Big Data: Building a Document Index From Web Crawl Archives

Usman A. Shami, Founder, #Let'sData
usmanshami@letsdata.io

1. Abstract

In this big data case study, we processed the Common Crawl Web Archives files using the #Let's Data compute to reduce the web archives to JSON documents that could be used to create a database index. We processed ~ 219K files, ~477 TB S3 data in 48 hours at a nominal cost of $5 per TB-Hour.

2. Problem Definition

Big Data datasets are generally huge datasets spread across a large number of files. Processing these files at high scale and in a reasonable amount of time requires creating a data pipeline which can be a significant engineering infrastructure effort, is rife with infrastructure costs and can take many man-months to build and perfect.

In this Big Data use-case, we want to process the Common Crawl[1] Web Archives files (219K files, 477TB uncompressed data) and transform this semi-structured data to structured JSON documents that can be used to create a database index. Such an effort would require an understanding of the data domain (Common Crawl Web Archive Formats), infrastructure challenges such as reliable and fault tolerant compute infrastructure, maintainability, operability and non – trivial compute code. For example, the compute code needs to deal with transient failures, rate limiting, batching for performance, ordering and deduplication, checkpointing and failure restarts etc. Logging, Metrics and Diagnostics infrastructure need to be built in.

Building such a dataset pipeline on #Let's Data eliminates these infrastructure requirements – the #Let's Data promise is that the enterprises should "Focus on the data, we'll manage the infrastructure". We used #Let's Data to process the Common Crawl Web Archives dataset – the system processed 219K files, ~477 TB of data in ~ 48 hours and extracted ~3 billion JSON documents[2] – this roughly translates to a TPS of 17K documents per second!

#Let's Data simplified the creation and management of this data compute pipelines using AWS services, reduced the development time, costs, enabled high performance, availability, and elastic scale.

3. Solution & Architecture

There are two different types of development efforts needed for such a Big Data use-case:

1. **The Functional Data Model:** Understanding the data formats for the big data functional domain and developing how to parse the data and extract output documents.
2. **The Data Pipeline Infrastructure:** This is the infrastructure code that is required to orchestrate the data pipeline, reading from the source, writing to the destination, scheduling computation tasks and data jobs, tracking errors and building in fault tolerance and the necessary diagnostics.

In traditional data pipeline development, one would spend a disproportionately large development effort in developing the data pipeline infrastructure. With #Let's Data, the focus is mostly on developing the functional data model, with only an integration effort to orchestrate and run the data pipeline.
Let's look at each of these development efforts in detail.

3.1 Common Crawl Data Model

The Common Crawl Dataset has the following characteristics:

- It has three filetypes the Archive, Metadata and Conversion files
- each data record (crawled link) has data that is spread across these three files:

---

[1] Common Crawl is an open repository of web crawl data and a fantastic resource for the www web crawl data. https://commoncrawl.org/

[2] To put the 3 billion number into perspective, Google processes around 8.5 billion searches per day https://www.oberlo.com/blog/google-search-statistics

    - the archive file has the http request and response with some high level metadata
    - the metadata file has the metadata about the records in the archive file such as record types and their record offsets etc.
    - the conversion template has the converted Html document
- each of these files follows a record state machine for each data record (crawled link) – for example,
    - the archive file state machine is REQUEST -> RESPONSE -> METADATA for each crawled link
    - the metadata file state machine is METADATA (Request) -> METADATA(Response) -> METADATA(Metadata) for each crawled link (remember that this is metadata about the archive file records)
    - the conversion file state machine is simple – a single CONVERSION record for each crawled link

With this high-level information, we do the following development tasks:

- **The POJOS:** create Java POJOs that map to each record type – this is the majority of the work, where you define how to create an object from a byte array and validating the integrity of the object.
- **The Parsers:** define a parser state machine for each of the file using the #Let's Data interfaces – this is relatively simpler, you encode the record types as a state machine and specify the start and end delimiters for each records
- **The Reader:** define a reader that constructs an output document from these file parser state machines using the #Let's Data interface – this is the simplest of the three, encode the record retrieval logic from the parsers and then construct an output record by combining the these.

We've shared our implementation of the common crawl model at the Git Hub repository:
https://github.com/lets-data/letsdata-common-crawl

3.2 #Let's Data Data-Pipeline

With the above common crawl data model, we can now simply orchestrate the data pipeline by specifying the dataset configuration. We'd be creating a pipeline that reads the common crawl dataset files from AWS S3, writes them to AWS Kinesis and uses AWS Lambda to run the parser and extraction code. We also do some access setup so that #Let's Data can automatically manage the read and write resources.

Here are the dataset configuration details:

- Read Connector configuration:
    - the S3 Bucket to read from
    - the JAR file that has the #Let's Data interface implementations
    - the mapping of #Let's Data interfaces to file types (archive file type -> archive file parser class name etc.)
- Write Connector Configuration
    - the Kinesis stream that we need to write to
    - the number of shards for the Kinesis stream
- Error Connector Configuration
    - the S3 Bucket to write the error records to
- Compute Engine Configuration
    - AWS Lambda compute details – these are the function concurrency, timeout, memory and log level
- Manifest file
    - the manifest file that defines the list of all the files that should be processed and their mapping – example:

```
Archive : file1.archive | Metadata : file1.metadata | Conversion : file1.conversion
Archive : file2.archive | Metadata : file2.metadata | Conversion : file2.conversion
```

…

- Each line in the manifest file becomes a #Let's Data task that can be tracked from creation to completion and has its own progress, errors and diagnostics tracing.

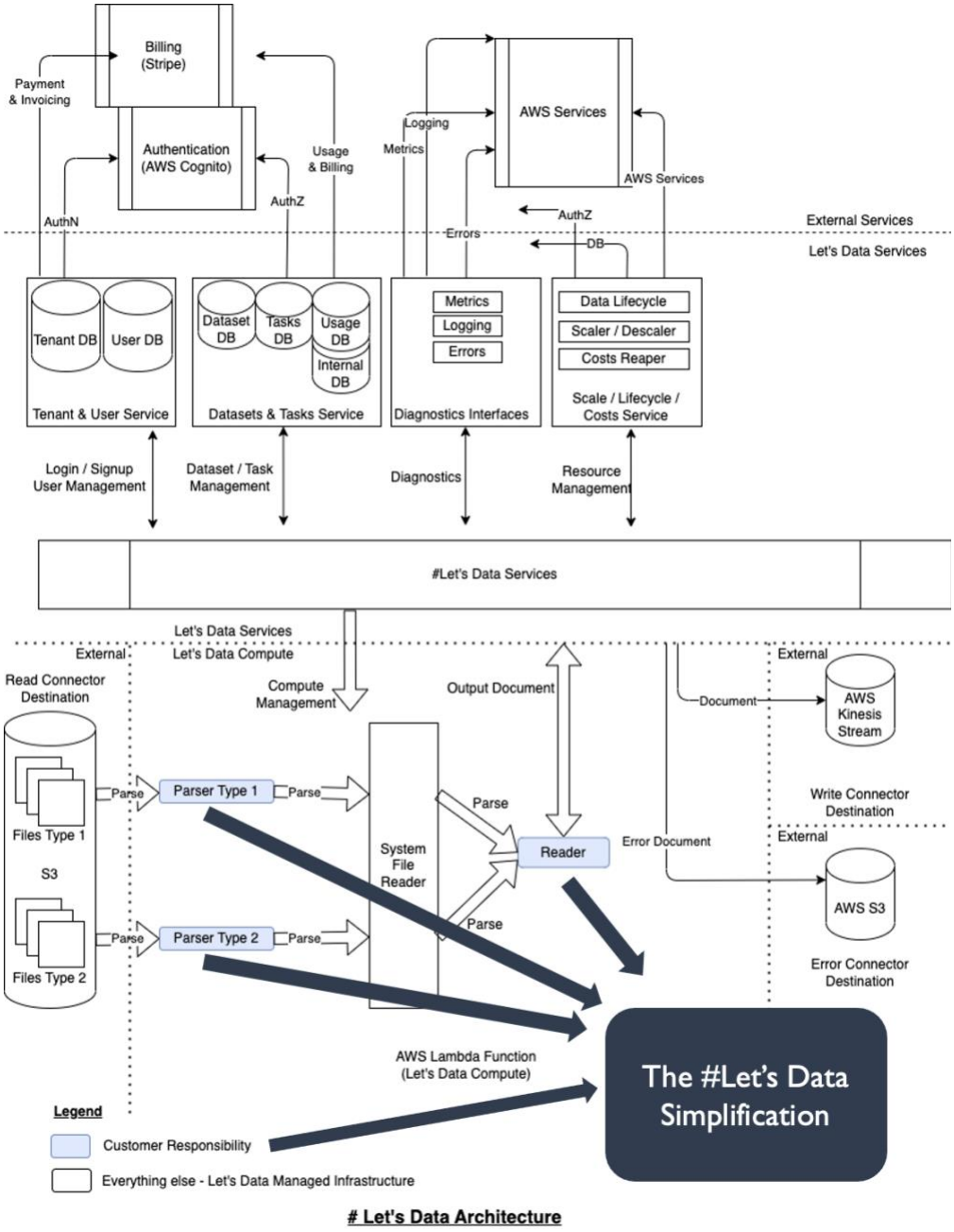
We use the #Let's Data CLI to create this dataset and monitor its execution via the CLI and Console.

```
# create the dataset
$ > ./letsdata datasets create --configFile dataset.json --prettyPrint

# view the dataset, monitor its creation
$ > ./letsdata datasets view --datasetName <datasetName> --prettyPrint

# list the datset's tasks
$ > ./letsdata tasks list --datasetName <datasetName> --prettyPrint
```

Here is the overall architecture of the data pipeline solution and the components (shaded in blue) that the customer was required to build.



 # Let's Data Architecture

Results

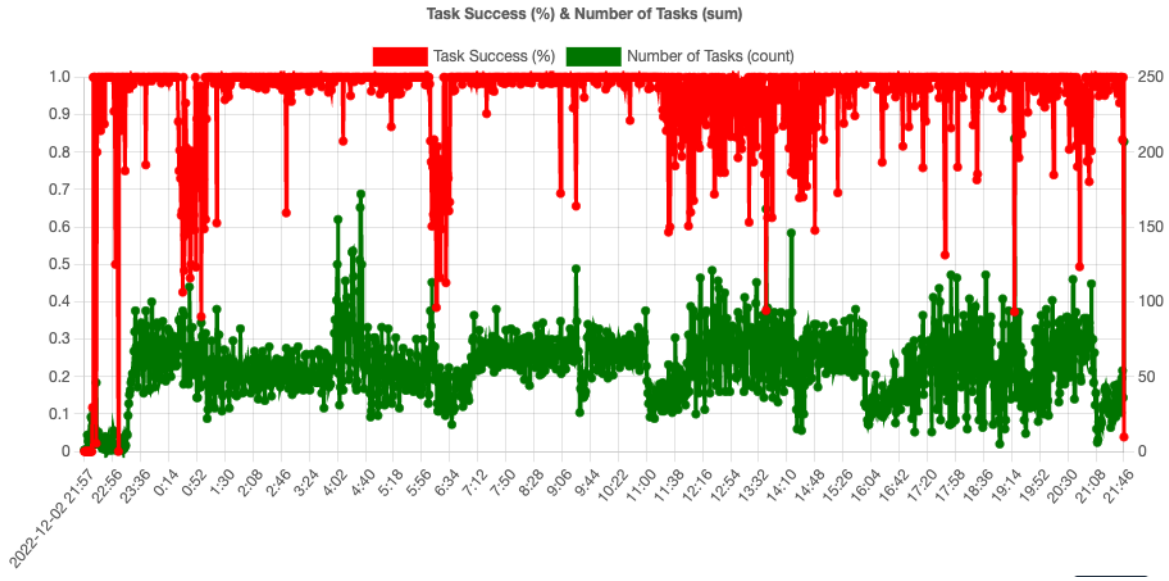
We ran this common crawl use-case on #Let's Data to test the limits of our infrastructure and were pleasantly surprised by the staggering scale we were able to achieve at nominal costs. Here are some results at a glance:

- The system processed 73K tasks (73K x 3 files) in ~ 48 hours
- Tasks executed on AWS Lambda with a concurrency of 500 parallel tasks
- 219K files processed, read 477 TB of uncompressed data from S3, wrote 13 TB to AWS Kinesis.
- Extracted ~ 3 billion records[3] that were written to AWS Kinesis Stream. ~16 million error records were written to AWS S3 as error records (0.5 % errors)
- The system peaked at reading 455 GB per minute from S3 and writing 12.36 GB per minute in AWS Kinesis, extracting 2.7 million records per minute (~45K records per second!)

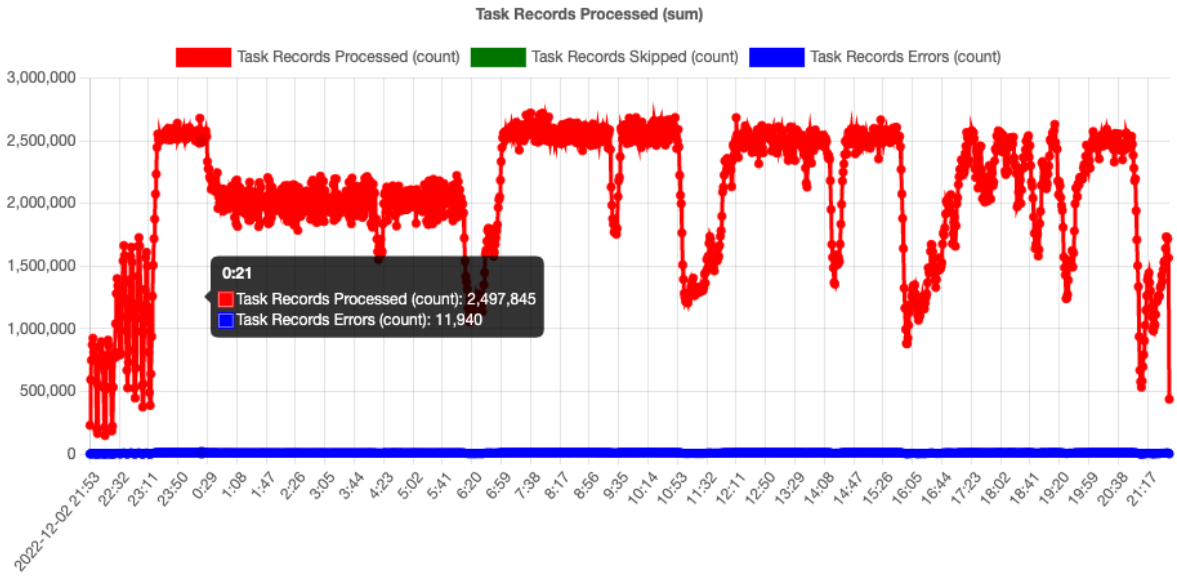
[3] To put the 3 billion number into perspective, Google processes around 8.5 billion searches per day https://www.oberlo.com/blog/google-search-statistics

- The costs for the dataset were $36,000 - approximate cost of $75 for each TB (uncompressed) and a $1.67 per TB-Hour

Here are some cool graphs to go with these results:

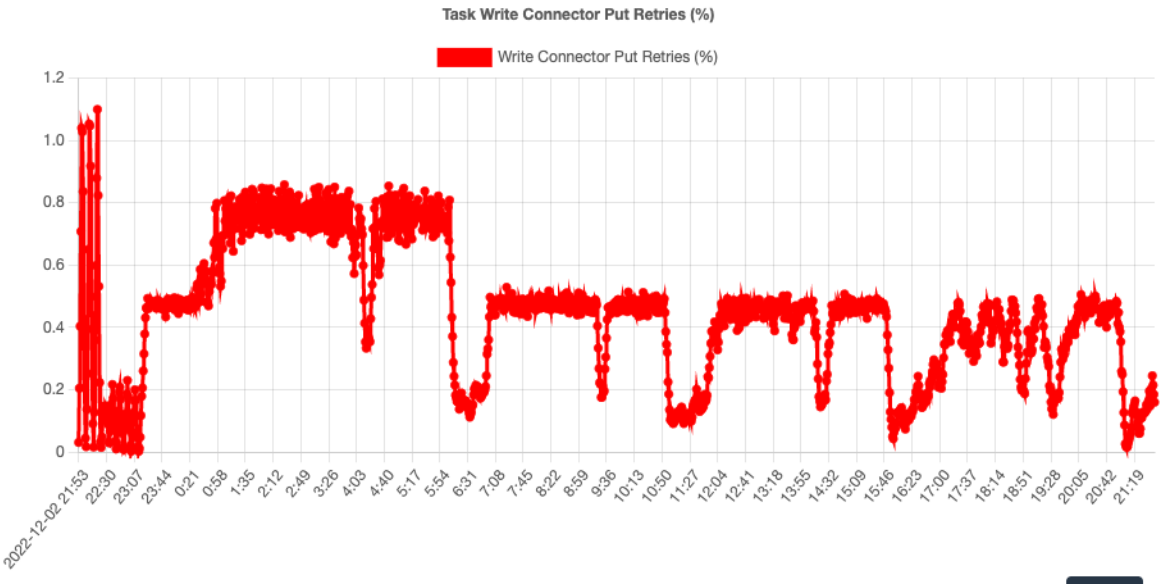


- Left (red): the percentage of tasks that completed successfully at that minute
- Right (green): number of tasks that completed at that minute

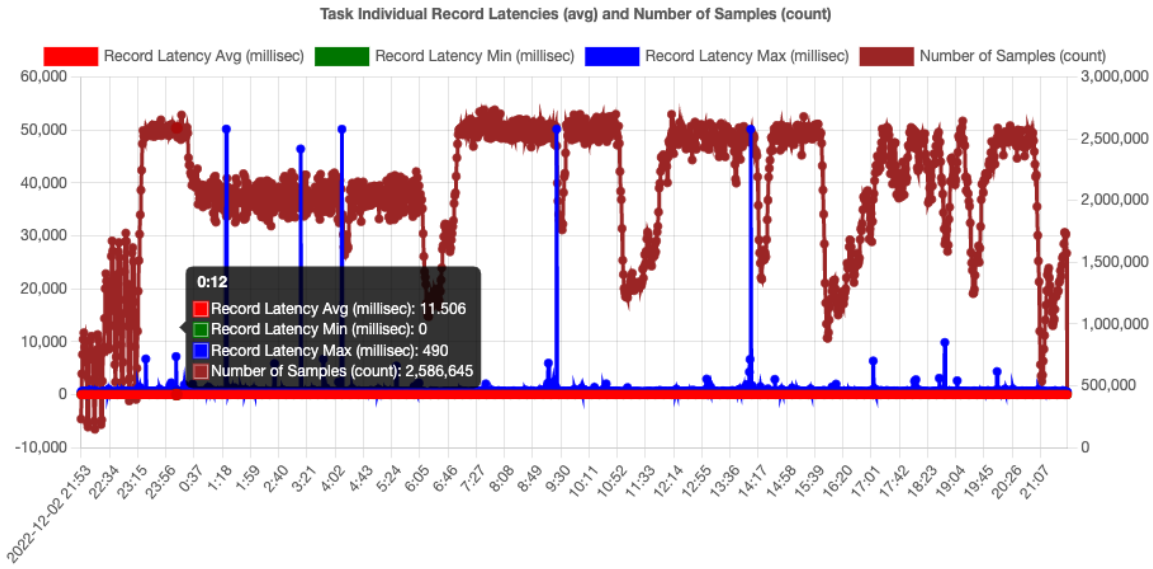


- Left: the sum of the number of records processed (red), skipped (green) and errored (blue) by each task at that minute

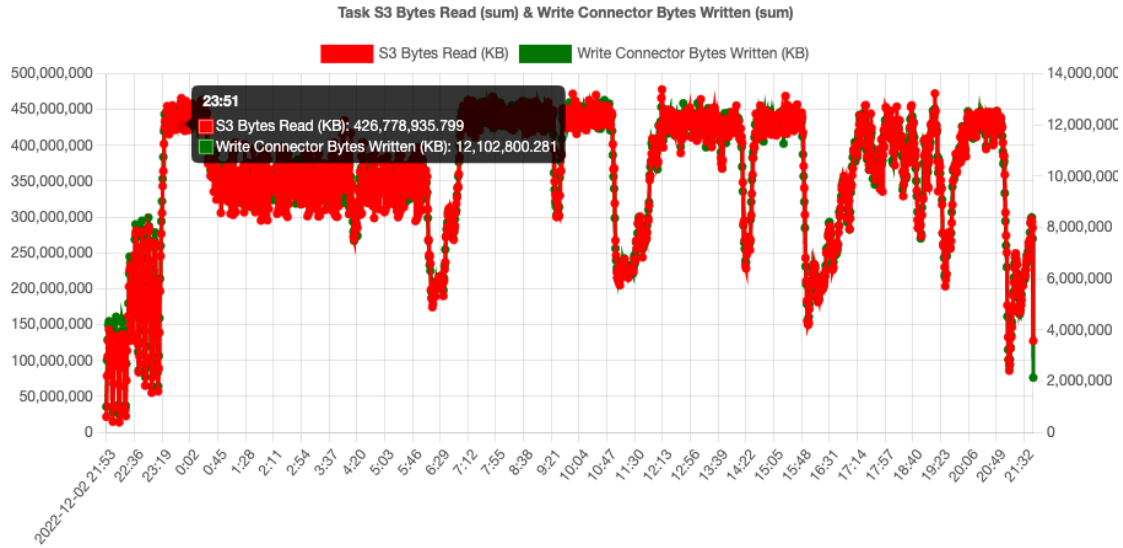
- Left (red): the sum of the Write Connector's Put API call by each task at that minute
- Right (green): the average latency of the Write Connector's Put API call
- Each Put API call is a batch call



- the average percentage of the Write Connector Put API calls that were retried by the tasks
- The Kinesis stream was underscaled in this test – ideally this should be zero (no retries). We could have scaled the Kinesis shards higher and gotten better results!



- Left: the (avg, min and max) latency of extraction of the record by the user handlers (readers and parser).
- Right: the sample count for the latency metric.



- Left (Red): the bytes read by the readers from S3 (in KBs) each minute
- Right (Green) the bytes written by the task to Write Connector (in KBs) each minute

TCO Analysis

Developing on the #Let's Data infrastructure has huge cost and time savings – here is a side by side comparison:

	Traditional Pipeline Development	**#Let's Data Pipeline**
S3 Read Module – Development	2 dev weeks	0
S3 Read Module – Integ. Test	2 dev week	0
S3 Read Module – Perf	2 dev week	0
Kinesis Module – Development	3 dev weeks	0
Kinesis Module – Integ Test	2 dev weeks	0
Kinesis Module – Perf	2 dev week	0

Lambda Module	3 dev weeks	0
Lambda Module – Integ Tests	2 dev weeks	0
Lambda Module – Perf	2 dev weeks	0
Task Management module	4 dev weeks	0
Resource Management	2 dev weeks	0
Error Records Handling	3 dev weeks	0
Logging	1 dev week	0
Metrics	1 dev week	0
Functional Domain Data Model	3 dev weeks	3 dev weeks
Let's Data Interfaces		1 dev week
State Machines	1 dev week	0 (included above)
Integration Tests	3 dev weeks	1 dev week
Prod Run – Devops support	3 dev weeks	2 dev week
Read Kinesis docs	1 dev week	1 dev week
Total	**42 dev weeks**	**8 dev weeks**

This is a 5X reduction!

Lessons Learned

This case study validated our engineering MVP – we can process large datasets at scale with a large reduction TCO. The case study also did find issues that we fixed:

- **Simplified dataset configuration** removing redundant / not needed fields
- **Schema fixes** where data partitioning was not working effectively for large datasets
- **Initialization workflow fixes** – 80K task ingestion caused our initialization workflow to timeout
- **Enable Log Levels** – verbose logging was enabled during the test run which resulted in a larger than expected amount of logs (and costs)
- **Token Sizes** – We generate a of tokens dynamically on each API call - with 80K tasks, the difference between a 1024 bytes pagination token vs a 512 bytes pagination token quickly adds up - we were hitting API response size limits and such reductions doubled the number of results we could return in each page.
- **Manually tweaked** the AWS Kinesis Stream's shard scaling during the run which accounts for difference in throughput and latencies during the run – need to write a dynamic optimizer (scaler / descaler)
- This was the first real large scale test of the system - while the system performed really well, we made a large number of fit and finish fixes across the stack