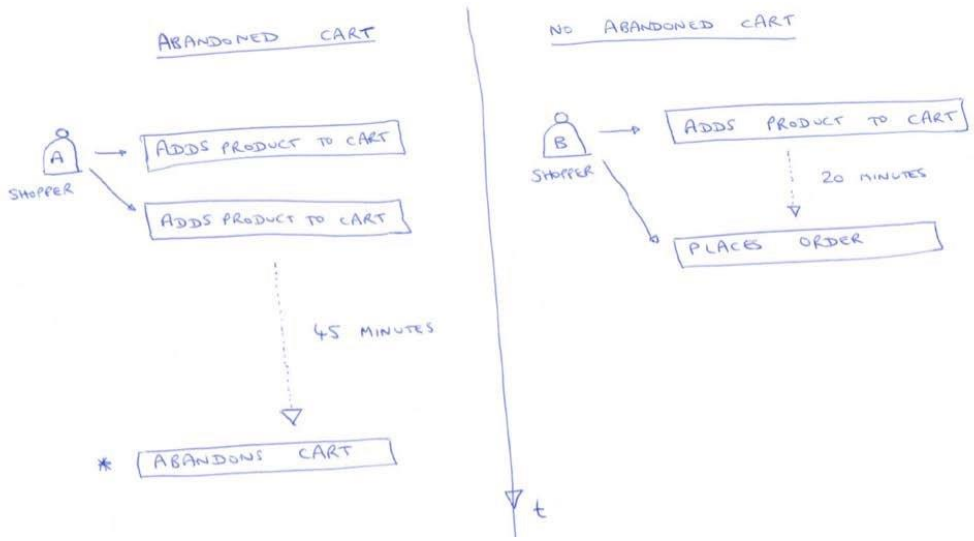


Figure 4. On the left-hand side, Shopper A has added two products to her shopping cart, and then 45 minutes have passed without any further activity, so we can derive a *Shopper abandons cart* event. On the right-hand side, Shopper B has added a product to his shopping cart and checked out within 20

minutes, not abandoning his cart.

Two examples of applying this algorithm are set out in Figure 4. This algorithm is not a particularly sophisticated approach, but it should help Nile get started tackling their abandoned shopping carts problem and it can always be refined further later.

So this completes the set of three business requirements that Nile has around the shopper's "viewing through to buying" workflow. In the next section, we will map these requirements back to our unified log toolkit, and introduce a new tool: stateful stream processing.

Unified log, e-commerce style

We are in luck: Nile wants to introduce Apache Kafka as a unified log across the business, and use the "viewing through to buying" requirements set out above as a first test case for the new architecture. It's up to us to map these requirements onto our unified log.

We can define an initial event stream to record the events generated by shoppers. Let's call this stream nile-rawevents, and in Figure 5 you can see our three different types of event feeding into the stream. To make this a little more realistic, I've made a distinction based on how the events are captured:

- *Browser-generated events* – the *Shopper views product* and *Shopper adds item to basket* events occur in the user's browser. Typically there would be some JavaScript code[2] to send the event to an HTTP-based event collector, which would in turn be tasked with writing the event to the unified log
- *Server-side events* – a valid *Shopper places order* event is only confirmed server-side, once the payment processing has completed. It is the responsibility of the web server to write this event to Kafka

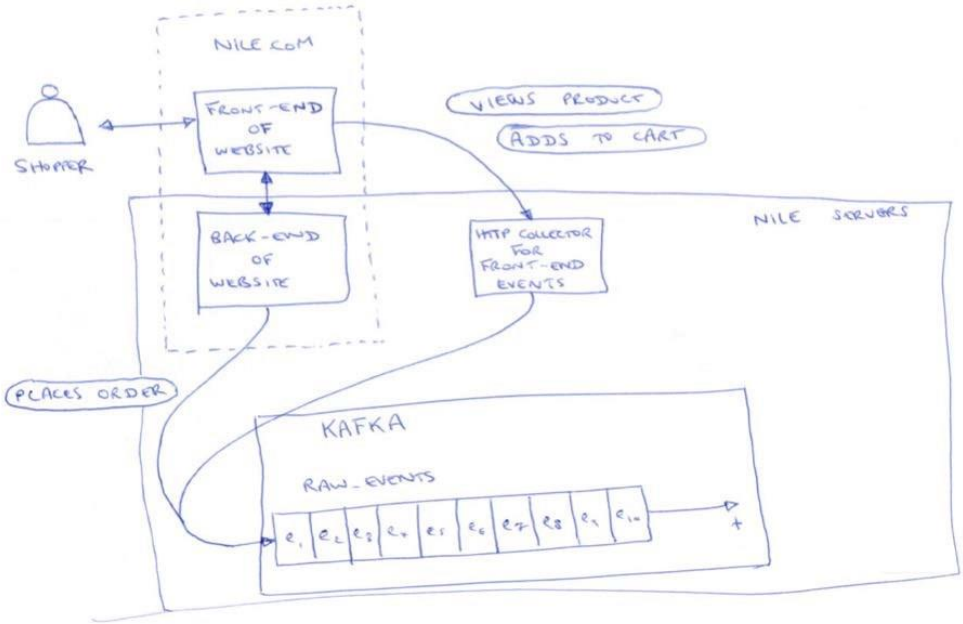


Figure 5. Our three types of event are flowing through into the nile-rawevents topic in Kafka. Order events are written to Kafka direct from the website's back-end; the in-browser events of viewing products and adding products to cart are written to Kafka via an HTTP collector in Nile's server farm.

Once we have our three different types of event flowing through into the `nile-rawevents` topic in Kafka, we can start to think about how to deliver Nile's three "viewing through to buying" requirements. We need to do some kind of event stream processing. Without getting sucked into the implementation details, we can say that we need three discrete processes or algorithms:

An hourly sales calculator – which reads *Shopper places order* events from the `nilerawevents` stream and maintains a sum of total revenues for each hour

A product look-to-book tracker – which reads *Shopper views product* from the `nilerawevents` stream and maintains a count of total views for each product. It also reads *Shopper places order* events from the stream and maintains a count of total units sold for each product

An abandoned cart detector – which reads all three types of event, and applies the algorithm set out earlier to identify shoppers who have abandoned their shopping cart. When

an abandoned cart is detected, a new *Shopper abandons cart* event is generated and written to a new Kafka topic, called `nile-derivedevents`

Figure 6 shows our three new stream processing jobs, all reading events from the `nilerawevents` topic, and either maintaining their own aggregates or, in the case of the abandoned cart detector, writing new events to a `nile-derivedevents` topic.

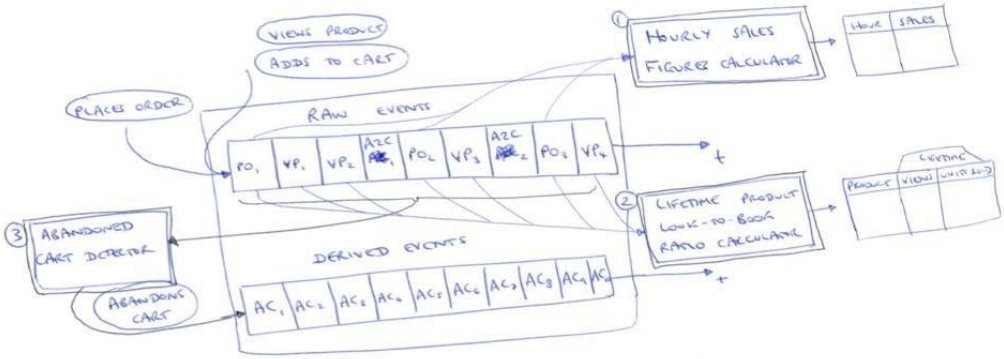


Figure 6. This figure follows on from Figure 5. Our three stream processing jobs consume events from the raw_events topic in Kafka, and either maintain aggregates or derive new events to write back to a new Kafka topic, called derived_events.

There is quite a lot to take in here: we are introducing a second Kafka stream, and sketching out three separate pieces of event stream processing that we have to perform. All three processes meet our definition of multiple event processing, whereby we have to read multiple events from the event stream to generate some kind of output. Specifically:

- The hourly sales calculator and product look-to-book tracker both require *aggregations*: applying aggregate functions such as minimum, maximum, sum, count or average on multiple events
- The abandoned cart detector requires *pattern matching*: looking for patterns or otherwise summarizing the sequence, co-occurrence or frequency of multiple events