Computing is still pretty immature. Not that it does not work; it does. In fact, lots of customers are happily computing away in the cloud, for purposes as diverse as additional compute power, development and test environments of any required configuration, access to software more immediately (with fewer internal resource requirements than traditional), in-house implemented software and with the ability to create shared environments outside the organizational firewall, where all kinds of workers (employees, contractors, suppliers, customers, and others) can collaborate.

Rather, we are all still testing the boundaries of exactly what we would like Cloud Computing to be:

- vendors want to take advantage of a hot new marketing term
- customers bring their normal expectations to a new technology: they expect it to solve every problem.

Cloud Computing is no different than other technologies. It can offer extraordinary and growing capabilities that will make it attractive to most business customers for some things. But no technology will ever solve every problem and it is when the users recognize the boundaries of a new technology and routinely use it for its strengths that a technology begins to mature and be genuinely useful.

Amy D. Wohl
Editor of:
Amy D. Wohl's Opinions and
Amy D. Wohl's Opinions about SaaS
www.wohl.com

40GB of daily database processing — on $10/day

Management introduction

Extrabux, based in San Diego, California (http://www.extrabux.com), is an online comparison shopping engine (CSE). Purchases made through Extrabux attract a discount (negotiated by Extrabux) from stores and retail chains. In addition to finding the lowest price for a product online, by integrating coupons with its comparison shopping engine, an additional differentiator for Extrabux is that it passes the majority of the discount obtained back to the shopper. The shopper can choose to keep it or to donate it to nominated charities.

For the Extrabux service to work it must process huge volumes of data each day -- already at 40GB+/day and climbing. Rather than use a conventional Linear Licensed Software (a.k.a. proprietary) database product, Extrabux uses Non-Linear Licensed Software (NLLS) via the version of Hadoop provided by Amazon's Elastic MapReduce.

As Patrick Salami, Senior Software Architect at Extrabux, discusses in this case study, the results --- while not necessarily simple to obtain -- are spectacular in terms of volumes processed, the reduced time taken and the cost efficiencies obtained by Extrabux data processing system design; this is built entirely in the Amazon cloud.

Some background

Extrabux is a comparison shopping web site. It finds the lowest price online for millions of products by integrating coupons as well as cash back rewards into its powerful search engine. To achieve this, It has direct relationships with a number of top online retailers plus it works with affiliate networks which assist merchants in publishing their data as feeds. Today Extrabux provides access to 2,000 of the top online retailers and continues to expand its reach. It add more retailers every day.

Through its relationships with those retailers, Extrabux currently has access to a total of 70+ million discrete products for sale. The details of all these items and prices are sent to Extrabux daily as a series of

individual data feeds which need to be processed each day in order that the most up-to-date product prices and information to appear on the Extrabux web site. In practical terms this amounts to some 40GB (and growing) of raw data which arrives via batch feeds.

Unfortunately, this raw data is differently formatted by each source. It needs, therefore, to be parsed into a common format so that it can be processed and indexed in order that Extrabux customers will be able to search for and compare products and prices on the Extrabux web site. To do this, the raw data arrives at a certain time early each morning. In order to stay competitive and provide the most up-to-date data to our customers, Extrabux must complete processing and indexing all the disparate data feeds as soon as the raw data is available.

Our first imperative is, therefore, that we be able to complete all the processing in as short a time as is economically possible. To compound this challenge, the size of the data feeds and number of items may vary each day -- as we add new retailers and as retailers send us information about new products. This is highly variable:: one day we might receive (say) 60M items; another it might be 80M. Nevertheless, the window for processing never stretches; by having the most up-to-date data we obtain our competitive advantage. The Extrabux system is, therefore, architected in such a way to enable us to process a virtually unlimited amount of data in exactly the same time window.

In addition, I should emphasize that the data we receive really is raw. Product records are not pre-processed or filtered in any way; there is no sorting, no order and not even a common format across the various feeds. This means that we have to apply a series of relatively complex algorithms to the data before it is even ready to be indexed. For example, we need to categorize each product using a custom machine-learning algorithm plus we must group identical products together by a unique identifier, such as via a UPC. Prior to that, we have to make the data homogeneous which includes filling out missing fields where possible and filtering out incomplete or invalid records.

Why use MapReduce?

Although we have been using MapReduce since we started building this system, we did not consider using it as a replacement for a relational database until we discovered that the RDBMS technology solution was simply not up to the task (I will talk about why, later). The attraction of MapReduce is that it can exploit a very large number of computer nodes in which the Map stage takes input -- record by record -- and, after any applicable processing, re-organizes the records in the desired fashion and distributes them across the distributed computing nodes that comprise the MapReduce cluster.

For us the advantage of the MapReduce approach is that we can distribute our 40GB of raw data across multiple nodes for processing before it is re-assembled into a clean, structured data set that is ready to be indexed. Even though the amount of input data may change dramatically, our processing time remains constant - if more data needs to be processed, we simply add more nodes to the cluster.

In practice we use multiple different JAR files to process the data, which primarily consists of:

- parsing and filtering the data
- applying our custom categorization scheme
- calculating the applicable cash back and coupon elements
- grouping identical products together.

Amazon's Elastic MapReduce service makes it easy for us to process the data in multiple steps, using different JAR files if needed. Amazon also enables us to flex the processing power required depending on each day's input data volume.

In other words, we can add to, or reduce, the number of nodes and storage deployed each day for processing. Indeed, we now use an automated script which looks at the size of the data to be processed and, from this, determines the number of MapReduce and Solr index nodes that are required for that day. Furthermore, we only provision the MapReduce nodes until the data has finished processing and we only pay for the compute hours that we actually use.

From deploying the needed resources to ending the processing the 40GB takes about 90 minutes; of this time, a certain portion is spent writing data to S3 (storage) and other Elastic MapReduce overhead. 'Our' processing only takes about 60 minutes.

In processing terms, our normal usage of Elastic MapReduce exploits some 32 nodes, each with 2 CPU

cores (in Amazon's terms -- 'c1.medium' nodes); uses 1.7GB of RAM memory plus S3 storage for loading the 40GB of input data and for storing all the correctly formatted and sorted output data.

Why was the RDBMS approach discarded?

When we started out we expected that the traditional RDBMS approach would be sufficient for storage and ad-hoc retrieval of our bulk data (in essence, for generating metrics). We thought this because much of the processing we do is complex: there is much more than filtering and sorting. For example, we create detailed metrics about which retailers have the most products as well as the most bad product records (so we can work with them to reduce this) plus we generate statistics that look at how product data changes over time -- which helps us organize the data for our online customers. In addition we have found it necessary to define, and calculate, a myriad of other data points about products, about retailers and about customers, as well the many relationships between these -- so that we understand how to improve our customer experience.

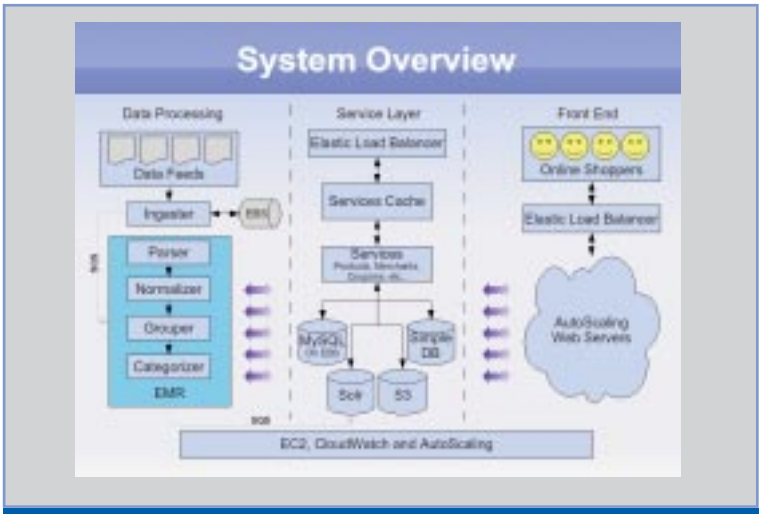

Figure 1

Initially we tried MySQL. Indeed we tried to make this work for too long a time -- because it did seem to be the logical choice.

We did manage to store our daily product data in a MySQL database with over 70m records. But regardless of the configuration, many of our queries would run for hours or even days. That loading the database with the fresh data took upwards of 6 hours, even when using the faster "load data infile" method, defined the scale of the problem.

We saw some performance gains when using MySQL Cluster within the Amazon Cloud (with 4 data nodes, a management node and the MySQL node). But, while this worked better, the latency of the disks and network in the cloud introduced variables that we could not predict. Not only was loading slower but queries still took much longer than we needed, and it was all even more expensive (because of the additional 6 large servers required), not to mention the overhead associated with setting up and maintaining a live MySQL Cluster at all times. (Trying to store 40GB+ in memory and keeping the cluster up at all times was simply not cost effective; the alternative was storing the data on disk, which partially defeated the purpose of using MySQL Cluster.)

The net result was still too slow to work with -- however imaginative we were. We knew that we needed something different. To cap it all, when we discovered that if the relational database approach was to work we would need to hire a full time (and expensive) DBA (which had never been in our business plan), we knew we had to change direction.

Making the change

Looking back, we worked really hard to try to solve the problems with a relational database (we do not think it was MySQL at fault -- using any other RDBMS would have had a minimal material difference). We already knew, from our combined past experience, that a proprietary database of the size/speed we needed from the likes of an Oracle or IBM or Microsoft would be too expensive (and still require that DBA). The decision was taken -- after learning about and taking courses on MapReduce -- that the relational approach was not going to work for us.

We realized that by using Amazon's Elastic MapReduce we could bring up a Hadoop cluster as needed not only to process our data, but also to enable us to perform metrics and data warehousing without the need for an RDBMS. The new approach works. Queries go through in minutes instead of the hours they previously took.

What makes this even more attractive is that Amazon's Elastic MapReduce's APIs let you automatically spin up a cluster on n-number of nodes, run the specified JAR files on the data and then automatically take the whole cluster back down when finished; we only pay for what we use. It got even better. We do not even have to use a conventional database to power the Extrabux web site.

After the data processing in MapReduce, we index the data using a read-only indexing system called Lucene. Lucene actually does the indexing and provides the basic query syntax. We use Solr to sit on top of Lucene to provide an XML interface (into Lucene) and so provide additional features used by our search engine, such as 'faceted search'.

When a shopper makes a search on our web site, the shopper interacts with our PHP front-end which sends the shopper's search term to Solr. Solr then sends an XML formatted response --based on results obtained from the Lucene index -- back to the PHP front-end, with the search results being the products that match the search term. If the shopper then clicks on a particular product, a new query is sent from PHP to Solr -- this time for the details of the particular product being requested, This returns the detailed product information, the list of retailers who carry that product and their respective prices, etc. Due to the large number of products that our search engine comprises, the Solr index is 'sharded' across many servers, all hosted within the Amazon EC2 cloud. Our automation framework determines how many Solr shards are needed each day, based on the amount of data to be indexed, and we deploy each shard with a portion of the data.

To support this process, we leverage other Amazon Web Services such as SimpleDB, SQS (Simple Queue System), AutoScaling and S3, which allow us easily to:

- launch the shards once the data is ready,
- let each shard find a portion of the data to index
- record each shard's status and IP address (so that our front-end knows which shards to talk to when it needs to make a query.)

The beauty of this is that we have not had to create a conventional database for our bulk data (we continue to use MySQL for small-scale data such as merchant information). Everything is handled through non-traditional means, which also happens to be both faster and much, much cheaper.

Pitfalls and lessons learned

The biggest pitfall, if it can be described as this when we have such an ever changing and growing environment, is defining and automating the data flow so that new data is delivered, processed and indexed reliably every day,. We possess a constantly evolving system. All I have described is neither simple nor smooth in practical terms. Attention is always needed; however, we have spent many hours perfecting and automating our current process and we are confident that our system will perform reliably when our new comparison shopping site goes live, even under extreme load.

Our number one lesson learned was that we did not have a need for an RDBMS to process and store our large-scale data; everything we needed to do could be done with scalable open-source solutions. For example:

- much of our data warehousing is now done using Hive, an SQL-like query language that is built on top of MapReduce
- for data analysis we found that we are able to utilize Pig, a scripting language also built on top of MapReduce.

After briefly experimenting with MySQL, we found that using interfaces to MapReduce (like Pig and Hive) meant, we were able to obtain much better results much more easily and cost effectively.

We are now in our fourth major revision of the MapReduce code -- and moving towards a service-oriented architecture (SOA). Now that we have a good understanding of how the different components in our system interact with each other (the coupons, the cash backs, the merchant information, etc.), we are starting to work on making MapReduce as well as our front-end talk to our web services. The various components of our system are nearing feature completion and we have a clear picture of how the components need to interact with each other – which is what makes the SOA approach possible, albeit at the expense of having to re-engineer some of our code.

Although we found that MapReduce is a more sensible solution for Extrabux, is still a relatively new tech-

nology and its documentation and best practices are not as fully developed as for RDBMS systems. Indeed, we had to build our system the hard way -- by trial and error. We learned, for example, that it is often impractical to link our MapReduce jobs directly to other data sources, such as Solr; we had to develop workarounds. Similarly, Solr can be 'touchy' sometimes: a small irregularity in the data can cause it to break (and its administration is less than user-friendly).

Another area we needed carefully to address when developing for the cloud is planning for node failures and building a self-healing and robust system architecture. This is not, of itself, a major problem – but it required us to think about problems differently,. We now think of each server instance as doing only one thing and as being fully self-contained so it can be re-launched automatically in case it fails, without disrupting the system. In addition, we needed to design our system to utilize the underlying Amazon Web Services that are available in addition to EC2 --such as S3, Elastic Block Store, SQS, SimpleDB, Elastic Load Balancing, Auto Scaling, etc. That said, bandwidth between nodes and between different services such as S3 is a continuing source of challenge and requires great care and planning, otherwise jobs can take far too long to process.

Yet, for $10/day for all the processing described, the pitfalls and lessons learned seem a good trade-off.

So I guess my final lesson learned is that, even if you do have 40GB/day of data input to process, you do not have to spend a fortune on traditional software. There are alternatives available, especially when working in the cloud, that work much better and are more flexible and cost-effective if you are able to invest some time in building a custom data processing solution.

Management conclusion

40GB of data processed daily (from load to indexing of 70M constantly changing input records) for US$10/day is astonishing value. That is less than US$5K/year – a trivial % of what a traditional RDBMS would have cost for maintenance alone.

Extrabux demonstrates how powerful and effective Hadoop is. It also confirms that the N-LLS approach is valid and why it will be increasingly attractive.

Patrick Salami
Senior Software Architect
Extrabux
(www.extrabux.com)