IBM Watson

Accelerating Document Retrieval and Ranking for Cognitive Applications

Presenters:

Tim Kaldewey – Performance Architect

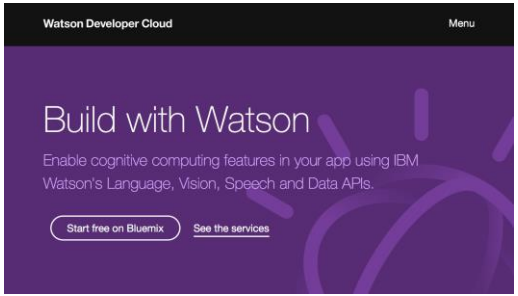
David Wendt – Performance Engineer

IBM

Disclaimer

The author's views expressed in this presentation do not necessarily reflect the views of IBM.

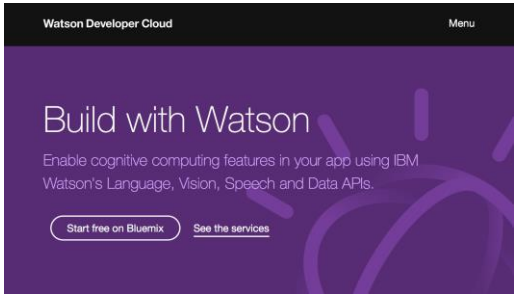
Watson evolution



*http://www-03.ibm.com/software/businesscasesudies/us/en/corp?synkey=Y362451T34615G34

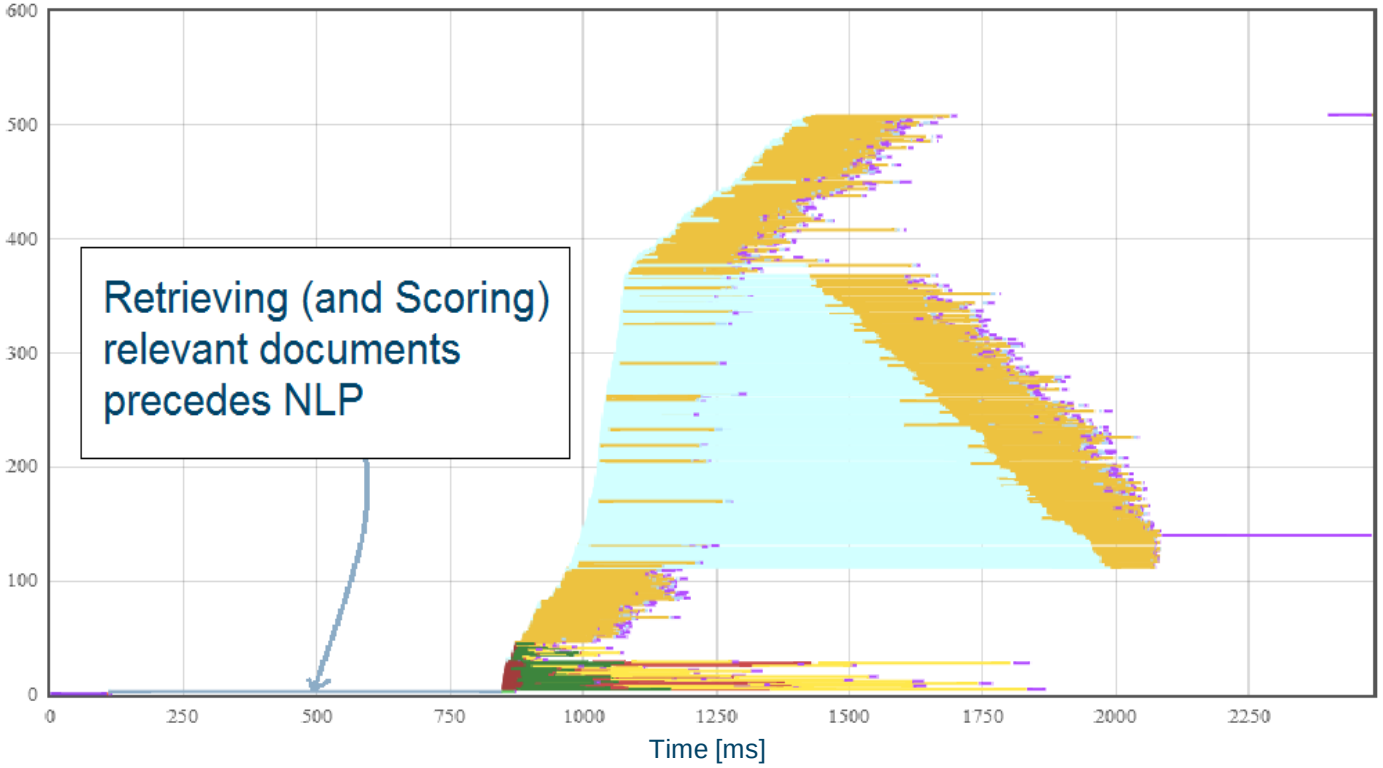
Watson evolution

40x*



*http://www-03.ibm.com/software/businesscasestudies/us/en/corp?synkey=Y362451T34615G34

A “brainwave” for answering a question



Background

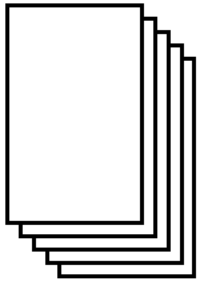
- Querying unstructured data (text) to identify relevant documents is a **prerequisite** for many cognitive data processing tasks (NLP)
- The large number of queries and the **volume** of unstructured data require a highly performant mechanism

  Example: - Lucene index of Wikipedia (5 million docs) is 105GB
  - Average search comprises 7 terms (keywords)
  - On average 115 thousand documents scored per search
- Scoring of candidate documents and passages is highly parallelizable.

➔ Acceleration can can be leveraged to **improve response time** and/or enable more complex queries to **improve accuracy**

Document Search

This provincial government of Canada is officially known as the government of Newfoundland and what region?



Index is organized in term-document format

- Retrieve the documents that are most likely to have the answer(s) to the question

- Search for documents that contain the words from the question
- Rank the documents based on
  - How frequent the words and word combinations appear in each document
  - The distance between these words in those documents

Anatomy of Lucene Query

Turn text into a Lucene query to retrieve relative documents.

This provincial government of Canada is officially known as the government of Newfoundland and what region?

```
+canada +newfoundland +provinci +govern +offici +known^0.5 +region
"provinci govern"~2 "govern canada"~2 "offici known"~2^0.9
"known govern"~2 "govern newfoundland"~2 "offici region"~3
```

- Words are stemmed and some stop words (the, of, as, …) are removed.
- **Key**words become term clauses: `canada newfoundland provinci govern offici …`
  - Scores are computed based on term frequency.
- Word pairs (phrases) become span clauses: `"provinci govern"~2 …`
  - Scores are computed based on frequency of phase and word distance between words

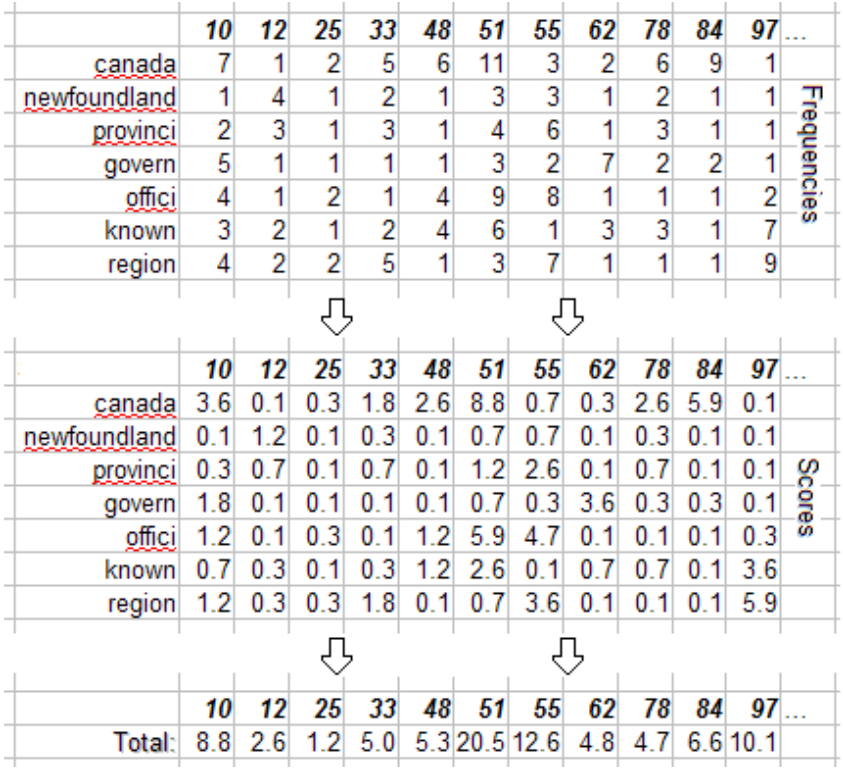
- Complex queries (e.g. nested span clauses) can improve accuracy by scoring higher more relevant documents.

Scoring term clauses

- Lucene is very efficient making only one-pass to match and score
- Index format is optimized for speed in matching terms to documents

- For each document, score each term clause and then sum the scores

- Scorer takes three values:
  - Term frequency
  - Document length
  - Term probability



	10	12	25	33	48	51	55	62	78	84	97	…
canada	7	1	2	5	6	11	3	2	6	9	1	
newfoundland	1	4	1	2	1	3	3	1	2	1	1	
provinci	2	3	1	3	1	4	6	1	3	1	1	
govern	5	1	1	1	1	3	2	7	2	2	1	
offici	4	1	2	1	4	9	8	1	1	1	2	
known	3	2	1	2	4	6	1	3	3	1	7	
region	4	2	2	5	1	3	7	1	1	1	9	

Frequencies

	10	12	25	33	48	51	55	62	78	84	97	…
canada	3.6	0.1	0.3	1.8	2.6	8.8	0.7	0.3	2.6	5.9	0.1	
newfoundland	0.1	1.2	0.1	0.3	0.1	0.7	0.7	0.1	0.3	0.1	0.1	
provinci	0.3	0.7	0.1	0.7	0.1	1.2	2.6	0.1	0.7	0.1	0.1	
govern	1.8	0.1	0.1	0.1	0.1	0.7	0.3	3.6	0.3	0.3	0.1	
offici	1.2	0.1	0.3	0.1	1.2	5.9	4.7	0.1	0.1	0.1	0.3	
known	0.7	0.3	0.1	0.3	1.2	2.6	0.1	0.7	0.7	0.1	3.6	
region	1.2	0.3	0.3	1.8	0.1	0.7	3.6	0.1	0.1	0.1	5.9	

Scores

	10	12	25	33	48	51	55	62	78	84	97	…
Total:	8.8	2.6	1.2	5.0	5.3	20.5	12.6	4.8	4.7	6.6	10.1	

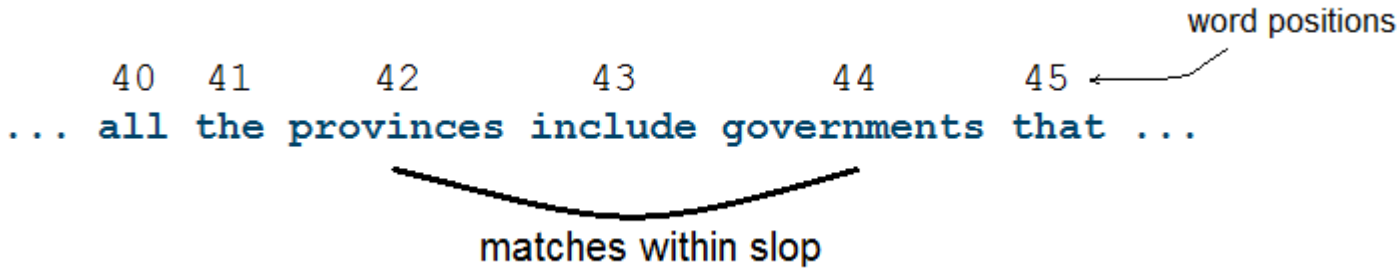
Scoring span clauses

"provinci govern"~2 "govern canada"~2 "offici known"~2
"known govern"~2 "govern newfoundland"~2 "offici region"~3

Scoring here uses a ‘sloppy’ frequency value calculated based on how often the term pair appears and how close together the terms are to each other.

Clause form: **span(term1,term2,slop,order)**

Example: `span(provinci,govern,2,false)`



Scoring span clauses – continued

```
span(provinci,govern,2,false)
```

- Position vectors vary in length per term per document.

	10	12	25	33	48	51	55	62	78	84	97	…	doc-ids
provinci	2	3	1	3	1	4	6	1	3	1	1		frequencies
govern	5	1	1	1	1	3	2	7	2	2	1		

	10		12			25	33			48	51				55						62	78			84	97
provinci (positions)	22	41	2	11	28	51	3	21	33	34	1	12	17	25	5	10	38	59	78	81	5	12	44	98	42	23

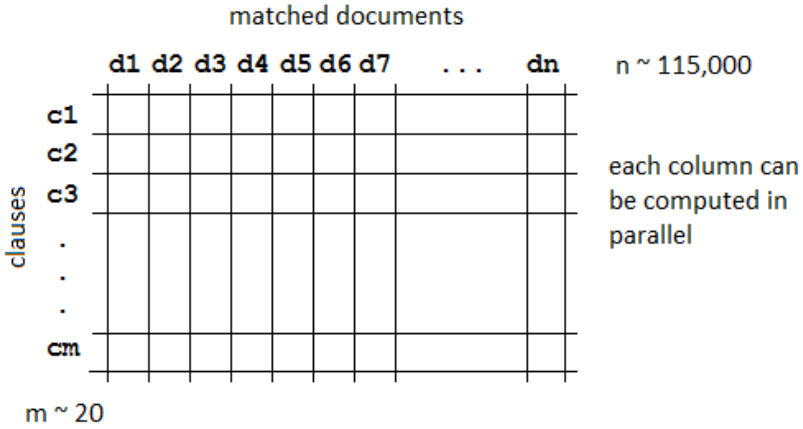
	10					12	25	33	48	51			55		62							78		84		97
govern (positions)	17	23	43	55	76	17	50	4	53	11	18	26	6	80	13	23	32	41	58	67	85	10	50	44	75	54

	10	12	25	33	48	51	55	62	78	84	97	…
sloppy freq	0.2	0.0	0.1	0.5	0.0	0.4	0.2	0.0	0.3	0.1	0.0	

	10	12	25	33	48	51	55	62	78	84	97	…	
Total:	1.2	0.0	2.8	5.1	0.0	4.3	2.1	0.0	3.4	2.7	0.0		scores

Analysis

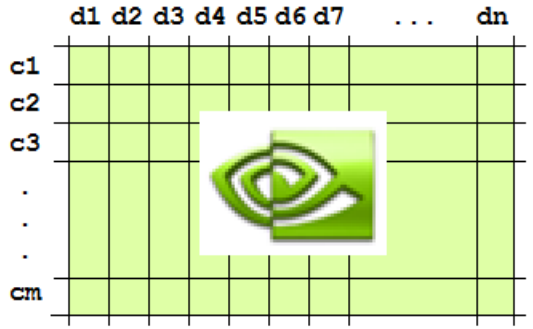
- Scoring for each document is independent from other documents



- At the end, scores are sorted to provide the document rank order

Perfect for GPU

- Floating point operations for thousands of items (documents) that can occur in parallel
- Each query clause is implemented as a set of kernels and the scores accumulate in a float array where each element is the score for a unique document
- The top N ranked document ids are returned to the host application



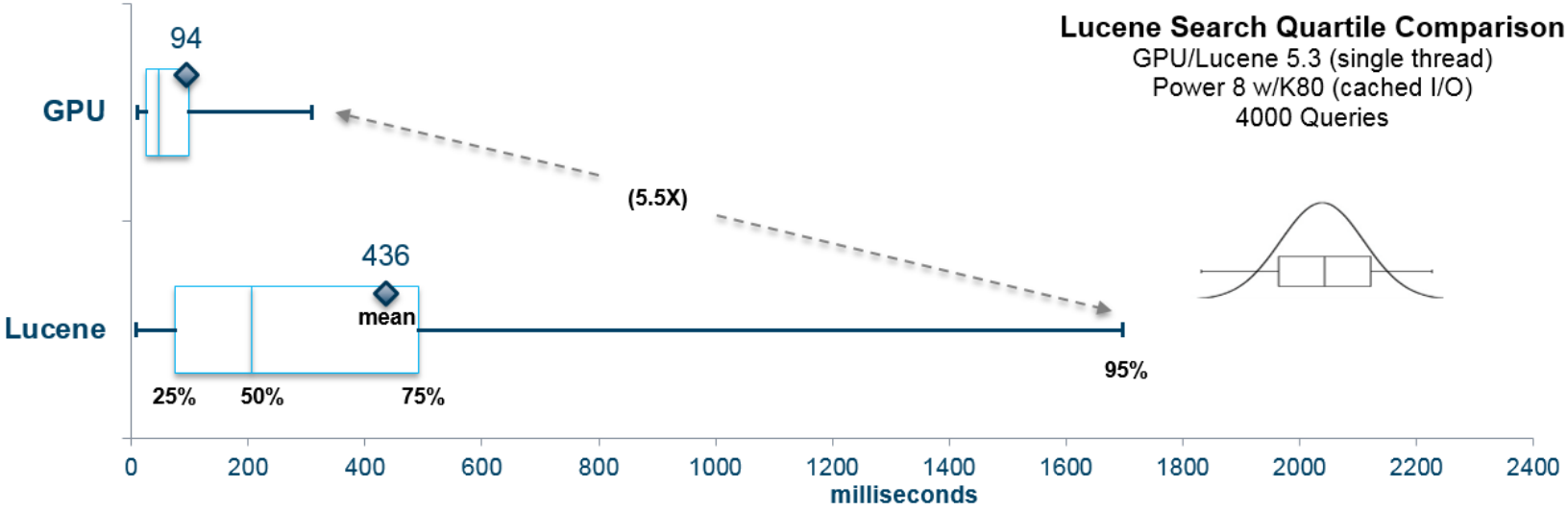
Scoring on the GPU

- We used the **thrust** libraries for sorting and intersecting to more easily include a CPU-only alternative
- All term clauses are scored first and can be calculated in a single kernel (loop)
- Spans are computed to maximize caching of term position values
- Once scored, the results are sorted and the top N document ids are returned along with their scores

```
// score all term clauses
__global__ void KernelScoreTerms(...)
{
   const int NV = GPU_TPB;
   int tid = threadIdx.x;
   int block = blockIdx.x;
   int gid = NV * block;
   int index = gid + tid;
   if( index < numIds )
   {
      float docScore = scores[index];
      float docLen = docLens[index];
      for( int j=0; j < clauseCount; ++j )
      {
         int iidx = clauses[j].tidx;
         float boost = clauses[j].boost;
         float prob = termInfos[iidx].prob;
         int* freqs = freqVals + termInfos[iidx].freqOffset;
         float freq = (float)freqs[index];
         float score = devLMDirichletScore(mu,docLen,prob,freq,boost);
         docScore += (score > 0.0) ? score : 0.0;
      }
      scores[index] = docScore;
   }
}
```

Only 5 custom kernels were required.

Results



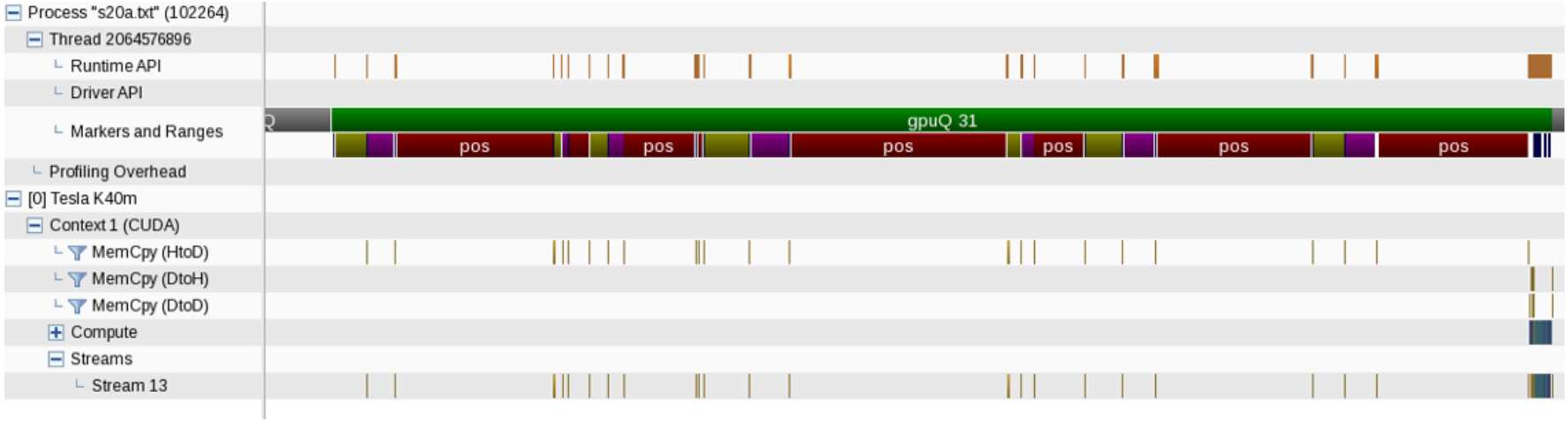
	Total (s)	Min	Max	Mean	StdDev	5%	25%	50%	75%	95%
Lucene	1,741	<1	8422	436	670	8	75	205	492	1697
GPU	456	<1	2788	94	162	10	26	47	99	308
SpeedUp	3.8		3.0	**4.6**		0.8	2.9	4.4	5.0	**5.5**

Making it Real

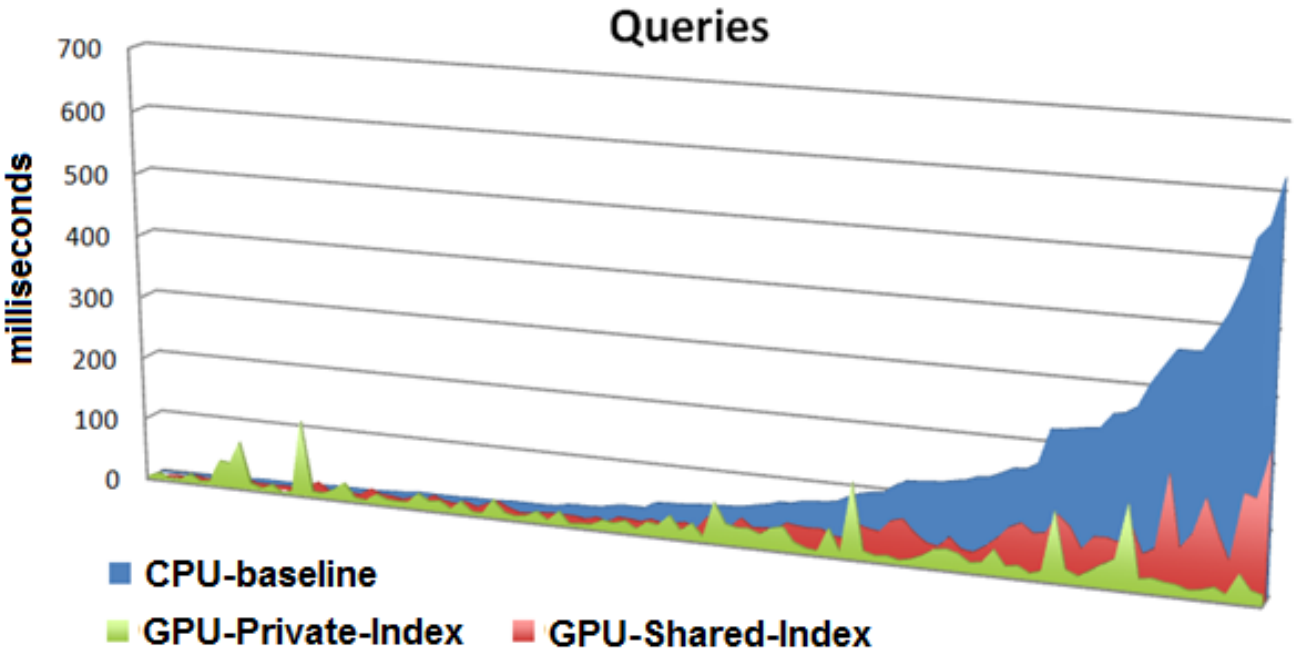
- Accessing the index data: ids, frequencies, positions
- Managing GPU access
- Recursion for nested clauses
- Scoring special cases
- Coverage of query types

Shared index data

- First approach was to create a custom index with only the values we needed for scoring.
- Sharing the index with the rest of Lucene would be ideal but how much would this cost us?

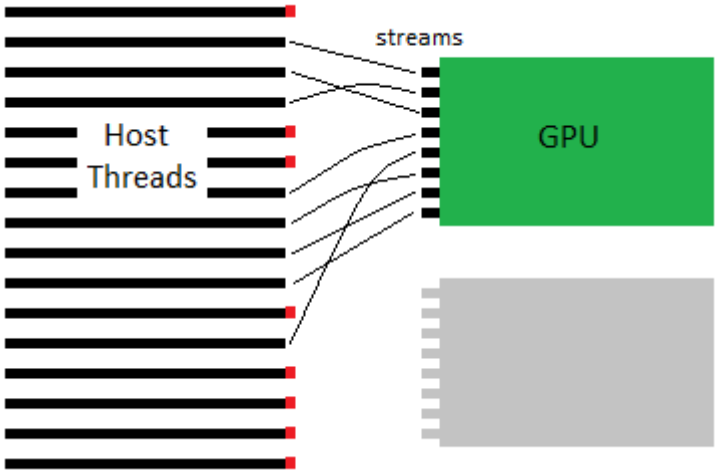


Shared index data - results



Managing GPU access

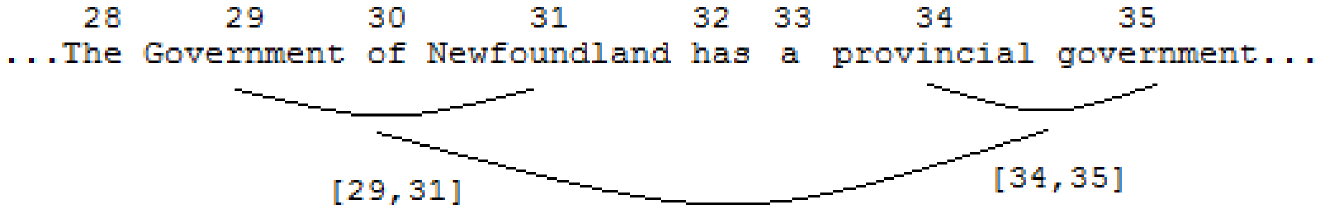
- Need to handle simultaneous queries from many host threads
- A dedicated set of streams – one per host thread – to handle each query
- Limited the number of streams based on the available GPU memory and index size
- Once the GPU is fully utilized, additional host threads can be blocked or can fallback to calling Lucene directly



Recursion for nested spans

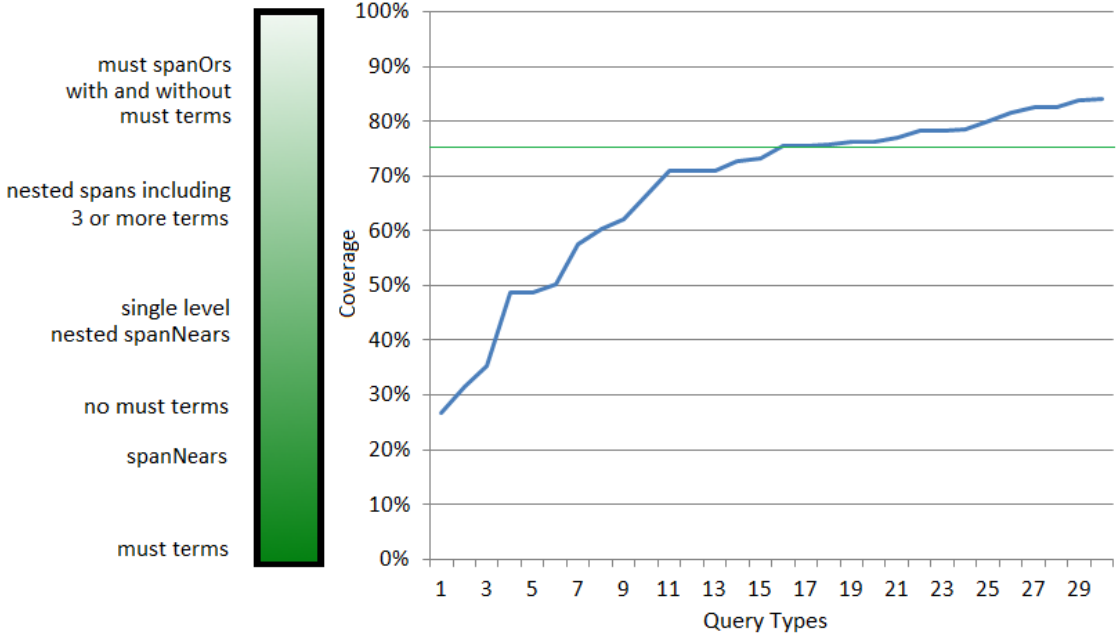
- Although CUDA supports recursion, having an unknown stack-size becomes an issue.
- Implemented the recursions as loops and managed a fake stack in global memory

```
span( span(govern,newfoundland,2,false), span(provinci,govern,2,false), 4, true )
```




```
compute sloppy frequency for all 4 terms with distance: [31,34]
```

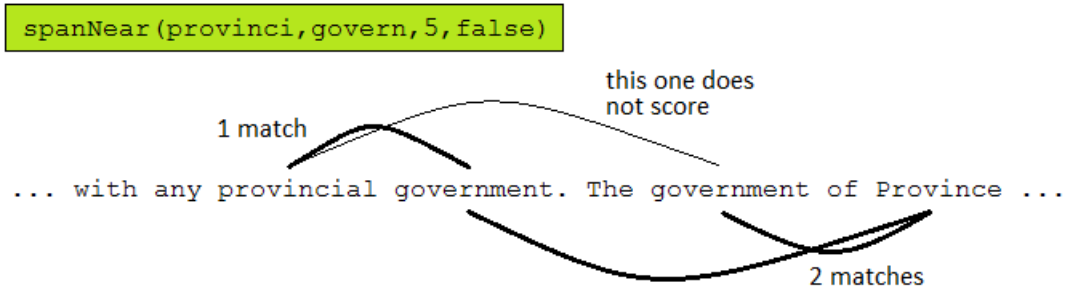
Query Types vs Coverage



- Query types are unique combinations of search clauses: terms, spanNear, spanOr, nested spans, etc.
- Coverage progression is from most common clause type to least common.

Scoring span clauses has special cases

- There are some special cases like when phrases overlap.



Conclusion

- Speed up by half an order of magnitude
- Many challenges: shared index, query types, recursion, …

- GPU performance is even higher for complex queries
  - Words resulting in many documents requiring more threads
  - Complex span clauses with many position values

- Speeding up query allows building more complex queries and scoring documents better which may help improve accuracy

Questions?