Realtime Classification Engine

Msc Intelligent Systems Dissertation

Andrew Haines
University of Sussex
ajh21@sussex.ac.uk

October 25, 2015

Abstract

With the invent of processing platforms that allow for cheap, commodity hardware to run horizontally distributed applications at unprecedented scale, never before have we seen such an explosion of data collection and processing. Because of this abundance of processing power and ever cheaper storage costs, there is a growing trend to capture and collect events, limited only by what can be instrumented and regardless of an immediate requirement. These persisted events sit dormant in data centers waiting for their vast coffers to be harvested for valuable insights and application. Many of the algorithms and developments that have arisen out of this have relied on the fact that even though such sporadically collected data is noisy, with enough of it, the law of large numbers suggests that signals can still emerge and be fed into a new breed of intelligent systems[16]. The problem with this approach is that, although tractable to large institutions and organizations, smaller, more modest individuals will not have the infrastructure or financing to support such data collection even though their reach is enough to warrant its need. Tech startups in particular face this problem with a careful balancing of the business insights and user requirements they can provide at a trade off of vast equity deals in seeding rounds.

In this project we attempt to design and build a closed domain application that runs at 'big data' scale but on a single machine instance. The closed domain of the application is one of a classifier that, given an entity or event expressed in a feature space, it recommends suggested classes based on a continuous stream of online data input. The ideas and concepts that semantically define this recommender are nothing new but its implementation and structure is novel, excelling previous standards by making extensive use of modern computer architecture to process events far quicker than traditional techniques.

Contents

1 Introduction

More and more large institutions these days rely on the analysis of vast quantities of data to inform business decisions and provide performance feedback. From tracking the performance of advertising campaigns, constructing weather models, and powering recommendation engines, Big Data processing is rapidly becoming the cornerstone of the online technology industry. Large companies such as Google and IBM spend hundreds of millions of pounds in capex to construct and maintain the processing platforms required to mine this data. These systems are typically generic constructions developed across many thousands of machines and delivered company-wide to cater for all business units in the organization. Resources such as storage and compute time are then shared across all applications of the company. To accommodate this, either fully distributed but virtualised operating systems that coordinate a 'grid' of machines and networking topologies or complex multi processing core workhouse instances are used to execute such data hungry applications. There are many different paradigms and approaches for powering these systems but they all share one common drawback - these infrastructures come at the cost of an astronomical financial footprint, costing several hundreds of thousands of dollars for comparatively modest setups to multi million dollar investment for corporate scale facilities.

The focus of this project is whether the existing designs and ideologies of these approaches can be borrowed, blended and re-devised to create a system with a real life intelligent application that runs at big data scale but with a fraction of the monetary expenditure. If such a system is possible, the value to individuals where existing big data solutions are out of reach could be unprescended.

2 Background

2.1 Big Data Processing Paradigms

The current state of the art in big data computation falls primarily into three distinct camps.

2.1.1 Batch Processing

The first archetypical model is one that follows a batch processing methodology. The concept is that data is persisted onto a distributed file system in near raw form. Processing then operates on the data either as a one off task for insight analysis or bootstrapping purposes or, at periodic intervals that produce transformed data that can be used in downstream processes. To illustrate this, consider a possible implementation of a 'trending now' feature on a large web property's home page - used to inform users of what is currently popular on the domain at that time.

To determine this, the web property instruments all interactions with various content on its systems. This could be tweets sent from a mobile device app or news articles clicked on via a user's web browser - all events are logged to a central analytics system with a pre defined payload. This payload could contain information about the user[1], the local time the event occurred at, information about the device, the content UUID interacted with... the tuple of data that could be chosen is endless. The analytics servers then typically place the event, via a messaging system onto the distributed file system[2] in a '`rawevents-YYYYMMDDHH-xxxx`' series of files. A scheduled task then runs at say every hour and consumes these latest files, extracts named entities from them and contructs a histogram of their occurrences. The named entities are then ranked according to the histogram's frequency and the top x results are persisted into a '`trending-now-YYYYMMDDHH`' timestamped file. This comparatively tiny file is then sent to all the client facing webservers and powers the last hour's 'trending now' list.

Using the batch approach enables the data to be processed 'offline' in the sense that data is processed some time after the events have occurred. This has a number of advantages over the cost of not being real time. Firstly, the programming concepts can be far easy to implement. Consider the calculation of a median value of a vast list of numerical objects. Using batch you would simply sort the list and pick the middle one as you have all the data for that given hour available. Indeed, if you choose to not remove the intermediate '`'rawevents'` files then larger periods can be scaled up

[1]gender, age, personal preferences, etc

[2]This file system is distributed in the sense that the data actually sits on multiple machines possibly in geo graphically diverse data centers but appear as if it is one contiguous structure of files.

by simply increased the file ranges. Handling in the event of error is also far easier as processing can easily be rerun. The other downside is that the demand on storage capacity is huge but, given the growing trend to capture all data and archive it for later insights, this tends to already be available.

There are 2 main concepts within batch processing that allow it to structure the control flow of the application.

Map Reduce

This technique is an adaptation of an old functional programming paradigm where by two phases of computation are established - the map phase and the reduce phase. Google took this paradigm and ported it to a distributed setting in 2004[8]. The general idea is that during the map task, execution is moved to where the data is rather then data being moved to whether the executable code lies. The reason in the big data setting should be obvious as the IO costs involved in transporting vast quantities of data to executor machines is fair costlier and time consuming then moving a small executable to each of the machine that store the event files. Such machines are referred to as data nodes. Execution in the map phase then runs on the data nodes and is used to filter, transform and assign special keys to each record that this data node holds. The output, being typically alot smaller, 'shuffled' to reducer nodes which may (some IO costs) or may not (no costs incurred) be on different machines before finally being sorted by the special keys assigned during mapping. This allows all items assigned to a particular key to be available on a single reducer. In the case of the trending now example, this would equate to each instance of a named entity being sent to a single reducer so that it's component of the histogram - its particular count - can be computed (see Figure 2.1). In 2011 Yahoo open sourced the HadoopMR[1] module that forms part of the hadoop operating system.

The problem with this approach is that, although it scales well when the distribution of the keys is uniform, in the face of skew in the key set, this bias is propaged down to the reducer load and can mean that a small number of machines can do the majority of the work. Also, as we have seen from the trending now example, the calculation of the distribution of terms is only one part of the entire process. It was assumed here that the entities of all the content that was interacted with had been previously computed. To do this using Map Reduce we would need another set of MR tasks that would extract the content from the raw events, normally only defined by the UUID, prior to the task described. The output is also just the raw counts, we would need a further step to then sort the entities by their counts and only persist the top x. This illustrates what typically happens with complicated MR applications in that a number of sub tasks of mapping and reducing have to occur with intermediate temporary files coordinating the data flow between them. This can be both limiting in how a application is structured and also inefficient due to unnecessary persistence of these temporary files. To solve this the DAG approach was devised:

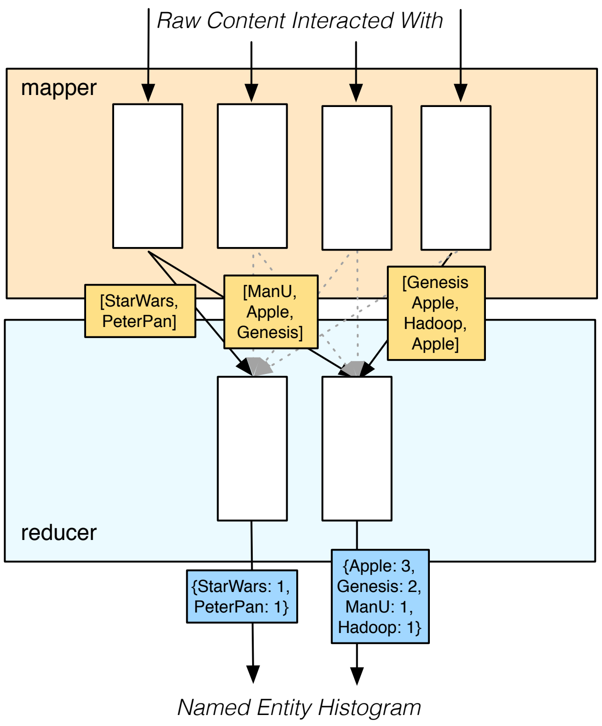

Raw Content Interacted With
mapper
[StarWars, PeterPan]
[ManU, Apple, Genesis]
[Genesis Apple, Hadoop, Apple]
reducer
{StarWars: 1, PeterPan: 1}
{Apple: 3, Genesis: 2, ManU: 1, Hadoop: 1}
Named Entity Histogram

Figure 2.1: MapReduce Trending Now Example. For Brevity only output from 2 mappers are shown. The inputs to the process are the raw content data and the mappers determine which named entities are present. These are then keyed on the textual representation of the entity and all entities of the same name are sent to a single reducer. The output is then a frequency count of all named entities

DAG

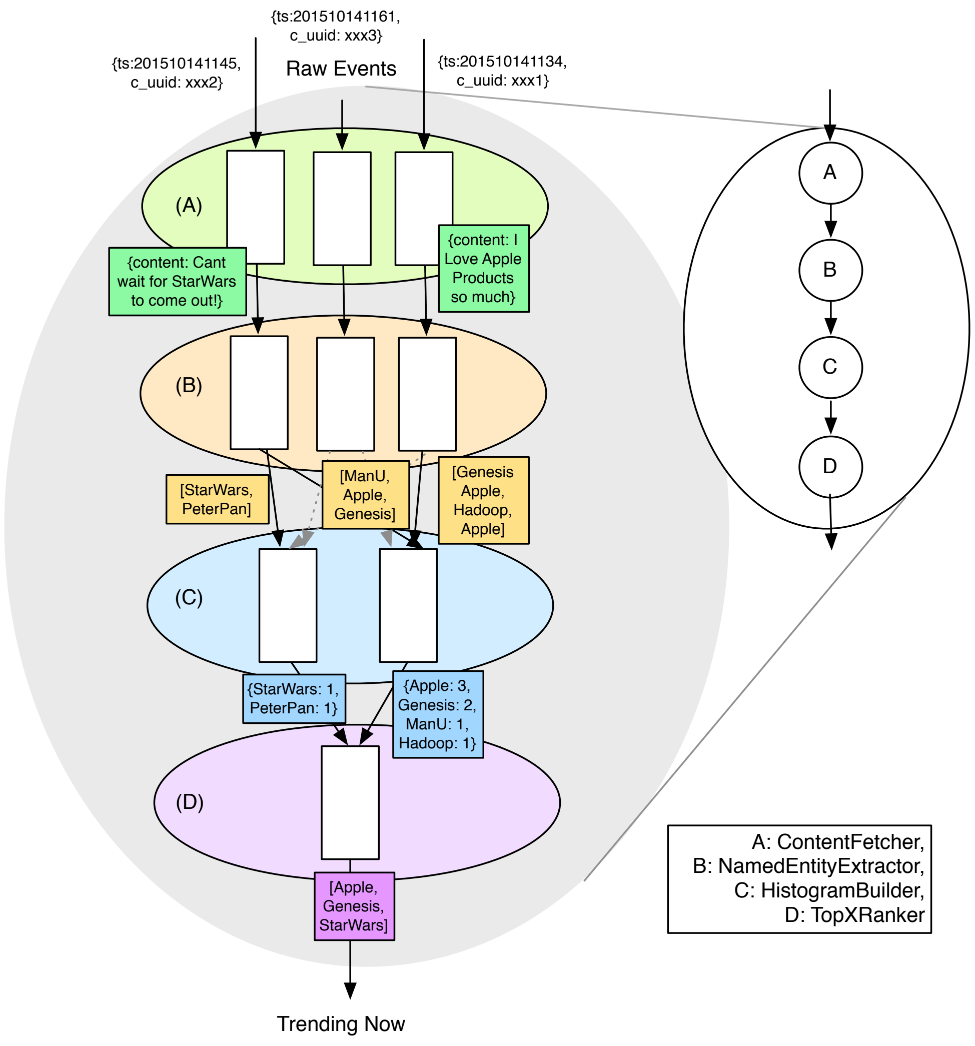

{ts:201510141161, c_uuid: xxx3}
{ts:201510141145, c_uuid: xxx2}
Raw Events
{ts:201510141134, c_uuid: xxx1}
(A)
{content: Cant wait for StarWars to come out!}
{content: I Love Apple Products so much}
(B)
[StarWars, PeterPan]
[ManU, Apple, Genesis]
[Genesis Apple, Hadoop, Apple]
(C)
{StarWars: 1, PeterPan: 1}
{Apple: 3, Genesis: 2, ManU: 1, Hadoop: 1}
(D)
[Apple, Genesis, StarWars]
Trending Now
A
B
C
D
A: ContentFetcher,
B: NamedEntityExtractor,
C: HistogramBuilder,
D: TopXRanker

Figure 2.2: DAG Trending Now Example. In this example the entire application is shown, not just the histogram building. Data flows through the graph along the edges to the nodes. In this example, the DAG framework is able to determine that both the content fetcher and the name entity extractor need never move any data around and so the computation actually happens on the same nodes without ever having to write to disk

The Directed Acyclic Graph[10, 12] approach tackles the deficiencies of MR by defining a graph of tasks that have to be executed concurrently. Nodes in the graph are sub tasks that need to be performed and the edges are data flows between them. Nodes are distributed as in MR by moving them to where the data is but there exists far more flexibility in how an execution flow can be maintained and structured. In the trending now example, a node can exist for extracting the entities of the file, where the determined entities are then fed into the distribution calculation node to obtain counts prior to another node that then filters out the x largest entities (see Figure 2.2). Libraries such as tez[2] and spark[3] are two popular frameworks currently gaining traction in the industry.

2.1.2 Stream processing

One of the big problems with batch processing is that it is not real time. For most applications this isn't too much of a draw back but imagine budget caps on an advertiser spend. Waiting for the next hour's processing window to cut off an advertisers supply could be extremely costly. Assuming that processing never overruns[3], near realtime updates could be possible by reducing the processing window of the batch to a few minutes to help mitigate this issue but this isn't going to help a nuclear cooling processor from reacting to a sudden over heating. Stream processing provides a solution to this by processing events in a stream manner as they enter the system and distributed across a cluster of nodes. This means that events are processed, for the most part, either directly from the network bus or in a temporarily buffer, but never needing to be saved on the distributed file system. Processing is structured as a topology using concepts such as spouts and bolts to represent the data flow and processing respectively. In the trending now example, processing occurs in a very similar manner to that of the DAG approach. The difference being of course that at no point does a node get to see anything other than the current event. This is fine in our trending now example until we get to the ranking stage.

Choices have to be made in streaming about concepts like where an accumulating histogram is stored and for how long does it persist for? There is no point in keeping the histogram for all time as this defeats the purpose of the trending ***now***. One option would be to reset every hour, assuming the behavior of the batch processing methodology - except that it wouldn't. A few seconds after the reset and there may not be enough data seen in the histogram to make accurate predictions meaning that periods just after each reset are inherently noisy as the law of large numbers does not hold. Another approach would be a rolling window but this means that you need to keep all events in memory for each window. The larger the window the more likely to exhaust memory whilst smaller windows may not capture enough data. Also consider the median calculation mentioned previously. Without access to all the events, online versions such as a TDigest[15] need to be used which are far harder to implement and understand. Not only is stream based programming a lot harder but they are also extremely sensitive to the throughput coming and what can actually be processed. If your cluster is not large enough to process the influx of events coming into the system, then events are either dropped or buffered, reducing it's real time nature.

Obviously the upside of the stream approach, other than it's realtime operating behavior, is that it's also super fast as IO is reduced to bare minimum. Apache Storm[4][4]

[3]There is overhead with starting up distributed tasks that can easily make stall resource allocation and cause SLA's to be missed

[4]a commonly used distributed stream processing framework open sourced from Twitter[7]. Apache Spark also offers a streaming mode but underneath the hood this is really a micro batch mode to approximate real time[13]

claims to be able to process 1M events a second using just a 10 node cluster[5]. Furthermore, if storage capacity is deemed a premium, then this approach can allow big data processing to function without the need of a distributed file system.

2.1.3 High Throughput Systems

On the other side of the technology spectrum, high throughput processing used in real time exchanges such as those within financial and scientific industries, are able to process huge streams of data but instead of scaling horizontally by adding more machines into a cluster, a single workhouse instance is used with multiple CPUs, huge memory addresses and dedicated transport buses that coordinate the data between. These are concurrent systems that are considered to scale vertically because obtaining more processing power and memory involves the reconfiguration of the instance itself by adding better CPU's or memory modules. These bespoke systems execute code that is specifically tailored to the underlying hardware which makes them incredibly efficient and able to outperform similar speced Hadoop or Storm clusters. Although typically favoring CPU bound tasks, these so called 'supercomputers' can still process vast amounts of data.

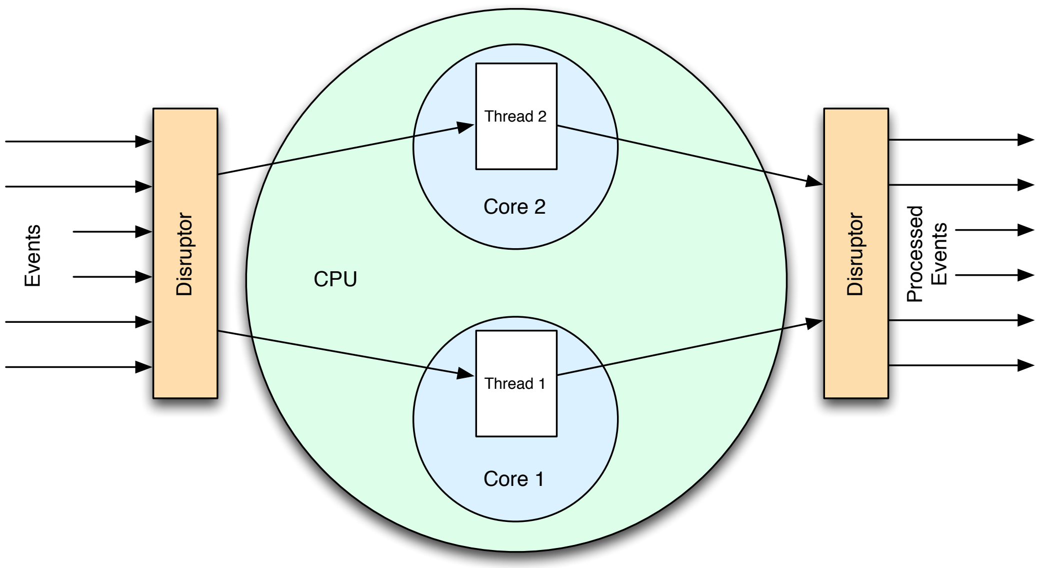


Figure 2.3: An example of a 2 core event driven data flow using disruptors to coordinate the transfer of events

Further to this, the LMAX exchange in 2011 released a high throughput processing toolkit that borrowed from similar principles of software and hardware cooperation. Using techniques borrowed from synergising hardware and software, this toolkit is capable of supporting 25M events a second on a single commodity instance[11]. This is not actual computational work mind, just moving events from one conceptual area of a program to another under concurrently safe conditions. None-the-less, achieving

[5]It is worth mentioning that the author's experience of such throughput requires slightly more nodes in a real setting

such high rates of transfer in in a thread safe manner is still highly impressive. It accomplishes this by using lock-less queues known as *disruptors* that exploit a perfect *mechanical sympathy* between high level programming paradigms and the under lying hardware infrastructure. The basic idea is that disruptors can act as brokers for events, passing them to different areas of code that may or may not be on different threads. This enables event driven design where threads are only used to pin execution to a particular CPU core[14]. These long running threads then can operate on the events from the disruptor using the single writer paradigm which ensures that write contention is no longer a concern and only care about visibility to other reader threads need be considered(see 2.3). The worker threads can then pass events through the system without costly synchronisation and thread context switching.

Disruptors obtain their ultra fast design by utilising cache line structures and memory barriers present on modern cpus. They are essentially queues implemented in ring buffers where the majority of access is completely unsynchronised - care only be taken to ensure that threads have visibility of events on the buffer. Using a `volatile` pointer to the index of the buffer, as long as all event processors do not read past the writer position, no locking is required.

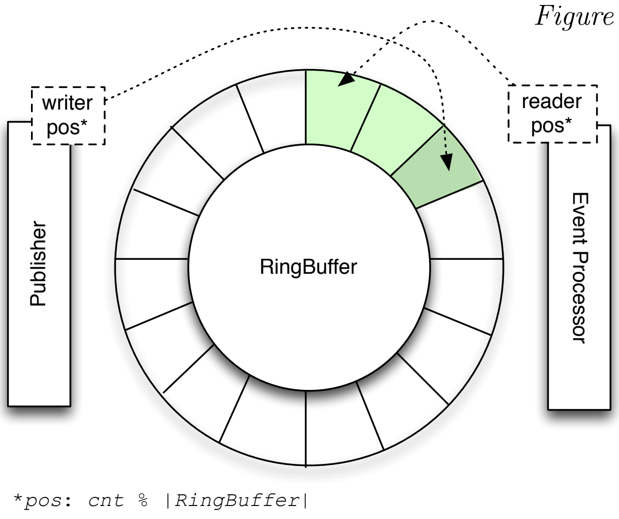


Figure 2.4: The ring buffer at the heart of the disruptor design. For clarity the writer and reader positions are illustrated as if they are concerns of the Producer and EventProcessor respectively - however this is not the case in implementation as the ring buffer owns the pointers in order to ensure $writerPos < readerPos$

In the one scenario where the reader and writer are pointing to the same index, CAS operations are used to *busy wait* on mutual exclusivity on this pointer but without coordination amoungst many threads. Under low contention, CAS operations are extremely efficient. Furthermore, as long as the invariant of $writerPos > readPos$ is held, unlike other cyclic buffers, no end pointer is needed as data can be overwritten when the buffer wraps around.

2.2 Machine Learning Applications at scale

One of the major fields to utilise such large quantities of data is Machine Learning. Tasked with finding systematic patterns in the data and providing classifications and recommendations, ML techniques are used throughout large tech companies to enrich the user experience and provide business analytics. So much so that further abstractions have been provided on top of the data processing paradigms already discussed to enable quick implementations of standardised algorithms and techniques. Libraries such as Mahout[5] run on top of the hadoop platform and provide scalable interfaces to Logistic Regressors, Random Forests, Collaborative Filtering, etc that engineers can use out of the box at scale. All they need do is chose an algorithm, direct it to the data they wish to be analysed with appropriate hyper parameters and the framework does the rest. Spark in fact is so popular with machine learners that it has bundled its MLlib library as part of the core kernel of the framework, focusing on ML at it's heart. When we move into the streaming implementations as provided by Spark's MLlib library for example, the same considerations mentioned earlier need to be applied in the development of how data is structured and processed at each stage of the computation.

Similarly, the popular scientific computing package MatLab offers parallel computing support via its Distributed Computing Toolbox that can utilise 64 CPU cores and beyond, exposing ML libraries to vertically scaled options.

3 Motivation & Scope

As seen, there exists a plethora of frameworks and libraries that enable large machine learning applications to be developed at unprecedented scale. But these approaches all have their draw backs. With typical server instances costing upward of $5K, even a modest Hadoop setup can cost in the order of $50K. Any serious implementation also needs to consider fault tolerant deployments typically with hot-hot mirrored replications in geographically disparate locations. Assuming that data center and rack space need to be acquired, once on going maintenance costs are also factored in, costs to build such an infrastructure can easily approach the $500K mark. This investment is likely to be way out of reach for startups and other small institutions without serious seeding.

To combat these extreme up front infrastructure costs, services such as Amazon's EC2 and Google's ComputeEngine enable clients to 'rent' capacity from their own internal cloud infrastructures. Instances can be acquired on an 'as-needed' basis where entire Hadoop or Storm topologies can be spun up in minutes. This proves extremely cost effective for the periodic applications[1] but for more long term applications, although the up front costs are lowers, long term costs can easily dwarf investing in dedicated hardware[2]. It could be said however that if such vast resources are required then presumably there is a business model in place to support it. The downside is that moving off such a platform is extremely difficult as migrating data and code from a cloud system without down time is a monumental effort.

Cloud services are also not appropriate for certain types of applications. As they are typically virtualised, performance of a single instance is also dependent on other users congesting the resources surrounding it. CPU's, main memory contention, rack IO and network switching can all add unwanted variance to perceived performance making guaranteed response times difficult to predict. For real time applications to work properly, the stream of data in and out need to ideally operate at a consistent rate.

When considering bespoke, vertically scaled instances, cost is also the main inhabiting factor. A modest Cray XC30-AC retails at around the $500M mark which again puts it out of the reach of all but the largest organizations. The difference between the two approaches however is that Hadoop is a platform for running on commodity mass produced hardware that can be scaled outwards as and when extra capacity is required. Once the XC30-AC is exhausted of capacity, expanding the system is a hugely expensive task. Furthermore, these sorts of systems require bespoke knowledge of the underlying

[1] say a TV broadcast or sporting event requiring data capture

[2] Whilst working on EA's SimsSocial online game supported by a 200 node cluster, its yearly EC2 bill came to $1.1M

hardware. Specialised engineers qualified to operate such machines further compound the expense of such operation.

Dealing with such data is clearly a monumental effort and, as such, a reflective degree of supporting assets is clearly going to be required. As we have seen, one of the big advantages of distributed operating systems such as Hadoop and Spark is that they act in a very open domain - that is to say that they can solve a theoretically unbounded scope of problems. The ML libraries we discussed operate on these open domains and provide abstractions for a specific type of application - essentially restricting the domain. These layers of abstractions, although convenient from a development and resource sharing perspective, coming at the cost of performance. As the applications that sit on top of the open domained platform are unknown, mechanical symphony is not possible even though the type of applications that use the machine learning libraries are constrained. Consequently, inefficiencies are rife in the world of the distributed platforms where the tradeoff between simplicity, performance and cost has been focused on the former at the expense of the others[3]

Today's modern off the self servers now come equipped with multiple cores. A 24 core Dell PowerEdge R730xd retails at around $20k which should be considered a sensible, reasonable and comparably modest investment by most standards meaning that such high CPU specs are possible on a budget. By both constraining the problem domain and employing code that encourages a symbiotic harmony between components whilst capitalising on all available CPU cores of the underlying infrastructure of modern commodity hardware, the scope of this project is to build an intelligent real time system that runs at big data scale without the enormous investment costs required by the existing solutions available.

To frame the task appropriately, the system is a real time classifier that, given a stream of labelled event data, builds a temporal model that reflects this input based on positioning in the events feature space. When an unseen and unlabelled event comes into the system, the event is offered an intelligent label based on its internal model and the to-be-classified event's description in feature space. As the structure of the real world data changes, the model also reflects this change so that the classifications returned are representative of the current state of the environment. To provide an appropriate real world application for such a system, consider an adapted trending now feature like the one already discussed. This trending now provides a view into popularity of a site except with the additional ability to provide 'personalised' trends. To illustrate, consider a user that has a previous history of interactions with say technology. When a new iPhone is released, such an entity might triumph over the new One Direction album that maybe popular for other types of users at that moment in time. In this setting, the events that enter the system are user interactions with content. These events are reflected as a tuple of data that can define them in a feature space such as age, location, previous category of interests, time of day, gender etc. The label is then the named entities extracted from the content that they clicked on. When classification occurs for a given user on what their top trends are, the system then returns the top n labels that they are most likely to be assigned to. Figure 3.1 represents the setting in a spatial

[3]There is an argument that, in large institutions, the cost of engineers and bug maintenance where the focus is on simplistic systems actually out ways the infrastructure costs.

context.

Furthermore, like the existing commodity solutions already discussed, the system should not make any requirements or assumptions about the underlying hardware or operating systems. Functionally the system should work on all types of platforms but utilise optimization where it can. For this reason and because it offers low enough support of the underlying infrastructure with this hotspot engine, Java is the language that this system will be implemented on.

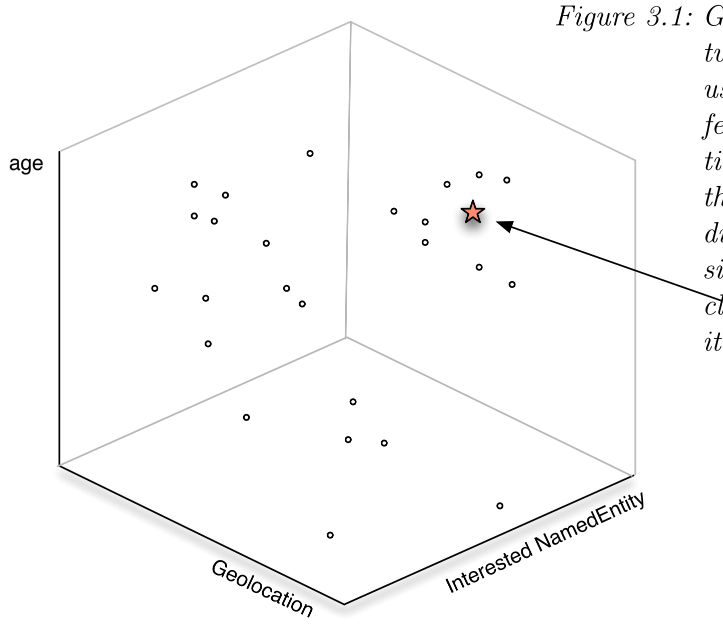


Figure 3.1: Given a trival example of 2 features of age and geolocation, users can be positioned in a feature space to predict the entities they are interested in. In the case of the red user, a prediction can be made that it is similar to the named entities clicked on by the users around it.

4 Design

4.1 Overview

The system proposed is one that places a synergy between the hardware and software at its heart, maximising not just all components of the underlying architecture but also ensuring that each component is utilised as efficiently as possible. The major components considered in this design:

- I/O bus
- Main Memory
- CPU cores
- L3, L2, & L1 cache lines

To understand how these components can be specialised, it is important to discuss how they work and what their role is in the overarching design of the system

4.1.1 Hardware components

I/O bus

Like all internal system buses, the I/O bus is responsible for moving data from an auxiliary part of the computer to the CPU. I/O buses normally exclude main memory access and focus on components such as keyboard input, audio or, in the case interested in this project, network communication. Any data bound server instance like the one in question in this project requires extensive use of the networking interface, so much so that network is the primary bottleneck in such systems as IP packets are marshaled,copied and de-marshalled around the system using the I/O bus as its main *highway* to all components. To ensure that this part of the system is as uncontended as possible, focus needs to be applied to reduce the overhead. In particular, the system keeps all marshalling and demarshalling as conservative as possible and uses the I/O bus to keep, as much as possible, this data away from main memory detours and interfaced directly with the CPU cores.

CPU Cores

In recent times we have seen Moores Law[1] being upheld by the fact that, although clock speeds on single dies are plateauing due to physical constraints of heating and

[1]A law describing the linear trend of core clock cycles/die-size over time

fixed atomic size, the number of clock cycles in a given CPU is still increasing due the addition of multiple cores on the same physical die. The Intel i7 CPU for example is fitted with up to 8 cores on a single die. Moreover, modern servers can be found with multiple CPU's means that upward of 24 cores can be found in modern systems.

Traditional procedural programming is still possible on such architecture but will be limited to a single core. Within a multitasking environment such a design is ok but when considering a single server instance responsible for a single task, threading or multi process design needs to be considered. A common technique is to use the threading model that conceptually isolates a unit of atomic work to occur concurrently with other threads. In a typical web server for example, the threading policy needs to be carefully tuned to obtain the maximum performance. Too few and the system is under utilised whilst too many results in contentious context switching. The problem with the threading model is it does not really reflect with is happening in the underlying hardware. Under the hood, processing is signaled using interrupts[2] which tell the cpu which execution address to jump to next when IO events are received. The only real concurrent unit of work in a computer is what is actually executing on a CPU core at any one time. As such, the threading policy of this system should be such to mimic this behavior - 1 thread for each CPU[3] with an event driven methodology to reflect the interrupt nature of the supporting buses.

Main Memory and Caching

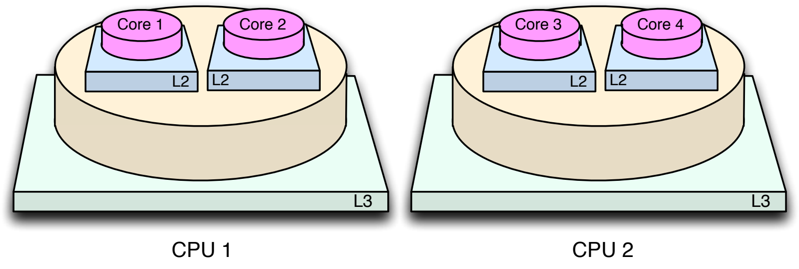


Figure 4.1: Illustration of the different cache visibilities based on a 2 CPU, 4 core setup. Visibility flows downward

Modern CPU's gain extensive performance improvements by using dedicated caching that sits on top of the processing units. These small but extremely fast memory elements sit directly within the CPU die allowing for frequently required data to be accessed at a fraction of the cost of accessing main memory. Caches map main memory in the form of cache lines that are contiguous areas that exploit both temporal (items used at time t are likely to be used at time $t+1$) and spatial locality (items located near each other are likely to be used at a similar time). Data of a cache line is moved from slow main memory into fast cache memory as it is being accessed. To optimise it's use of memory,

[2]In fact threading is achieved by *multiplexing* these interrupts so that the illusion of atomic and concurrent units of work can be achieved

[3]Ideally using thread affinity where the OS is instructed to only execute a given thread on a given core

the system needs to exploit the locality principles of the cache lines, avoiding cache misses and keeping data as close to the CPU for as long as possible.

The downside with such design is that when data is sitting in cache lines, it is not guaranteed to be visible in other parts of the system. Data stored in the L2 cache of Core 1 of Figure 4.1 cannot be seen in Core 2. Furthermore, data in the L3 cache of Core 3 cannot be seen by Core 2. Consequently, care has to be taken to set up correct memory barriers[4] when data is needed to be flushed back into main memory for other cores to see it.

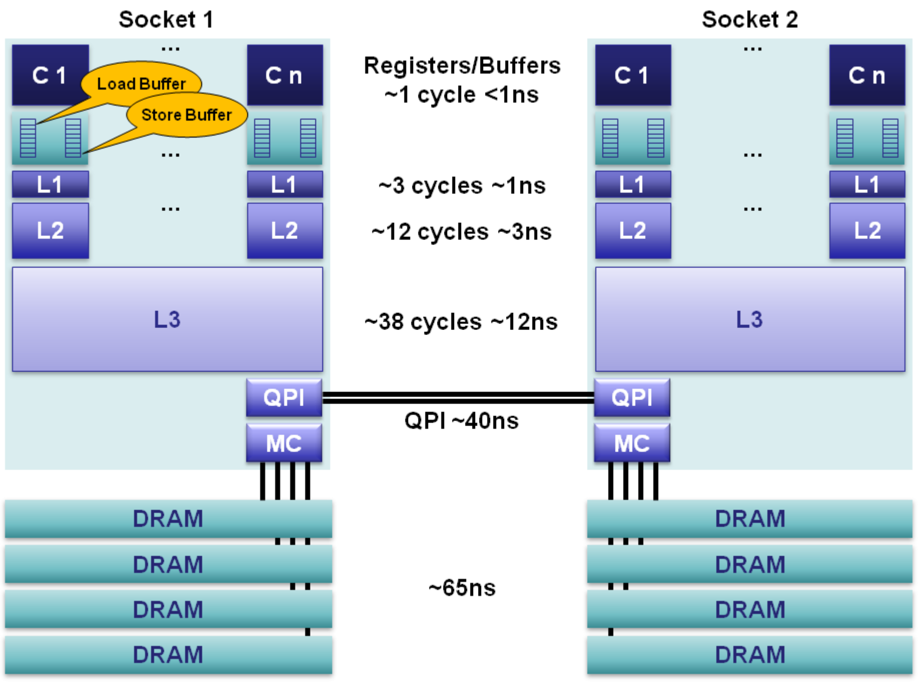


Figure 4.2: Although there are multiple cache lines present in modern designs, as observed here the real speed benefit is avoiding main memory and as such this is the focus in this project.

Another consideration that's worth highlighting is that within managed virtual environments such as java, although all these components can be exploited using care consideration of the JVM contracts, nothing is guaranteed. The garbage collector in particular can become a catastrophic obstacle to such optimizations. Consequently, as much as possible, object pooling is used[5] and a precedence placed to avoid unnecessarily object creation.

4.1.2 Software Architecture

As previously mentioned, in order to optimize all the hardware components mentioned above an event driven design needs to be adopted, filing off to threading only to dis-

[4]In java for example, memory barriers are created using the `volatile` & `sychronized` keywords and the `Sync` primatives from the concurrency package

[5]Actually implemented transparently within the disruptor

tribute the computational load across processing cores. To avoid the OS from scheduling another thread to a core whilst being worked on and thereby invalidating the caches, the number of threads is equal to -1 the number of cores in the system[6].Once events have been handed over to a specific core, the thread that owns the event also *owns* its visibility completely. That is to say that no other thread is expected to have read or write access to it whilst in this core's care. This allows the application to enforce locality benefits in the data without worrying about visibility to other threads and thus providing a setting that minimises the copying of cache lines to and from main memory and satisfies the desire to keep data as close to the Core as possible. When the core is finished with the event, a model of releasing ownership of a collection of events at once is adopted by *pushing* these events downstream to another component where visibility is required.

To facilitate the movement of these events from the input stream into the care of dedicated cores and finally, back into main memory, the use of the disruptor queues already mentioned is selected. Note only are these queues lightening fast but they also put in place the required memory barriers mentioned to, like a runner in a relay, pass off ownership of the event once it is finished with and providing the necessary visibility guarantees. Another side effect of this design, as will be seen, is that it is completely lockless, only relying on the CAS operations in the disruptor under busy load. Furthermore, there is only one custom `volatile` variable when running the system in *Sync* mode. A rough architecture plan of the system is presented in Figure 4.3

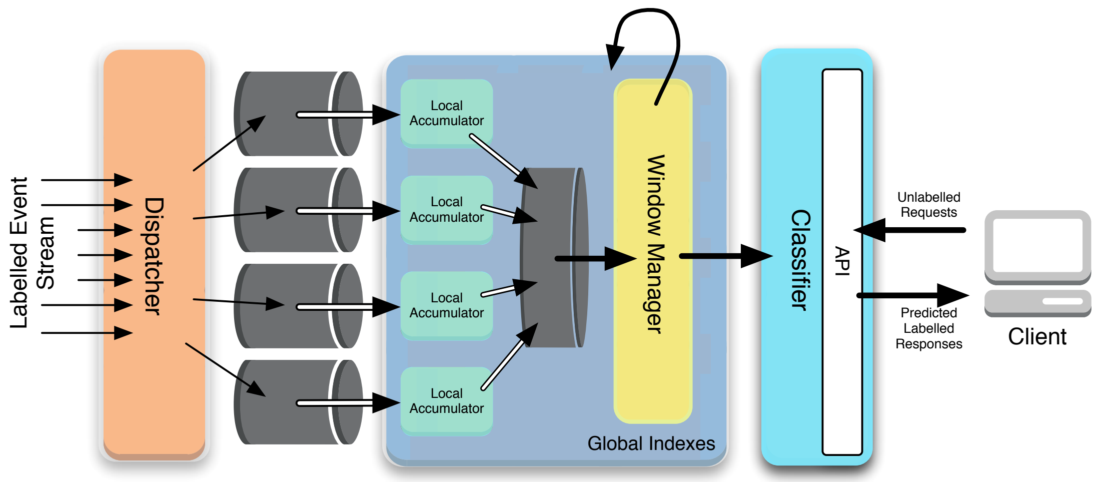


Figure 4.3: The main modules and data flows of the system

Events flow into the system and feed into the **Dispatcher** module by a single main thread. Here the events are randomly pushed into a series of disruptor queues, one for each CPU core using a simple uniform distribution. Each disruptor can be thought of as defining the work queue for each core whilst a dedicated thread associated with each one, consumes events one at a time. These worker threads then construct an accumulated view of a so called window of events. To aid in this, the **Accumulator**

[6]-1 is used to dedicate a core to the main scheduling thread of the system that deal with the IO coordination

module exposes a highly cache efficient data structure that each thread has a *local* instance of. These local instances are completely owned by each thread permitting both read and write operations and exist to fully exploit the cachelines of the underlying associated core. In order to maximise temporal locality, a shared *global* index that can be used within the local accumulator so that more long term insights into the data can be shared across threads. To avoid unnecessarily cache flushes, this shared instance is completely immutable allowing only read access across the workers. Enforced using `final` declarations, data access from the global indexes can be essentially copied into the cache lines of each worker thread, maintaining the single ownership properties required of the worker data accesses - In this respect, the global index can be thought of as a read only instance that each thread has a copy of[7].

Once a window of events has been processed, the accumulated view is then pushed back into the main thread via the use of a single shared disruptor queue. The main thread then consumes these accumulated events and aggregates them together using a **Manager Manager**. This module takes each of the accumulated events and combines them into a single *window* which represents a combined view of the data for a particular interval of time. As the accumulator threads push more and more windows into the manager, so it also keeps track of each interval, accumulating further to produce a single overarching representation of all accumulator windows and intervals. This serves as the main aggregated model of the system. This aggregated model is not only exposed to the **Classifier** to produce predicted labels to unseen events, but also back into the global accumulator so that the worker threads can profit from this longer sighted view of the data.

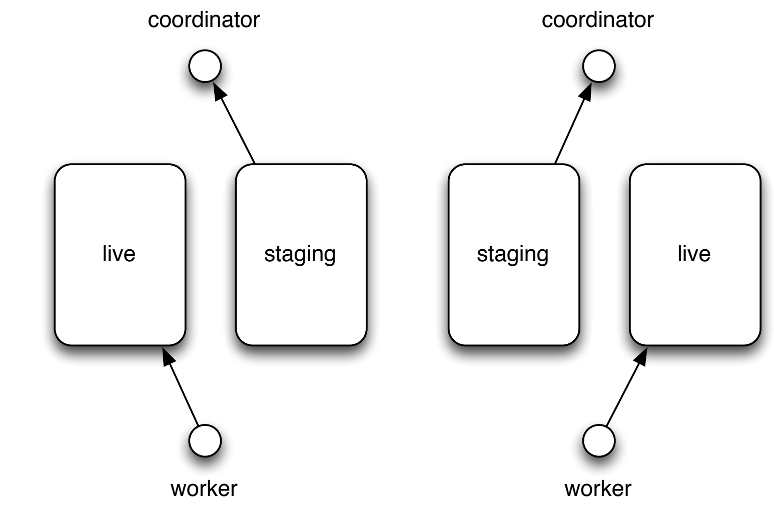
4.1.3 Execution Modes

To coordinate the determination of when a window of events has been processed the system uses a time expiration rather than a number of samples policy. The reason is that it parallels better the temporal nature of the window and makes the aggregation stage in the window manager easier to implement. Consequently, There needs to be a protocol for determining when the window has expired. There are 2 modes of execution.

Async

This is the most simplest mechanism to implement and revolves around a coordinator thread that, for the most part just sleeps. Every t seconds it wakes up and proceeds to read the contents of each accumulator, copying it into main memory before sending it into the window manager. The difficulty in this approach is that case needs to be taken when reading from the accumulator as this process not only needs to be visible (This is data owned by another thread and therefore offers no guarantees that the data will be in main memory and not on a cache line somewhere) but also atomic. Whilst we are reading from the accumulator we cant have that worker also updating it at the same time else data corruption becomes a risk. Consequently, if we use synchronization to coordinate the access any advantages of cache optimization are negated as the implicit

[7]When a worker accesses an item in the global index, the OS is free to copy the data into the local cache of whatever core the thread is assigned to



a: t **b:** $t + 1$

Figure 4.4: How ownership of the accumulators change during the end of an interval

memory barrier that comes with such locking will cause all writes to be written back into main memory, essentially throttling the event processing of the system.

To avoid this, a setup up borrowed from *hot-cold* DR topologies of server cluster is adopted where 2 copies exist of each local accumulator - one live and one staging. The purpose is that only one of these accumulators is owned by the worker thread. When the coordinator thread wakes up, it switches the live to staging and vice versa meaning that the worker thread relinquishes sole ownership of the live accumulator and takes on the staging one. Once the live accumulator is now staging, the coordinator thread can safely and atomically access the data in the accumulator. This can be implemented using just a single `volatile` bit mask that determines which accumulator is live.

Sync

The other execution mode is one where there is no coordinator thread. Each worker is responsible for determining when a window is ready and the accumulated event should be pushed downstream as part of its actual processing task. This means there is no ownership transfer or extra memory impacting accumulator copies. One potential drawback of this approach is that, if there are no events in the system, the thread cannot check to see whether it current accumulator has expired and window intervals can become stale. To combat this, *heart beat* events are added by the dispatcher at the boundary of each interval to ensure timely expiration of windows occurs. Because the main copying of the accumulators into main memory is performed on the work thread itself, this approach leads to short pauses in the event stream at window

4.2 Naive Bayes

Potentially any classifier could be used but, as there is a focus to optimize the model to exploit the cache lines, one that can form a model representation using simple counts is extremely useful. Addition of simple variables in a continuous array is something that is extremely cache friendly and, as such the Naive Bayes technique is a perfect fit. Naive bayes works by building a probabilistic model using prior and posterior combinations of feature value/label pairs under strict naive assumptions of independence. The idea is that, by building probability distributions from observed events, unseen classifications can be predicted by maximising the likelihood estimates from the built distribution of all classifications based on the unseen event's feature values.

$$\hat{y} = \underset{k \in \{1,\ldots,k\}}{\operatorname{argmax}}\, p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

Framing the classifier as a personalised trending now predictor, the formula is changed slightly so that, instead of returning the classification with the maximum probability, we return the n top most probable classes. By setting $n = 1$ this functions the same as the original argmax revision.

$$\hat{y} = limit(n, \underset{k \in \{1,\ldots,k\}}{\operatorname{rank}}\, p(C_k) \prod_{i=1}^{n} p(x_i|C_k))$$

The prior $p(c_k)$ term captures the traditional trending now function whereas the posterior $p(x_i|C_k)$ captures the personalised extension.
To build the probability distribution using an accumulator methodology is simple. When a labelled event is observed, an accumulator slot is referenced based on the classification value (to capture the prior) and one for each discrete feature value / classification pairing (capturing the posterior). The values in each of this slots is then incremented by 1. The probability for each prior term is then the total number of events seen of each classification value over the total number of events and the posterior terms is calculated as the total number of events seen of each feature/classification pairing over the total number of events of that particular class.

$$p(C_k) = \frac{n(C_k)}{\sum_i n(C_i)}$$

a: Prior term calculation

$$p(x_i|C_k) = \frac{n(x_i|C_k)}{n(C_k)}$$

b: Posterior term calculation

For continuous values, distributions can be assumed to represent probability density functions. In the case of a normal distribution, for a given continuous feature or classification type, the prior and posteriors can be estimated by building online streamed based mean and standard deviations based on the event stream.

$$p(C_k) = \frac{1}{\sigma(C)\sqrt{2\pi}} e^{-\frac{(C_k - \mu(C))^2}{2\sigma(C)^2}}$$

a: Prior term calculation

$$p(x_i|C_k) = \frac{1}{\sigma(C_k)\sqrt{2\pi}} e^{-\frac{(x_i - \mu(C_k))^2}{2\sigma(C_k)^2}}$$

b: Posterior term calculation

As it is not possible to store all event data in memory, calculation of the μ and σ

variables needs to be conducted in an online fashion[8] where just 3 variables can be used to capture the distribution parameters (namely n, μ and M_2). Each variable can occupy a single slot in the accumulator and are stored sequentially to exploit the spatial locality of cache lines.

$$\mu_{i+1} = \mu_i + \frac{(x_i - \mu_i)}{(n_i)}$$

a: Mean Calculation

$$M_{2_{i+1}} = M_{2_i} + (x_i - \mu_i)(x - \mu_{i+1})$$

b: Intrim Calculation

$$\sigma_i = \frac{M_{2_i}}{(n_i - 1)}$$

c: Variance Calculation

4.3 IO

In order for events to enter the system, a sensible IO strategy is needed not just to transport the bytes but also to deserialize them into a format that can be made sense of. This deserialising is a major bottleneck in any IO heavy system so care needs to be taken to ensure it is as efficient as possible

4.3.1 Selectors

As previously mentioned, the IO bus on modern architectures relies on interrupts that occur when data is available on the IO bus. In order to mimic this as much as possible, the system utilises the selector mechanism of *nix operating systems that approximates a 1-1 mapping between these interrupts and a kernal event from the OS. Java's NIO module provides an interface to this system. Furthermore, normally memory is indexed in java using a specialised heap that the GC can maintain. Traditionally, all data accessed within the JVM needs to be present in this heap but for networking, this means that data on the IO bus needs to first be copied into main memory prior to being processed. For the stream based methodology that this system utilises, this unnecessary detour not only adds extra transfer time as data is proxied through the heap but also puts extra burden on the garbage collector. To limited this, advantage is taken of the off heap byte buffers of the NIO package. These abstractions ensure that data is taken directly from the network interface into the CPU for serialising, and thus avoids commuting to the heap.

4.3.2 Protostuff

Serialisation that governs data interchange needs to be defined so that a contract of how events can be passed from up stream producers into the system. As deserialising is one of the main overheads of any high throughput system, one that focuses on both flexibility and speed is paramount. Consequently, Google's protobuf[6] format is provided using

[8]This online version proposed by Knuth[9] is an adaption of Welford's online variance method and is used due to its better handling of overflow

the protostuff extension as a default implementation[9]. This was chosen because of it's external schema design, variable length encoding and sequential byte stream support that aids in good cache utilisation. Unfortunately, protostuff does not support proper off heap byte buffers so an extension was specifically created to exploit this[10].

4.4 WindowManager

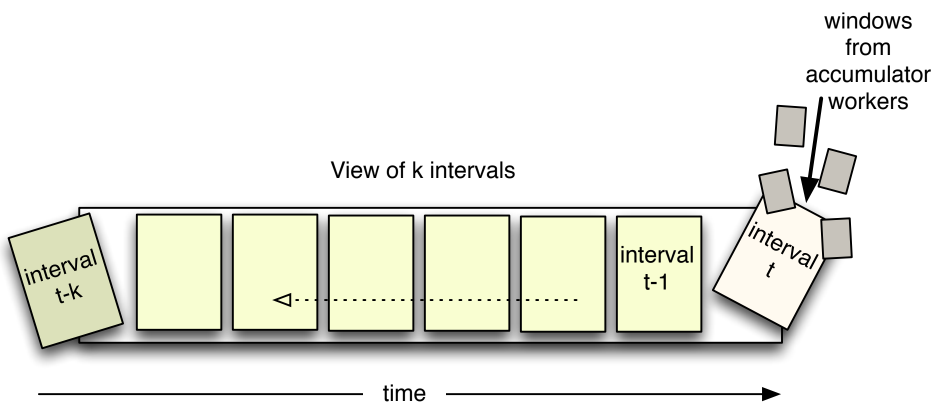


Figure 4.8: Illustration of how a new window interval arriving into the Window Manager will push out an older interval

The window manager is responsible for aggregating different event windows and intervals and controlling the expiration of each. It follows a rolling window design (see Figure 4.8) where by a view of k intervals of i time is maintained at any given moment. Each interval, created at time t is built from multiple windows from the accumulators which are combined together until the interval is no longer current at $time = t + i$. At this point the interval is made immutable and a new interval created and set as current.

The total view of the window manager is then maintained by a separate aggregated state that represents all the aggregated intervals together. For discrete features, whenever an interval at time t is added to the view, the prior and posterior counts of this new interval are summed together whilst, when interval $t - k$ leaves the view, its counts are subtracted from the aggregated state. When dealing with continuous values, the parameters of the underlying models need to also be summed and subtracted. For the case of the normal distribution, the parameters are summed and subtracted using the following bias corrected equations:

[9]The application is capable of supporting any demarshalling technology via the generic `EventMarshalBuffer` interface

[10]Now contributed back into the protostuff project

$$\mu_c = \frac{n_1\mu_1 + n_2\mu_2}{n_1 + n_2} \tag{4.1}$$

$$\sigma_c = \frac{(((n_1 - 1)\sigma_1) + ((n_2 - 1)\sigma_2) + n_1(\mu_1 - \mu_c)^2) + n_2(\mu_2 - \mu_c)^2)}{n_c - 1} \tag{4.2}$$

Equations 4.4.1: Addition of normal distribution parameters for each windows/intervals

$$\mu_s = \frac{n_1\mu_1 - n_2\mu_2}{n_1 - n_2} \tag{4.3}$$

$$\sigma_s = \frac{(((n_1 - 1)\sigma_1) - ((n_2 - 1)\sigma_2) - n_1(\mu_1 - \mu_c)^2) - n_2(\mu_2 - \mu_c)^2)}{n_c - 1} \tag{4.4}$$

Equations 4.4.2: Subtraction of normal distribution parameters for each windows/intervals

4.5 Accumulators

The accumulators are specifically tailored data structures used to optimize the cache layouts of the underlying hardware and represents the cornerstone of aggregating events into the accumulated models and make up each window. Essentially, they represents an indexable sequence of memory that focuses on structuring the indexes in such a way to exploit both the temporal and spatial qualities of the cache lines. This abstraction of position and value is the heart of the structure. The accumulator is essentially an expanding tree that indexes a leaf based on a given slot number that references the required value. Each leaf is actually a primitive array of integers created with a length equal to that of the cache line of the underlying CPU cores. The values that these slot numbers refer to relate to the variables needed for the naive bayes classifier and are determined by different lookup strategies according to the instance type of the accumulator.

The idea being that, most frequently accessed data sits together in the same leaf array and therefore the same cache line, optimising for cache hits and avoiding fetches into main memory. The structure grows as more slots are needed so that large and unnecessary allocation of array space can be avoided. To obtain a slot in the accumulator, its location in the tree is determined using a 3 level accumulator line structure where each line can be considered a depth in the tree. which branch at each level is then determined using the high order bit terms of the slot value. This results in slot numbers where the low order terms are similar to be more likely to reside in the same cache line.

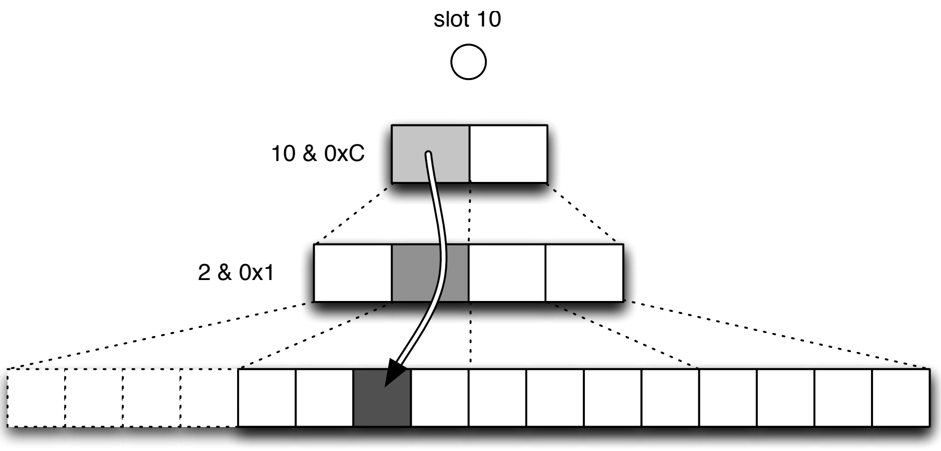


Figure 4.9: shows a trival example of indexing slot 10. Note that the left most outer leaf is not assigned demonstrating the expanding and space saving design

4.5.1 Index Instances

For each worker thread there exists a dedicated accumulator instance that it has complete ownership of. All reads and writes are solely for the purposes of the owner thread which allows the cache optimistions detailed previously to function. In the naive bayes framing, a particular prior or posterior pairing needs be assigned to a particular slot in order to index into the accumulator in a deterministic manner. This lookup follows a hierarchical model where a given feature or feature/classification pairing is first checked in a global instance. If the global instance returns a slot number then this is used to index the accumulator - if not, a local instance is used.

Local Instance

Local mode indexes as ones that are completely unique to a given worker and are mutable in that new indexes cannot be assigned. If a given feature or feature/classification pairing is not present in the index, it is subsequently added against the next available slot in the accumulator. When slots are determined from local indexes, only spatial locality can be exploited as, being on the critical event stream path of the system, they need to function as quickly as possible.

Global Instance

This is a read only shared instance that all worker have access to. After every time the window manager is updated with a new interval, indexes for all seen required variables (both for discrete and continuous feature types) are re computed based on a ranking of how frequently they have been seen. If $p(x_1|C_1)$ is the most observed posterior and $p(x_2|C_2)$ is the second most observed, then these will be reshuffled in to slots 0 and 1 respectively. By performing this reordering once every interval, temporal locality can be exploited by referencing the updates into the same leaf and therefore same underlying cache line. As this is a shared instance, a single `volatile` reference is shared amongst all workers and, as it is read only, no explicit locking is required, just the visibility guarantees. On each interval update, the window manager generates a new global index and reassigns the `volatile` reference to this new one for all the workers to see. To avoid boundary cases during updates, the global reference maintains its own namespace in the accumulators. This means that a given variable may be indexed from both a local and global slot and, as such, when constructing prior and posterior probability distributions, both need to be considered.

5 Implementation

Discussion on how the design is implemented. Only areas of implementation note are provided as this section could get out of hand! Core interfaces such as EventConsumer, Accumulator, FeatureHandler, ClassificationHandler etc.

5.1 Maven Modules

How the application is structured and the reasons why for extension and customisation

5.2 Distributions

How the distribution framework for continuous features is defined

5.3 Configuration

overriding classpath configuration of jaxb objects.

5.4 Testing

How the DataSet classes abstract the test and training sets and produce their results.

6 Performance

6.1 Classification Performance

PerformanceTest results. AUC curves on the different data sets tested

6.1.1 Synthetic Test

Synthetic test to see if it can learn multivariate distributions in the presence of noise. Also mention how it is used in the load test to generate BigData equivalent volumes.

6.1.2 Adult Wages

A classification Task

6.1.3 Space Shuttle

A regression Task

6.2 Throughput Performance

LoadTest results. TCP vs UDP

7 Conclusions

7.1 Results

Discussion on how the results presented in the performance section compare with other systems.

7.2 Limitations

No FeatureExtraction.

7.3 Future Work

Plugging in more classifiers. Spatial models such as RBF's etc Static analysis should be conducted, Bucketed Continuous accumulator types using object pooling from protostuff rather than Schema.newMessage()

Bibliography

[1] URL: `http://hadoop.apache.org/docs/current/`.

[2] URL: `https://tez.apache.org/`.

[3] URL: `http://spark.apache.org/`.

[4] URL: `http://storm.apache.org/documentation/Rationale.html`.

[5] URL: `http://mahout.apache.org/`.

[6] URL: `https://developers.google.com/protocol-buffers/docs/reference/proto3-spec`.

[7] et al A. Toshniwal S. Taneja. "Storm @Twitter". In: *SIGMOD*. Twiiter. 2014.

[8] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI*. Google. 2004.

[9] Donald Knuth. *The Art of Computer Programming*. Addison-Wesley, 1998.

[10] et al. M. Isard. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks". In: *EuroSys* (2007).

[11] Et Al M. Thompson D. Farley. *Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads*. Tech. rep. LMAX, 2011.

[12] et al M. Zaharia M. Chowdhury. "Spark: Cluster Computing with Working Sets". 2010.

[13] et al M. Zaharia T. Das. "Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing".

[14] TMurgent Technologies. *Processor Affinity - Multiple CPU Scheduling (White Paper)*. Tech. rep. TMurgent Technologies, 2003.

[15] Otmar Ertl Ted Dunning. "Computing Extremely Accurate Quantiles Using t-Digests". https://github.com/tdunning/t-digest/blob/master/docs/t-digest-paper/histo.pdf.

[16] Kenneth Cukier Viktor Mayer-Schönberger. *Big Data: A Revolution That Will Transform How We Live, Work and Think*. John Murray, 2013.