CIS 455/555: Internet and Web Systems

XFilter

October 11, 2021
Plan for today

- Basic crawling ✔
- Mercator ✔
- Publish/subscribe ✔
- Streaming XML ✔
- XFilter
What If I Want XML Data (or XML with particular Data)?

May want to retrieve documents with particular patterns

- All articles
- Articles about sports
- etc.

Diagram:

- Crawler
  - “spout”
  - Filter on documents
    - “bolt”
Interest-based crawling

Suppose we want to crawl XML documents based on user interests

We need several parts:
- A list of “interests” – expressed in an executable form, perhaps XPath queries
- A crawler – goes out and fetches XML content
- A filter / routing engine – matches XML content against users’ interests, sends them the content if it matches
In a crawler or pub/sub system, we can think of there being a stream of documents:

But when parsing, the parser creates a stream of events per document...
Parsing Events w/ Simple API for XML (SAX)

```xml
  <channel>
    <title>NYT > Technology</title>
    <link>
      https://www.nytimes.com/section/technology?partner=rss&emc=rss
    </link>
    <description/>
    <language>en-us</language>
    <copyright>Copyright 2017 The New York Times Company</copyright>
    <lastBuildDate>Mon, 23 Oct 2017 12:41:40 GMT</lastBuildDate>
  </channel>
</rss>
```

StartElement: rss
StartElement: channel
StartElement: title
StartElement: link
EndElement: link
Text: NYT > Technology
EndElement: title...
XML Events and Bolts

(news.rss, startElement, rss)
(news.rss, startElement, channel)

news.rss

SAX Parser Bolt

Filter Bolt
How Might the Filter Bolt Look? A Simple State Machine!

startElement: rss?

startElement: channel?

startElement: title?

text: NYT > Technology?

Store and index the document with the corresponding ID
Plan for today

- Publish/subscribe ✅
- Streaming XML ✅
- XFilter
- Google File System
Suppose we want to use full-blown XPath to define our matches:

```
/rss[channel/title/text()="NYT > Technology"]
```

And we want to support millions of users.
Scaling up XPath Matching!

- XPaths → regular expressions over traversals from root → descendant
  - Can take the various events and use finite state machines (with some extensions)

- What if we are matching many paths at the same time?
  - Indexing, state management?
XML-Based information dissemination

Basic model (XFilter, YFilter, Xyleme):

- Users are interested in data relating to a particular topic, and know the schema
  - /politics/usa//body
- A crawler-aggregator reads XML files from the web (or gets them from data sources) and feeds them to interested parties
Engine for XFilter [Altinel & Franklin]
How does it work?

- Each XPath segment is basically a subset of regular expressions over element tags
  - Convert into finite state automata
- Parse data as it comes in
- Match against finite state machines
- Most of these systems use modified FSMs because they want to match many patterns at the same time
Path nodes and FSMs

- XPath parser decomposes XPath expressions into a set of path nodes
- These nodes act as the states of corresponding FSM
  - Path nodes are not generated for "*" nodes
- Simple FSM for `/politics[@topic=“president”]/usa//body:`

```
Q1_1          Q1_2
politics ◄—► usa ◄—► body
  Q1_3
```

© 2021 A. Haeberlen, Z. Ives, V. Liu
Decomposing into path nodes

Q1 = `/politics[@topic="president"]/usa//body`

<table>
<thead>
<tr>
<th>Q1-1</th>
<th>Q1-2</th>
<th>Q1-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
</tbody>
</table>

Query ID

Position in state machine

Relative Position (RP) in tree:
- 0 for root if it’s not preceded by “//”
- -1 for any node preceded by “//”
- Else = 1+ (no of “*” nodes from predecessor node)

Level:
- If current node has fixed distance from root, then 1+ distance
- Else if RP = -1, then -1, else 0

Finally, NextPathNodeSet points to next node

Q2 = `//usa/*/body/p`

<table>
<thead>
<tr>
<th>Q2-1</th>
<th>Q2-2</th>
<th>Q2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>-1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Indexing states by element names

- Query index entry for each XML tag
  - Candidate List (CL) of active states
  - Wait List (WL) of potential states
- Events that cause state transition are generated by the XML parser
  - Copy from WL → CL
  - Populate the Level and RelativePos

Diagram:
- **Politics**
  - Q1-1
  - Q2-1
  - Q1-2
- **USA**
  - Q2-1
  - Q1-2
- **Body**
  - Q1-3
  - Q2-2
- **P**
  - Q2-3
Encountering an element

- Look up the element name in the Query Index and all nodes in the associated CL
- Validate that we actually have a match

startElement: politics

Entry in Query Index:

Table:

<table>
<thead>
<tr>
<th>Q1</th>
<th>Level</th>
<th>Position</th>
<th>Rel. Position</th>
<th>NextPathNodeSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td>Q1-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>WL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>CL</td>
</tr>
<tr>
<td>politics</td>
<td>Q1-1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Validating a match

- We first check that the current XML depth matches the level in the user query:
  - If level in CL node is -1, then ignore height
  - else level in CL node must = height

- This ensures we’re matching at the right point in the tree!

- Finally, we validate any predicates against attributes (e.g., [@topic="president"])}
Processing further elements

- Queries that don’t meet validation are removed from the Candidate Lists

- For other queries, we advance to the next state
  - We copy the next node of the query from the WL to the CL, and update the level
  - When we reach a final state (e.g., Q1-3), we can output the document to the subscriber

- When we encounter an end element, we must remove that element from the CL
Many firewalls and intrusion detection systems need to do similar hierarchical matching over network packets.

Meme detection, composite event detection, etc.
Recap: Publish-subscribe model

- Publish-subscribe model
  - Publishers produce events
  - Each subscriber is interested in a subset of the events

- Challenge: Efficient implementation

- XFilter
  - Interests are specified with XPaths (very powerful!)
  - Sophisticated technique for efficiently matching documents against many XPaths in parallel
Plan for today

- XFilter
- Stream Processing
  - Apache Storm (and StormLite)
  - Sequence, windows, and state
  - Operations
Recall: The Basic Pieces of Storm

- **Spouts** connect to queuing systems, sensors, ... and convert / import streams of tuples

- **Bolts** do data processing – item at a time, window at a time, etc.
  - They may also access storage, or call external libraries
  - Behind the scenes, all messages are logged in order in a coordinator called Zookeeper

- **The topology** “glues” data together the components
  - Spouts + bolts are named nodes in a graph
  - Streams are named edges
  - “Grouping” specifies how to shard among multiple workers doing the same bolt
Apache Storm (and Its Successor Heron): Distributed Streams among Distributed Modules

Spouts interface with the world, produce **streams**

Bolts receive streams, optionally produce streams + read / update **state**

Streams of tuples
Executors

- Each storm node (operator) has an output **schema** defining the structure of the output stream elements.

- Each Storm node may have multiple **executors** – essentially, threads – and may even “shard” across multiple compute nodes.

- The topology specifies how executor output is routed to the executors in the next operator.
Distributed Apache Storm

- Storm is focused on a cluster environment!
  - One coordinator node (Nimbus)
  - And many workers

- Storm allows backup Nimbus nodes – this is through a coordination service called Zookeeper
public class RandomSentenceSpout implements IRichSpout {
    SpoutOutputCollector collector;
    Random rand;

    public void open(Map<String, Object> conf, TopologyContext context,
                      SpoutOutputCollector collector) {
        this.collector = collector;
        rand = new Random();
    }

    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[]{
            sentence("the cow jumped over the moon"),
            sentence("an apple a day keeps the doctor away"),
            ...
        };
        final String sentence = sentences[rand.nextInt(sentences.length)];
        collector.emit(new Values(sentence));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sentence"));
    }
}
Example Bolt: Split into Words

```java
public class SplitSentence implements IRichBolt {
    OutputCollector collector;
    public void prepare(Map<String, String> conf,
                         TopologyContext context,
                         OutputCollector coll) {
        collector = coll;
    }

    public void execute(Tuple tuple) {
        String[] words = tuple.getString(0).split(" ");
        for (String word: words)
            collector.emit(new Values(word));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```
Example Bolt: Aggregation

```java
public class WordCountBolt implements IRichBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();
    ...
    public void execute(Tuple tuple) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null) {
            count = 0;
        }
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));    // Was set in prepare()
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```
Topology and Groupings

```java
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(), 5);

builder.setBolt("split", new SplitSentence(), 8)
    .shuffleGrouping("spout");
builder.setBolt("count", new WordCountBolt(), 12)
    .fieldsGrouping("split", new Fields("word"));
```

Round-robin Send to executor based on hash of fields
Groupings in Storm

- fieldsGrouping – distribute to node based on hash value
- shuffleGrouping – round-robin
- allGrouping – broadcast
Storm vs StormLite

- **Apache Storm** – $O(100K)$ lines of code
  - Somewhat confusing as there are multiple versions of the API
  - Executors can be distributed across machines

- **StormLite** – CIS 455/555 re-implementation of the Storm “Rich” APIs
  - StormLite programs should be compatible with Storm
  - Doesn’t include reliability mechanisms; main program is the Nimbus node
  - You’ll extend so executors can be distributed across machines in HW3
Storm Promises

- Guaranteed at least once processing of every data item
  - With Trident layer, can guarantee exactly once

- Fail-over if a bolt or a spout dies (looks for acknowledgments)

- Fail-over if the main Storm coordinator node dies

- So: you can write programs with confidence that things will “just work”
Summary of Apache Storm

- Provides mechanisms for **incremental processing** of stream tuples
  - Spouts produce tuples incrementally
  - Bolts process tuples incrementally
  - Storm connects them with a “topology”
Plan for today

- XFilter
- Stream Processing
  - Apache Storm (and StormLite)
  - Sequence, windows, and state
  - Operations
General Patterns

- Storm(Lite) lets you implement almost anything in spouts/bolts...

- But are there general principles and patterns?
  - We’ll summarize some of them
  - Many have been implemented in higher-level stream engines and languages
Basic Stream Operators

- Filtering and transformation
- Sequential matching
- Windowed operators and aggregation
- Unions, joins and splits
Filtering and Transformation

- We can take tuples and either:
  - Emit them only if they satisfy conditions (filtering)
  - Emit modified versions (transformation)

- This can be as a single operator in the stream, or as part of the logic in another, more complex operator
Sequence

- We’ve seen sequential and hierarchical pattern matching of streams
  - XFilter!

- Implement using a state machine
  - Plus a stack or a list of active and inactive states
Windows and Bounded State

- Often we want to look at “recent behavior” – “last 5 minutes” or “last 50 results”

- A “tumbling window” or “sliding window”

- Append entries to a queue / list
  - Evict ones outside of our window

- Typically we then perform aggregation over the window
  - e.g., average, count, min, max
Stream Windows

(“tumbling” based on time)

GPS

\[ \text{now now now now now} \]
Stream Windows

("sliding")

GPS

now now now now now

time
Partitions

- Sometimes we have a stream with sub-streams, identified by the data
  - e.g., a stream has GPS readings from two different phones, indicated by phone ID

- We want to have two sliding windows – we accomplish this by partitioning on the ID
Stream Windows

(“sliding + partitions”)

$\text{GPS}$

1 1 1 1

2 2 2

now now now now now

time
Stream Unions and Joins

- We can have an operator that takes two input streams
  - If they are *identical in schema* it can output the tuples – forming a *union* of the streams
  - If they are *dissimilar in schema* it can add each to a window, and then *join* the windows

for each $r$ tuple in $r$\_window
  for each $s$ tuple $s$\_window
    if condition($r$, $s$) then
      collector.emit(concat($r$, $s$))
Stream Splitting

- In certain scenarios, we can send the output of an operator to multiple downstream operators.

- Useful for broadcast or even conditional sends (main stream + error stream).
Summary: Basic Stream Operators

- Filtering and transformation
- Sequential pattern matching
- Windows, partitions, and aggregation
- Unions and joins
- Splitting

- All can be fairly easily implemented as Bolts
Plan for today

- XFilter ✓
- Stream Processing ✓
- Google File System (also Hadoop DFS)
  - GFS/HDFS and MapReduce
  - Key assumptions
  - Architecture and operation
- Introduction to MapReduce
  - Programming model
  - Data flow
  - A few simple example tasks
Suppose We Have a Very Unreliable Stream Processing Substrate

- We are reading records from a spout, and emitting events

- Each event gets sent to a bolt

- The bolts run on many compute nodes, and those nodes crash very frequently

- Key idea: distributed checkpointing with replication! This is one key idea underlying the Google File System (GFS) and MapReduce.
Key Questions

- How did Google scale up communication to support distributed (sharded) computation?
- We’ll look at the Google Filesystem, which was designed in a slightly different, pre-cloud era:
  1. People focused on solving reliability/scale via hardware
  2. OSes had 32-bit filesystems – files couldn’t be bigger than 4GB!
  3. People generally assumed filesystems should be general-purpose

- The Google approach:
  - Software-driven reliability (now standard practice)
  - Filesystem driven by an application domain (very large streams)
What Is a File System?

- We have a lot of expectations of a file system:
  - Hierarchical names
  - Persistent storage of data (though there are limits)
  - Files can be created, updated, appended, deleted

- Handles everything from temp files to logs to videos to database extents, and more
An Idea:
Can We Use Files to Communicate?

- Suppose we have two processes – can we treat a file as a queue?

- An appender adds records to the end of a file

- A reader starts reading from the beginning
Data Streams in Files

- Instead of sending Storm style tuples on data streams (with partitioning), communicate through files (on a computer cluster)
  - Code is going to take a very “stylized” form; at each stage each machine will get input from files, send output to files
  - Files are generally persistent, name-able (in contrast to DHT messages, which are transient)
  - Files consist of blocks, which are the basic unit of partitioning (in contrast to object / data item IDs)

- But we’ll have to think differently from standard files. Let’s review!
Background: Distributed filesystems

- Many distributed filesystems have been developed:
  - **NFS, SMB** are the most prevalent today
    - You probably use them here in SEAS – e.g., every time you log into eniac or a Linux box, or a Windows machine
  - Andrew FileSystem (AFS) was also fairly popular

- Hundreds of other research filesystems, e.g., Coda, Sprite, ... with different properties
NFS in a Nutshell

- (Single) server, multi-client architecture
  - Server is **stateless**, so clients must send all context (including position to read from) in each request

- Mostly mimics UNIX semantics
  - Opening a file requires opening each dir along the way
    - `fd = open("/x/y/z.txt")` will do a
      - lookup for `x` from the root handle
      - lookup for `y` from `x`’s handle
      - lookup for `z` from `y`’s handle
  - Server must commit writes immediately

- Pros/cons?
The Google File System (GFS)

- **Goals:**
  - Support millions of huge (many-TB) files
  - Partition & replicate data across thousands of unreliable machines, in multiple racks (and even data centers)

- **Willing to make some compromises to get there:**
  - Modified APIs – doesn’t plug into POSIX APIs
    - In fact, relies on being built over Linux file system
  - Doesn’t provide transparent consistency to apps!
    - App must detect duplicate or bad records, support checkpoints
  - Performance is only good with a particular class of apps:
    - Stream-based reads
    - Atomic record appends
Key assumptions in GFS

- Component failures are the common case
  - Thousands of storage machines, built from commodity parts
  - Need monitoring, error detection, fault tolerance, recovery!

- Special application workload
  - Small number of very large, multi-GB files
  - Primarily large streaming reads + small random reads
  - Many large, sequential appends (e.g., many-way merging or producer/consumer); random writes are rare
  - Multiple clients may be writing/appending concurrently
  - Exact sequence of appended records does not matter

- Benefits from co-designing file system & apps
  - For example, can relax consistency w/o burdening application
Files broken into fixed 64MB "chunks"

Master stores metadata; 3 chunkservers store each chunk
  - As with Napster, actual data transfer from chunkservers to client

Clients cache metadata, but not files!

Pros/cons?
The Master: Metadata and Mappings

- Controls (and locks as appropriate):
  - Mapping from files -> chunk IDs within each namespace
  - Maintains a log (replicated to backups) of all mutations to the above
  - Controls reallocation, garbage collection of chunks

- Also knows mapping from chunk ID -> \{machines\}
  - Doesn’t have persistent knowledge of what’s on chunkservers
  - Instead, during startup, it polls them
  - ... Or when one joins, it registers
Primary/backup (in general)

- **Scenario**: Multiple replicas, concurrent requests from different clients
  - How to ensure consistency?

- **Idea**: Designate one replica as the **primary**
  - Accepts all requests, orders them, and then forwards ordered requests to the **backups**
  - When should the primary report success to a client?
  - What has to happen when the primary fails?
  - What kind of consistency does this provide?