Quantifind’s story: Building custom interactive data analytics infrastructure

Ryan LeCompte
@ryanlecompte

Scala Days 2015
Background

• Software Engineer at Quantifind
• @ryanlecompte
• ryan@quantifind.com
• http://github.com/ryanlecompte
Outline

• What does Quantifind do?
• Technical challenges
• Motivating use cases
• Existing solutions that we tried
• Custom infrastructure
• Lessons learned
What does Quantifind do?

- Find intentful conversations correlating to a customer’s KPI and surface actionable insights

- Input
  - consumer comments (Twitter, Facebook, etc)
  - 3rd-party financial data

- Output
  - top-level actionable insights
  - interactive exploration
Technical Challenges

• Operate on a multi-terabyte annotated data set containing billions of consumer comments from millions of users over thousands of different dimensions

• We want to slice and dice & compute over any dimension

• We want to perform user-level operations; not everything can be satisfied by a pre-computed aggregation
Use Cases

• Flexible time series generation
• N-gram co-occurrences
• Cohort analysis
• Generate a time series of consumer conversations

• Flexible/arbitrary binning (year, month, day, hour, minute, second) with user de-duping

• Generate time series over data matching specific criteria (e.g., text/terms, other dimensions)
N-gram Co-occurrences

• N-grams are sliding groups of N adjacent words in a document

• We wish to find all n-grams that co-occur with a particular search term or phrase over an arbitrary time period or other dimension

• “I am hungry right now”

  • 1-grams: I, am, hungry, right, now
  • 2-grams: I am, am hungry, hungry right, right now
  • 3-grams: I am hungry, am hungry right, hungry right now
Cohort Analysis

• Capture a particular group of users satisfying certain conditions at some point in time

• Next, for those same users analyze their conversations after a certain event happens

• Example:
  • Capture users commenting that they want to see the movie Gone Girl before it’s released
  • Analyze new conversations for only this cohort after the movie’s release date
Previous Approaches

- Spark
- Postgres
- Elastic Search
Spark

• Experienced challenges when trying to compute certain n-gram co-occurrences for the entire data set on our 12 node cluster

• Tough to reduce the search space of the entire data set without manual work

• Challenging developer experience
  • Contention for cluster resources
  • Re-packaging Spark JARs after code changes
Postgres

- Tables are partitioned by vertical/time
- Query performance suffers greatly when concurrent and disparate requests are being processed
- Managing table partitions is painful
- Limited to SQL-style operations; tough to bring custom computation close to the data
Elastic Search

- Challenging to do summary statistics without the need to load columns into memory
- Rebuilding indexes can be annoying
- Still tough to build custom logic that operates directly on the data
What do we really want?

- We really want to access our raw data in memory to create flexible functionality (e.g., time series, cohort analysis, and n-gram co-occurrences)

- We want the operations to be on-the-fly, optimized, and reasonably fast (no more batch jobs)

- So, let’s take our data and “pack” it into a very compact binary format so that it fits in memory!
Can’t fit everything in a single machine

• We can shard the compacted data in-memory across multiple machines and communicate via Akka Cluster

• Make scatter/gather-style requests against the raw data and compute results on the fly
RTC: Real-time Cluster
RTC Components

- Pack File Writer
- Master
- Worker
RTC Pack File Writer

- Off-line job that consumes the raw annotated data set
- Packs the consumer comments and users into the custom binary protocol/format (.pack files)
- Packs the data such that it’s easily distributable across the RTC nodes
RTC Master

• Handles incoming RTC requests and routes them to the appropriate scatter/gather handler
• Distributes groups of .pack files to RTC workers once they register with the master (via Akka Cluster)
• Caches frequently accessed permutations/requests
RTC Worker

- Discovers and registers with the RTC master (via Akka Cluster)
- Loads its assigned .pack files from the filesystem and stores the data in memory
- Handles delegated requests from the RTC master
Akka Cluster

- Makes it super easy to send messages between actors running on different machines
- Allowed us to focus on our core use cases without having to worry about the network layer
- Facilitates distributed scatter/gather style computations across multiple machines
Compact Data

- Custom binary protocol for consumer comment and user records
- Each record has a fixed header along with a set of known fields and values
- User and comment records are packed separately; joined on the fly
Custom Protocol

- Certain fields are 64-bit SIP hashed vs. storing full text
- Fields that are one of N values are stored in a smaller type (e.g. Byte / Short)
JVM Garbage Collection

• Not explicitly managing memory & worrying about allocations is fantastic for most JVM development

• However, the garbage collector can be your enemy when trying to manage a lot of objects in large JVM heaps

• GC pauses greatly affect application performance
Off-heap Memory

- JVM gives us access to a “C/C++ style” means of allocating memory via `sun.misc.Unsafe`
- JVM does not analyze memory allocated via Unsafe when performing garbage collection
- Memory allocated via Unsafe must be explicitly managed by the developer, including deallocating it when no longer needed
RTC: Real-time Cluster

HTTP API Handler → RTC Master → RTC Worker

Off-heap data

RTC Worker

RTC Worker

RTC Worker
Working with off-heap data

// skip header and user id, only extract timestamp
def timestamp: Long = unsafe.getLong(2 + 8)

• Primary goal: never materialize data that you don’t need in order to satisfy the incoming request

• We never materialize a full record; instead, we only access the fields that are needed for the given incoming request (e.g. time series, n-grams, etc)
def binarySearch(
    unsafe: Unsafe,
    offset: Long,
    fromIndex: Int,
    toIndex: Int,
    searchTerm: Long): Int = {

    var low = fromIndex
    var high = toIndex - 1
    var search = true
    var mid = 0

    while (search && low <= high) {
        mid = (low + high) >>> 1
        val term = unsafe.getLong(offset + (mid << 3))
        if (term < searchTerm) low = mid + 1
        else if (term > searchTerm) high = mid - 1
        else search = false
    }

    if (search) -(low + 1) else mid
}
The Search Space

- Each worker maintains various arrays of offsets (facet indices) to records stored in off-heap regions.
- When a request comes in, the worker will determine which offsets it needs to visit in order to satisfy the request.
- There can potentially be hundreds of millions of offsets to visit for a particular request; we need to visit as quickly as possible.
RTC: Real-time Cluster

- **RTC Master**
  - HTTP API Handler
  - RTC Worker
- RTC Worker
  - Facet indices
  - Off-heap data
  - Facet indices
  - Off-heap data
  - Facet indices
  - Off-heap data
Facet Index Example

- Most of our queries care about a particular time range (e.g., a year, month, weeks, etc)
- How can we avoid searching over data that occurred outside of the time range that we care about?
Time Facet Index

- Workers have the opportunity to build custom facet indices while loading .pack files
- We can build up a TreeMap[Long, Array[Long]]
  - Keys are timestamps
  - Values are offsets to off-heap records occurring at that time
- Range-style operations are $O(\log n)$
- We only process offsets that satisfy the date range
Concurrent Visits

Array[Long]

<table>
<thead>
<tr>
<th>Record Offset 0</th>
<th>...</th>
<th>Record Offset 500</th>
<th>...</th>
<th>Record Offset 1000</th>
<th>...</th>
<th>Record Offset 1500</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td></td>
<td>Thread 2</td>
<td></td>
<td>Thread 3</td>
<td></td>
<td>Thread 4</td>
<td></td>
</tr>
</tbody>
</table>

- Each worker divides up their off-heap record offsets into **ranges** that are visited concurrently by multiple threads
- Results are partially merged on the worker and then sent back to master for final merge
I hear about how immutable structures are awesome but here's the thing, you want millions of ops/second? You need to stop allocating memory.
Lessons Learned

- Reduce allocations as much as possible per-request
- Avoid large JVMs with many objects stuck in tenured generation
- Keep “expensive to compute” data around in LRU caches
- Protect yourself from GC pressure by wrapping large LRU cache values with scala.ref.SoftReference
- Chunk large requests into smaller requests to avoid having workers store too much intermediate per-request state
- Use Trove collections / primitive arrays
Was it worth it building a custom solution?

- Able to create optimized core RTC functionality since we own the entire system and operate on raw data that isn’t aggregated
- Able to adapt to new challenges and business requirements; no longer fighting existing open source solutions
- Future improvements include adding more redundancy/scalability, even more compact data representation, etc
Summary

Akka Cluster +
Non-aggregated off-heap data +
Fast Scala Code =
WIN
Thanks!
We’re hiring!

http://www.quantifind.com/careers/