$ whoami

• Data Scientist @NASA JPL

• Passion for Information Retrieval and Web Search with emphasis on Federated Search.

• You can find me on a number of mailing lists including, Nutch, Gora, Any23, Tika, Open Climate Workbench, OODT, Usergrid, CommonsRDF (Incubating), Joshua (Incubating) and Htrace (Incubating) …

• 2nd ApacheCon in Vancouver! Great to be back.
So what are we covering?

• Overview of technologies

• Case Study – Implementing tracing within distributed Web search software

• Development tracking & future work

• Conclusion and recap

If you have a question, please interrupt at any stage. I am happy to take questions throughout the presentation and after.
What is Nutch?

“Highly extensible, highly scalable Web crawler”
Nutch is a well matured, production ready Web crawler. Nutch 1.x enables fine grained configuration, relying on Apache Hadoop™ data structures, which are great for batch processing.

“Pluggable parsing, protocols, storage and indexing”
Being pluggable and modular of course has it's benefits, Nutch provides extensible interfaces such as Parse, Index and ScoringFilter's for custom implementations e.g. Apache Tika™ for parsing. Additonally, pluggable indexing exists for Apache Solr™, Elastic Search, SolrCloud, etc.

The Nutch 2.X branch is an active development alternative taking direct inspiration from 1.X. 2.X differs in one key area; storage is abstracted away from any specific underlying data store by using Apache Gora™ for handling object to persistent data store mappings.
What is HTrace?

“Open source framework for distributed tracing”

Designed for use in big distributed systems such as the Hadoop Distributed Filesystem and the Apache HBase™ storage engine. However, there is nothing Hadoop-specific about HTrace. It has no dependencies on Hadoop, and is a useful building block for many distributed systems.

HTracing allow end-users to trace their requests. In addition, any other project that uses HTrace can follow the requests it makes to your project. That’s why we say HTrace is “end-to-end.”


Also See Colin McCabe’s “Introducing Apache HTrace” presentation @SCaLE - https://youtu.be/t-TwCLwYIGE
Technology Overview

Documentation for all Gora Datastore support can be found at
http://gora.apache.org/current/index.html#gora-modules
Why Target the Fetcher?

- Multithreaded
- The Fetcher is fairly complex in terms of both code and structure. Tracing is an excellent way for the community to better understand what the Fetcher is actually doing e.g. why you fetch the content that ends up in your CrawlDB!
- Because Nutch is a native Hadoop application, FetcherThread-s are executing on different machines. This is exactly what HTrace is designed for.
- Currently, no sampling data exists for the Nutch Fetcher... the hope is that community sampling can possibly further improve Fetcher performance.
Decomposition of the Nutch (2.x) Queue-based Fetcher

This fetcher uses a well-known model of one producer (a QueueFeeder) and many consumers (FetcherThread-s).

FetcherMapper
Reads a random integer written by GeneratorJob as its key while outputting the actual key and value arguments through a FetchEntry instance. This approach (combined with the use of partitioning by host) makes sure that Fetcher is still polite while also randomizing the key order. If one host has a huge number of URLs in your table while other hosts have not, FetcherReducer will not get stuck on one host but process URLs from other hosts as well.

FetcherReducer
QueueFeeder reads input fetchlists and populates a set of FetchItemQueue-s, which hold FetchItem-s that describe the items to be fetched. There are as many queues as there are unique hosts, but at any given time the total number of fetch items in all queues is less than a fixed number (currently set to a multiple of the number of threads).
Decomposition of the Nutch (2.x) Queue-based Fetcher Cont’d

As items are consumed from the queues, the QueueFeeder continues to add new input items, so that their total count stays fixed (FetcherThread-s may also add new items to the queues e.g. as a results of redirection) - until all input items are exhausted, at which point the number of items in the queues begins to decrease. When this number reaches 0 fetcher will finish.

This fetcher implementation handles per-host blocking itself, instead of delegating this work to protocol-specific plugins. Each per-host queue handles its own "politeness" settings, such as the maximum number of concurrent requests and crawl delay between consecutive requests - and also a list of requests in progress, and the time the last request was finished.

As FetcherThread-s ask for new items to be fetched, queues may return eligible items or null if for "politeness" reasons this host's queue is not yet ready. If there are still unfetched items in the queues, but none of the items are ready, FetcherThread-s will spin-wait until either some items become available, or a timeout is reached (at which point the Fetcher will abort, assuming the task is hung).
Case Study – Implementing tracing within Apache Nutch

HTrace prerequisites…
• Spans
• Scopes
• Tracers
• Wrappers
• Span Receivers
• Configuration
• Tracer Pool
• Passing Span ID’s over RPC
• Tracing + the Multithreaded Fetcher

See Colin McCabe’s presentation @SCaLE - https://youtu.be/t-TwCLwYIGE
HTrace Spans

A HTrace Span represents a length of time
• Description
• Unique Identifier (128bit)
• Start Time
• End Time
• Parents
• Process ID and IP address
• Arbitrary Time Annotations
• Arbitrary Key/Value Annotations

Trace Spans can generate A LOT of data… typically <1% of tracing is retained and referred to as sampling
/* Here we create an HTrace Tracer object. */

/* It is difficult to trace every operation. The volume of trace span data that would be generated would be extremely large! We therefore typically rely on sampling a subset of all possible traces. */

/* Tracer objects contain Samplers. When you call Tracer#newScope, the Tracer will consult that Sampler to determine if a new span should be created, or if an empty scope which contains no span should be returned. Note that if there is already a currently active span, the Tracer will always create a child span, regardless of what the sampler says. This is because we want to see the complete graph of every operation, not just "bits and pieces." Tracer objects also manage the SpanReceiver objects which control where spans are sent. */

/* A single process or library can have many Tracer objects. Each Tracer object has its own configuration. One way of thinking of Tracer object is that they are similar to Log objects in log4j. Just as you might create a Log object for a Hadoop NameNode and one for the DataNode, we create a Tracer for the NameNode and another Tracer for the DataNode. This allows users to control the sampling rate for the DataNode and the NameNode separately. */

/* Unlike HTraces TraceScope and Span, Tracer objects are thread-safe. It is perfectly acceptable (and even recommended) to have multiple threads calling Tracer#newScope at once on the same Tracer object. */

/* The number of Tracer objects you should create in your project depends on the structure of your project. Many applications end up creating a small number of global Tracer objects, in the case of the Nutch Fetcher, we create one Tracer and use it in a thread safe manner within the Fetch phase of a crawl cycle. */

public Tracer tracer = new Tracer.Builder("FetcherTracer").build();
HTrace Scopes

TraceScope objects manage the lifespan of Span objects. When a TraceScope is closed... so is the Span associated with it.

```java
@Override
public Map<String, Object> run(Map<String, Object> args) throws Exception {
    //Create a preliminary TraceScope merely for checking the basic
    //Fetcher configuration. Any trace spans created inside
    //checkConfiguration() will automatically have the CheckConfiguration
    //trace span we have created here as their parents. We don`t have
    //to do any additional work to set up the parent/child relationship
    //because the thread-local data takes care of it.
    TraceScope configurationScope = tracer.newScope("CheckConfiguration");
    try {
        checkConfiguration();
    } finally {
        configurationScope.close();
    }
    String batchId = (String) args.get(Nutch.ARG_BATCH);
    Integer threads = (Integer) args.get(Nutch.ARG_THREADS);
```
HTrace Span Receivers

Due to the pluggable nature of the HTrace SpanReceiver implementations, we can chop and change which collector is collecting from within the Fetcher process. LocalFileSpanReceiver (available within htrace-core4) default implementation.

<!-- Beginning of Tracing Dependencies -->
<dependency org="org.apache.htrace" name="htrace-core4" rev="4.1.0-incubating" conf="*->default" />
<!-- HTrace SpanReceivers, by default we use LocalFileSpanReceiver which writes the spans it receives to a local file. Other SpanReceivers can be used by activating one of the dependencies below and configuring nutch-site.xml appropriately. -->
<dependency org="org.apache.htrace" name="htrace-htraced" rev="4.1.0-incubating" conf="*->default" /-->
<dependency org="org.apache.htrace" name="htrace-flume" rev="4.1.0-incubating" conf="*->default" /-->
<dependency org="org.apache.htrace" name="htrace-hbase" rev="4.1.0-incubating" conf="*->default" /-->
<dependency org="org.apache.htrace" name="htrace-zipkin" rev="4.1.0-incubating" conf="*->default" /-->
<!-- End of Tracing Dependencies -->

The relevant SpanReceiver can be activated through HTrace Configuration.
HTrace Configuration

In particular we are interested in providing a default ‘htrace.span.receiver.class’ option within nutch-default.xml which can be overridden in nutch-site.xml, accessed from NutchConf and utilized within the HTrace Tracer constructor. Other Configuration will most likely be added as a better understanding of Trace sampling is achieved.

```java
Tracer tracer = new Tracer.Builder().
    name("FetcherTracer").
    tracerPool(new TracerPool("WriteFetcherTraceToLocalFile")).
    conf(HTraceConfiguration.fromKeyValuePairs(
        "span.receiver.classes", conf.get("htrace.span.receiver.class"),
        "local.file.span.receiver.path", traceFileName,
        "tracer.id", "{%tname}%")).
    build();
TraceScope scope = tracer.newScope("WriteFetcherTraceToLocalFile");
```
Recent developments within the tracing community have brought projects like Apache HTrace (Incubating) into the spotlight. The possibility of utilizing tracing logic to better understand distributed applications, systems and systems-of-systems. HTrace involves a specialized use of logging to record information about a program’s execution. Although many use tracing within distributed systems such as Hadoop and databases, few tracing experiments belong within the field of Web search.

This issue will combine comprehensive tracing mechanisms in Apache HTrace (Incubating) with the scalable, flexible build tools.
Development Tracking + Future Work

• Implement a new HTrace Tracer for other NutchJob’s e.g. IndexingJob, ParserJob, etc. Tracing of the FetcherJob seemed like the most appealing to me however there are other similarly interesting areas to be explored within Nutch.

• In particular, HTrace (pre-Apache) exists as a dependency within lucene-solr, we could implement inter-application tracing within the Nutch -> Solr indexing workflow.

• …
Thank you all… very much
Enjoy the week ahead and everything Vancouver has to offer.

Find me on Apache lists

lewis.j.mcgibbney@jpl.nasa.gov

lewismc@apache.org

@hectorMcSpector