Making Watson fast

E. A. Epstein
M. I. Schor
B. S. Iyer
A. Lally
E. W. Brown
J. Cwiklik

IBM Watson™ is a system created to demonstrate DeepQA technology by competing against human champions in a question-answering game designed for people. The DeepQA architecture was designed to be massively parallel, with an expectation that low latency response times could be achieved by doing parallel computation on many computers. This paper describes how a large set of deep natural-language processing programs were integrated into a single application, scaled out across thousands of central processing unit cores, and optimized to run fast enough to compete in live Jeopardy!™ games.

Introduction

To successfully compete in the question-answering (QA) game Jeopardy!**, IBM Watson* had to be as fast as the best human champions at answering Jeopardy! clues. For IBM Watson to determine its answer and an accurate confidence in time to push the buzzer first, it had to complete a complex computation in the few seconds it takes the host to read each clue. The research team anticipated that parallel processing would be critical to scaling Watson out to reduce latency and, for this reason among others, designed an "embarrassingly parallel" logical architecture with fine-grained units of analysis. For example, a deep evidence-scoring component [1] in DeepQA analyzes a single text passage at a time and provides a score indicating the strength of the evidence for each candidate answer in the passage. This approach sharply contrasts with earlier approaches (e.g., [2, 3]), in which the unit of analysis is an entire set of passages, and the task for components of this sort is reranking. By breaking the subtasks of QA down into very small units such as a single passage, the DeepQA architecture facilitates a particularly high level of parallelization.

Unstructured Information Management Architecture (UIMA) was chosen as the integrating framework for the large and diverse set of analytic components that comprise the Watson QA system, with the expectation that UIMA-AS (Asynchronous Scaleout) would enable scalability across multiple machines. Beyond those architectural commitments, development was initially focused exclusively on accuracy and confidence, and processing speed was largely ignored barring egregious performance. Only after making significant progress in QA accuracy and confidence estimation over a two-year period did we begin work to make Watson fast.

The initial Watson application typically took 1 to 2 hours to answer a question running with a single processor core. This paper describes how we scaled Watson to use thousands of processor cores, working in parallel, to respond in less than 3 seconds on average. The next section describes the UIMA implementation of the DeepQA architecture, which is essential for understanding the scale-out design. Following that are details on the scale-out design, including feasibility testing and finding bottlenecks. The remainder of this paper deals with effective performance optimizations applied to the analytics themselves.

UIMA implementation of the DeepQA architecture

The DeepQA architecture [4] defines, at a logical level, how the QA task is broken down into separate stages of processing that can be executed in parallel. In this section, we show how this architecture was implemented using Apache UIMA** [5], which is an open-source framework for assembling multimodal and natural-language processing (NLP) applications. UIMA also has mechanisms to facilitate parallel processing.

Implementing an analytic application on UIMA requires encapsulating analysis code into UIMA annotators, formalizing all analysis input and output data with respect to a Type System, organizing the data into work units called Common Analysis Structures (CASes) that are passed between annotators, and specifying the flow that each CAS takes through all possible annotators.

Digital Object Identifier: 10.1147/JRD.2012.2188761

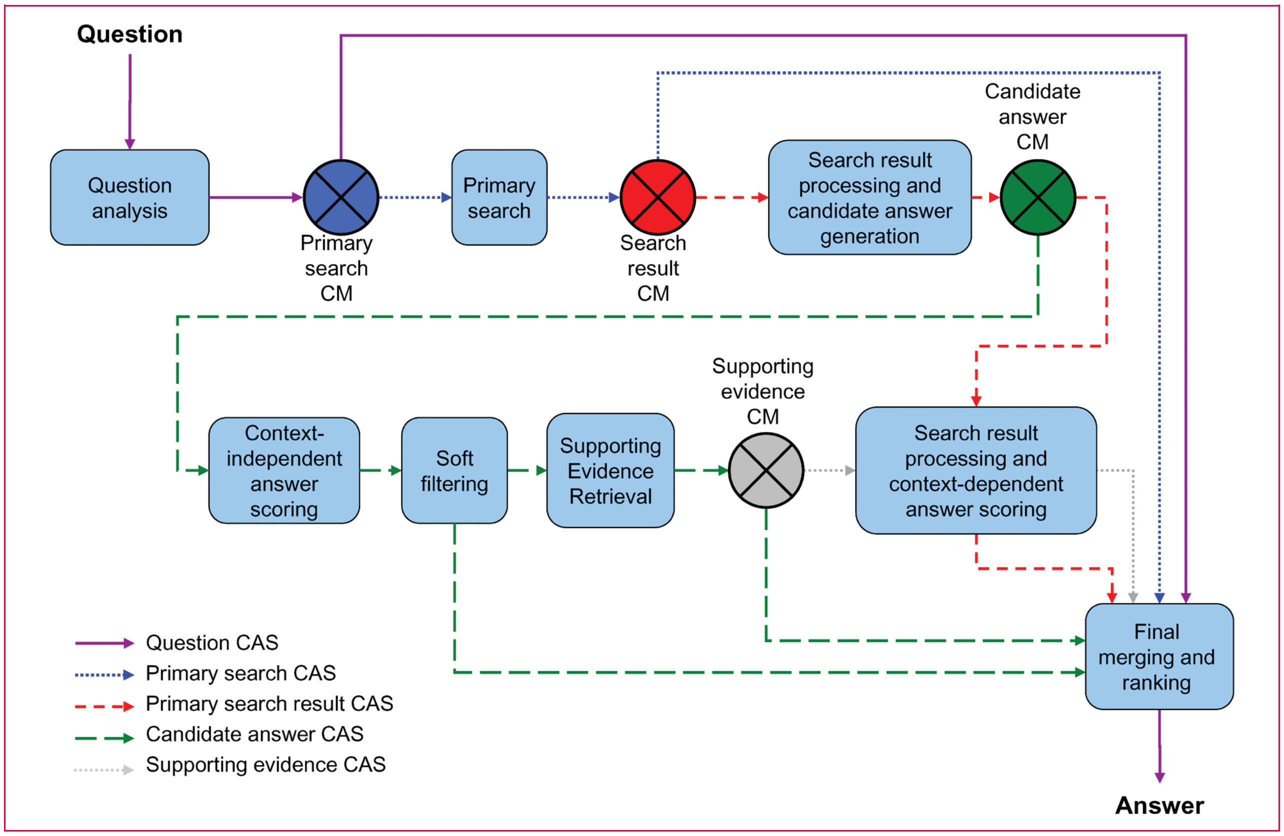
Question
Question analysis
Primary search CM
Primary search
Search result CM
Search result processing and candidate answer generation
Candidate answer CM
Context-independent answer scoring
Soft filtering
Supporting Evidence Retrieval
Supporting evidence CM
Search result processing and context-dependent answer scoring
Final merging and ranking
Question CAS
Primary search CAS
Primary search result CAS
Candidate answer CAS
Supporting evidence CAS
Answer

Figure 1

DeepQA/UIMA architecture.

Here is a more formal description of these UIMA concepts.

Type System—A declarative data model that defines the Types (classes) and their Features (properties) used by a UIMA application. An instantiated Type is called a Feature Structure.
Common Analysis Structure (CAS)—A unit of work in UIMA that is passed from one component to another. A CAS encapsulates Feature Structures that are organized into Views. Each View has a unique Subject of Analysis (e.g., a text string to be analyzed), the analysis results (metadata describing the Subject of Analysis), and indexes to the analysis results.
Annotator—An individual processing component in UIMA. Receives an input CAS, does analysis of its contents, and typically modifies it (e.g., adds, deletes, or modifies Feature Structures).
Flow Controller—Determines how CASes move among a collection of UIMA components.

One approach to data organization for Watson would be to have a Jeopardy! question (category + clue) as the Subject of Analysis with analysis results accumulated in a single CAS View as it moved through all processing steps. This approach would not be conducive to computational parallelism, where input data and results must be exchanged across many machines, nor would it be helpful to organizing the many megabytes of results for hundreds of candidate answers. UIMA offers another concept that can be used to solve both of these problems:

CAS Multiplier—A type of UIMA annotator that may create and output new CASes that are considered children of the input CAS.

Watson uses several levels of CAS Multipliers to create thousands of child CASes during the processing of a question. The child CASes are designed to contain just the data needed for each processing step. Organizing the data into multiple CASes allows them to be processed concurrently. Keeping the data in each CAS small minimizes the overhead of transferring data for processing on other machines.

The following sections detail the UIMA implementation of Watson. Throughout the following discussion, we refer to **Figure 1**.

Component types

Each DeepQA/UIMA component is implemented as a UIMA annotator and is assigned to one of several logical component types itemized as follows:

Question analysis—Attempts to understand what the question is asking and performs the initial analyses that determine how the question will be processed by the rest of the system.
Primary search—Retrieves answer-bearing content.
Search result processing—Performs NLP analysis on a text search result (either a primary search result or supporting evidence search result).
Candidate answer generation—Extracts candidate answers from that content.
Context-independent answer scoring—Scores candidate answers along many dimensions, such as geospatial similarity.
Soft filtering—Runs after candidate answer scoring to select the most promising answers, roughly the top 20%, for further analysis.
Supporting evidence search—Gathers and evaluates additional supporting evidence for the most promising candidate answers.
Context-dependent answer scoring—Scores the additional evidence retrieved for selected candidate answers.
Final merging and ranking—Combines all of the evidence for each candidate answer, merges answers as needed, assigns ranks, and computes confidence scores.

Type System

In any UIMA application, the basis for communication between the different components is the shared Type System. There are two kinds of types in the DeepQA/UIMA type system: general-purpose NLP types, which are used to represent a linguistic analysis of the question or of a text passage, and DeepQA-specific types, which define critical elements of the QA architecture. The DeepQA-specific part of the Type System includes the following important types:

Search—A query that is executed in order to retrieve content relevant to the question. This includes queries over text corpora, as well as semistructured or structured sources. Each query can result in multiple *SearchResults*.
SearchResult—A single result produced by executing a *Search*. This could be a document, a text passage, or an entity from a structured knowledge-base. A *SearchResult* can contain multiple *CandidateAnswers*.
CandidateAnswer—A potential answer identified from a *SearchResult*.
CandidateAnswerFeature—A (label, score) pair associated with a *CandidateAnswer*. Each *CandidateAnswer* has many *CandidateAnswerFeatures*, which are used by the Final Merging and Ranking component to select the best answer.

Flow controller

A UIMA application may provide a flow controller that dictates how CASes flow through the various components in the system. Many UIMA applications use a default linear flow, but in DeepQA, this does not suffice. Instead, the DeepQA/UIMA Flow Controller is aware of different "flavors" of CAS, each of which undergoes a different kind of processing. Each CAS flavor is essentially a linear flow, represented by a different colored line in Figure 1. These CAS flavors and their flows are as follows:

Question CAS—Initially contains the raw question. Processed by question analysis.
PrimarySearch CAS—Initially contains question analysis results. Processed by primary search, which adds a Search object and SearchResults to the CAS.
PrimarySearchResult CAS—Initially contains question analysis results and one primary SearchResult. Processed by search result processing, then candidate answer generation, then context-dependent answer scoring.
CandidateAnswer CAS—Initially contains question analysis results and one CandidateAnswer. Processed by context-independent answer scoring and soft filtering, then (if filter is passed) supporting evidence retrieval.
SupportingEvidence CAS—Initially contains question analysis results, one CandidateAnswer, and one SearchResult. Processed by search result processing and context-dependent answer scoring.

All CASes are ultimately delivered to the Final Merging and Ranking component, which collects information from them as they arrive and produces the final answers with confidence scores.

CAS Multipliers

CAS Multipliers transition between CAS flavors. For example, the "Candidate answer CM" in Figure 1 takes Primary Search Result CASes as input and produces Candidate Answer CASes. To do this, it examines the annotations made by candidate generation (CG) components. For each CandidateAnswer that was annotated, a new Candidate Answer CAS will be produced.

In the Watson application, every CAS is given a unique name consisting of the question identifier (ID), the type of CAS, the instance number for this child, and its entire parent history. For example: `508295_search002_` `hit013_candidate020` represents the 20th candidate answer for question 508295, which was found in the 13th search hit of the second primary search.

This organization of analysis results clearly identified the computational history for every analysis step and often allowed problem solving by inspection. For deeper understanding, the analysis at any point in the processing of a question could be recreated by giving the desired input CAS to a specified analytic component. This organization of data into separate CASes is a key enabler for the massive parallelism required by DeepQA.

Scale-out architectures

Two different scale-out systems were used during Watson's development. The development system was optimized for throughput, scaling out computation during DeepQA algorithm development in order to run many experiments, each consisting of hundreds or thousands of questions. The production system was optimized for low latency in order to answer a single question in the shortest possible time.

Development system

The Watson application is composed of more than 100 different analytic components, about 1 million lines of Java** and C++ code. Algorithm development was focused on optimizing accuracy, and experiments required the results from thousands of test questions in order to be statistically significant. Researchers focused entirely on improving accuracy and confidence measures.

Algorithmic flexibility and the ability to run many experiments were of key importance during algorithmic development. Algorithmic flexibility meant easily evaluating the addition of new corpora or the addition of new analytics at any processing stage. To support experimentation, overall question processing throughput was more important than the latency for individual questions. To these ends, the analysis was broken up and run in pieces, saving intermediate results that could be reused effectively.

The core UIMA framework allows components to be easily combined as "delegates" into an aggregate analysis engine. Aggregates representing the entire application or any subset of components could be assembled quickly. Individual components and small aggregates were debugged on user workstations. Larger aggregates or components with large resource requirements were debugged on larger "worker" machines with 32 GB of RAM. After the aggregate was debugged, we scaled out for throughput by deploying multiple instances of the aggregate on each of many worker machines, each instance working on a different question at the same time.

Production system

The interactive game-playing configuration was dubbed the production system. Here, algorithmic flexibility was secondary to QA latency. For example, reorganizing the passage search subsystem for fast query times trumped the ability to easily change corpora. Scale-out granularity changed from replicating the full analysis pipeline as in the development system to replicating individual components such as Candidate Generation (CG) in order to derive all candidate answers for 7one question in parallel.

However, even as work began to create a scaled out version of Watson, it was clear that algorithmic development would have to continue right up to the final game in order to meet the necessary QA accuracy. Therefore, it was critical that the time and effort to migrate new development code into the low latency production environment be as small as possible.

Feasibility testing of UIMA-AS for Watson

UIMA-AS [6] is a general solution for scaling out UIMA components. The key concept of UIMA-AS is to scale out a UIMA aggregate analysis engine across a set of CPUs, on one or more machines, without requiring any changes to existing component code or descriptors. UIMA-AS adds to UIMA a new "deployment descriptor" that specifies how each delegate is to be deployed. For example, an annotator could be replicated and run in separate threads in the same process as other annotators, or it could be replicated as separate processes running on different machines. The UIMA-AS framework manages the multithreading and interprocess communication.

Consistent with the core UIMA paradigm, each processing step has a client sending a CAS to a UIMA component and receiving back the modified CAS. In an aggregate analysis engine, the client is the aggregate controller (part of the UIMA framework), and the delegates are the UIMA components. For delegates colocated in the same process as the aggregate controller, there is no overhead for delivering a CAS; when delegates are deployed as separate processes, each CAS is serialized to and from the delegate. UIMA-AS uses the JMS (Java Message Service) standard for interprocess communication.

UIMA-AS had the right properties as far as sharing components between development and production systems, but we needed to know if the framework would be able to scale out across hundreds and possibly thousands of CPU cores with the efficiency needed to finish the computation in 3 seconds or less. To test this, we simulated a production system with dummy analytics but real data flow. In essence, the only code running was the UIMA framework, putting data into CASes and transporting CASes between processes deployed across a cluster of machines.

The dummy analytics created the expected number of child CASes, and the "computation" in each component simply added to the input CAS the expected number of feature structures with actual analytic results. The number of instances of each component was determined by the

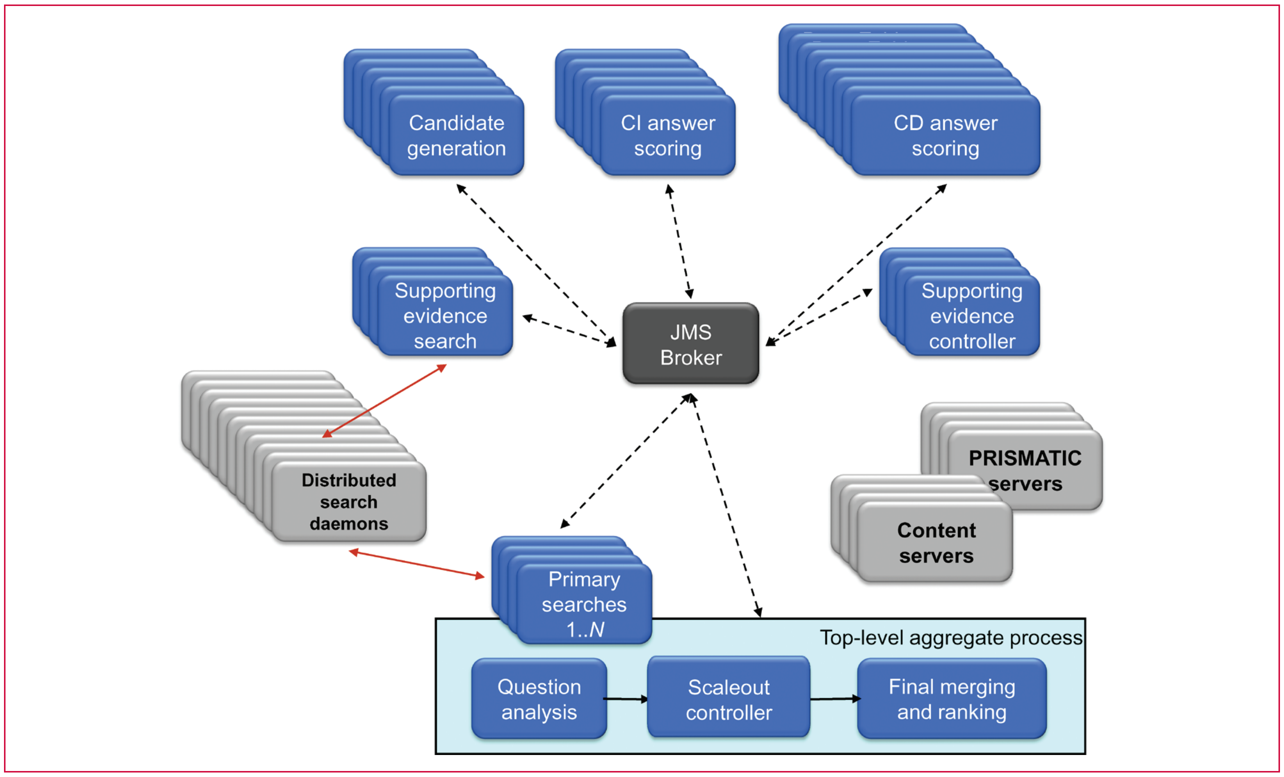

Figure 2

Major components in the Jeopardy! system.

typical number of pieces of work that the component would receive for a question. For example, a total of 80 primary search hits are typically returned from all primary search components; hence, 80 instances of CG were deployed, i.e., eight per process on each of ten different eight-core machines. This allows all 80 search hits to be processed in parallel, assuming that the UIMA-AS framework delivers the data fast enough.

The feasibility test deployed a total of 110 multithreaded processes on 110 eight-core machines. When this test took longer than 3 seconds, it was quickly apparent that UIMA-AS needed to be improved. Detailed measurements showed that virtually all of the time was only in two places: CAS serialization and network communication.

CAS serialization time was first reduced by having services return only the parts of the CAS that were changed. UIMA calls this delta CAS serialization. Further reductions came from using UIMA's more efficient, but less flexible, binary CAS serialization method. Serialization time was also reduced by serializing CASes in multiple threads. The default UIMA-AS framework uses a single thread to deserialize CASes returned from a remote component. By optionally specifying replicated reply-queue listeners, this processing bottleneck is avoided.

A final serialization improvement from the feasibility work was at the application level. The results of question analysis, kept in a CAS "question" view, are needed by all components and, hence, were being copied into every child CAS and serialized out on every remote processing step. To avoid this, the Question View is now broadcast to all remote UIMA-AS processes in single operation, using the UIMA serialization mechanism along with the JMS ability to broadcast a single message to many consumers. This substantially reduced the content of all CASes sent to remote processes.

After these changes, the feasibility test ran in less than 1 second, with no measurable CPU time in any component. That is, UIMA-AS introduced negligible CPU overhead, and the remaining time was due to network communications, which turned out to be insignificant in the presence of the actual analytic computations.

Production system deployment

Figure 2 shows the final production system used for the Watson Jeopardy! game-playing system. In total, there are 400 processes deployed across 72 IBM POWER* 750 machines. About half were large-memory Java processes, and the other half were small-memory Indri [7] Distributed Search Daemons. All UIMA-AS services

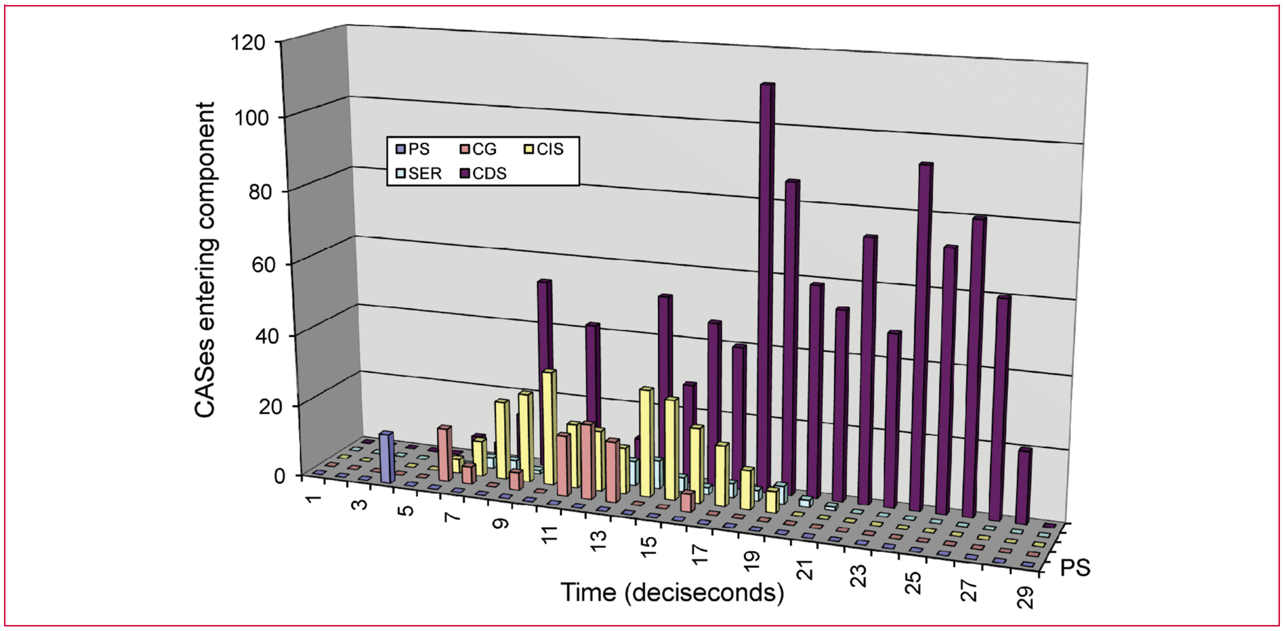

Figure 3

CAS arrival frequency at different components. Arrival time delays from one component to the next are indicative of processing duration. For example, primary searches all start at the same time but end from 0.2 to 1.3 seconds later.

communicated with each other via JMS. Primary and Supporting Evidence searches have Indri clients directly connecting to the search daemons. Most of the UIMA-AS components accessed content servers, PRISMATIC servers, or both (these servers described below).

Figure 3 tracks CAS flow through the cluster for a typical question. All primary search components started immediately following question analysis. As each primary search completed, primary search hit CASes arrived at CG; in this example, the longest primary search took 1.2 seconds. After CG, the candidate answer CASes began arriving at context-independent scoring and on through the rest of the pipeline.

Finding bottlenecks in Watson

In the course of processing a single question in the fully scaled out version of Watson, an average of 1,600 CASes are serialized out and back among UIMA-AS processes. Most of the UIMA-AS processes contain dozens of individual UIMA annotators, replicated in multiple threads. The entire system runs asynchronously. An early problem was finding an effective approach to identifying timing bottlenecks.

The first approach was to develop a generic tool that periodically (default interval is 1 second) captures a number of performance statistics for each delegate in a UIMA-AS process. Instantaneous statistics include input and reply queue depths for delegates and the number of free CASes available to each CAS multiplier (a fixed number of CASes is the main throttling mechanism used by UIMA-AS). Interval statistics include the number of CASes delivered to and the CPU time used by each delegate. This monitor, now part of the UIMA-AS run time package, was useful early in the scale-out effort, finding the serialization-related bottlenecks in the simulator.

The second and more essential approach was to integrate timing measurements into the Watson application code. A small amount of code added to the custom flow controller component captured statistics for the main components described in the "Component types" section. For each component, the number of CASes, along with their average and maximum durations, was logged.

With just these few numbers, any timing outliers could be isolated quickly. Questions with poor timing were rerun, and the CASes that were sent to the problem component were captured. With this CAS data as input, the component was run in a single-threaded UIMA application in order to see more detailed timing. Commonly, it was only one or a small number of CASes that exposed the problem.

Table 1 shows times for two answers: fast path and full path. The fast-path answer, which is derived without supporting evidence, is computed in order to give Watson a chance to buzz in on the shortest clues.

Table 1 Performance over 122 questions. (CDS: Context-Dependent Scoring; CIS: Context-Independent Scoring; FM: Final Merger; SER: Supporting Evidence Retrieval.)

	Time (ms)	
	Average	*Maximum*
Fast path	2,183	5,236
Full path	2,527	5,511
Question answering	374	670
Passage search max	846	3,247
CG num	97	165
CG average	96	239
CG max	398	1,164
CIS num	280	495
CIS average	190	673
CIS max	550	1,784
CDS num	1,027	3,300
CDS average	34	84
CDS max	328	703
SER num	49	165
SER average	266	909
SER max	548	2,120
FM max	350	925

Figure 4 shows QA latency over ~2,000 test questions. The top histogram was for the first live end-to-end system, after six months of work. The bottom system used corpus preprocessing (see below), added a number of new analytics, doubled the input corpus size, and used more than twice the hardware resources.

Getting everything into RAM

The speed requirements for Watson compelled the avoidance of disk I/O delays while running; hence, our design point is to put all the data we need to answer a question into RAM.

Our initial production system hardware resources primarily consisted of machines with 32 GB of memory. This may sound like a large amount of memory, but the Watson application had many resources consisting of huge tables of information that individually exceeded these sizes. The most important techniques we used to reduce memory requirements and deal with side effects of keeping large amounts of data in the Java heap are described below.

Reduce size of references—Java comes in two types: one running in a 32-bit address space, and using 4 bytes for references (pointers), and one running in a 64-bit address space, and using larger reference pointers. A 32-bit address space is normally limited to 4 GB (4 GB = 2 to the 32nd power). Many 64-bit Java implementations support a special option that allows the Java to address up to 32 GB (e.g., eight times larger than the 32-bit Java) with references that take only 4 bytes; they do this typically by shifting the references left by 3 to get an offset into the heap. Running with this option can greatly reduce the footprint of applications that have lots of interobject references.

Shrinking Java Objects—We used a measurement-driven approach, looking for where we could get the most leverage, and found many opportunities. In doing this work, we considered the cost to get data from memory through the various cache levels into the CPU core relative to the time it took to perform operations on the data once it was fetched. This led to designs where we would encode things within an "int" and use bit operations (masking and shifting) to extract the parts, more or less for "free" in terms of performance.

Eliminating Java per-object overhead—One item that turned up early was the overhead per Java object. We had many data structures where the "natural" implementation was a Java object, and the number of these objects was in the multiple-millions. Depending on the object and the Java implementation, there is an overhead per object, typically 16 or more bytes per object.

To avoid this overhead, we store multiples of these objects inside one containing array. If the objects are arrays of things, we store these inside one big array, together with their lengths. The references we call "handles", are then direct indexes into this one big array.

For arrays of arrays of "int"s, instead of the obvious implementation `int[][]`, we store these in one `int[]` array, each subarray along with its length. When more efficient, we store the lengths in other arrays (e.g., byte arrays). This could happen, if, for instance, you know the length of the subarrays will always be under 256; then the lengths can be stored in a separate byte array.

Storing strings—Watson uses many strings. Storing these strings in the standard Java "char" format takes 2 bytes per string. Since the vast majority of our data was in plain ASCII, we took the approach of storing the data in UTF-8 encoded form, which averaged close to 1 byte per character.

Watson usually "interns" strings to share string storage space for identical strings and returns an "int" handle representing the string; it does this by storing the UTF-8 version of the string in a special hash table, optimized for space saving (see next section).

Special hash tables—Watson makes extensive use of lookup tables and uses HashMaps for this. Watson's use of these follows a pattern of loading the table at start time and then looking things up in the table while running. We exploit this characteristic (that the table is "read-only" after loading) to do several space (and speed) optimizations. The result is tables that take only 10% to 15% of the space needed for normal HashMaps.

The read-only aspect of these tables is used when implementing storage for the bucket chains for collisions. For

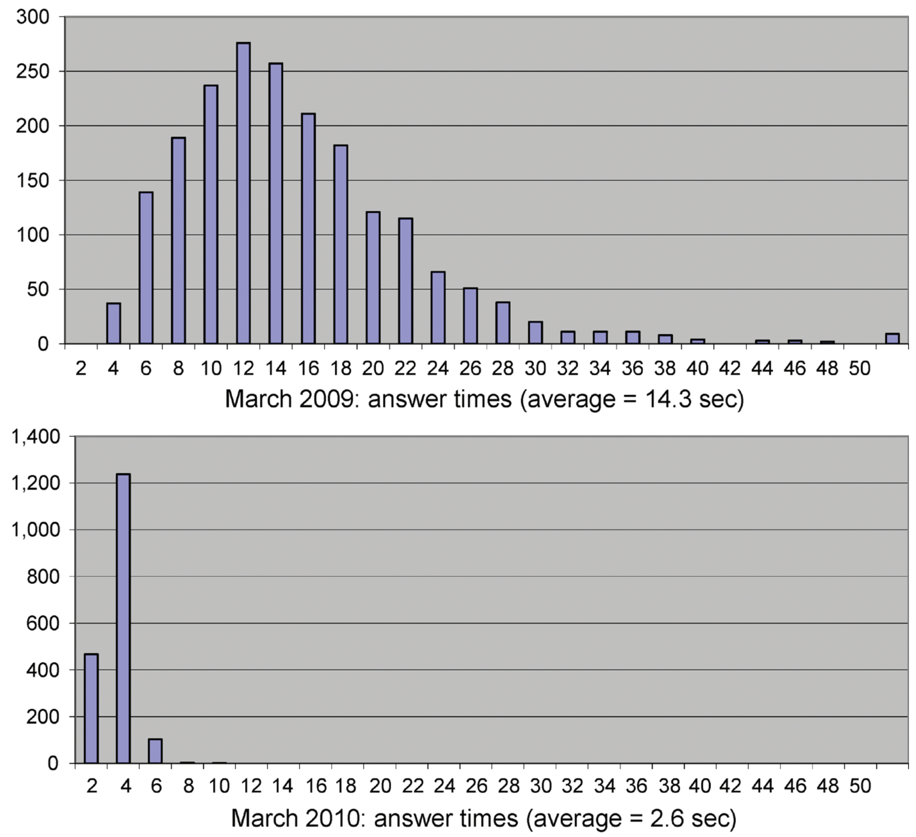

Figure 4

Latency improvement from first end-to-end application.

each hash location in the hash table, there is a bucket-chain, that is, a list of items, all of which share the same hash code; each entry in the chain is typically a key and its associated value. Rather than use the conventional implementation of a linked-list of entry items, we use an (usually primitive) array implementation. This is feasible because the array is fixed (because the table is read-only). Furthermore, the per-object overhead of these arrays is eliminated using the above approach of storing multiple arrays in one Java object. For HashMaps used for interning, since the key and value are the same, we eliminate storing both of these.

For HashMaps where the value is a set of primitives, and the key and the individual values can be coerced into similar kind of primitives, we build special versions of space-saving Maps where the key and value(s) are stored together as items in the bucket chain. Each collapsing of this kind saves two references and two object overheads and potentially speeds up access because of the locality of the subitems to each other.

Storing the characters in 1 byte versus 2 accounts for only a small portion of the space savings because the space for character data is a small percentage of the total HashMap space; the rest is from the optimized HashMap design and eliminating Java object overheads. For example, we ran one test using random strings of length between 1 and 10, with an average length of 5.5. For the space-optimized version, we saw that the character storage (at 1 byte per character) took up 46% of the space; for the standard HashMap approach, the character storage (even at 2 bytes per character) took up only 6% of the space used.

Serializing/deserializing very large space-optimized objects—We balanced the need for fast load time and some amount of data compression when we designed serialization for these custom structures. General compression utilities were too slow; hence, we did variable-length encoding of integers to achieve some compression of the disk storage needed (which was often in the multigigabyte range). Use of NIO (New I/O) and memory-mapped I/O was very effective in reducing the loading time, sometimes from half an hour down to a couple of minutes.

Sesame RDF triple store improvements to in-memory version—Watson often makes use of a large amount of data stored in RDF (Resource Description Framework) triple formats. Starting with the in-memory Sesame [8] implementation, we were able to shrink the space used for this to approximately one-half the size by removing features not needed such as run time updates and using some of the space-saving techniques above. Using NIO and a custom external representation sped up loading by a factor of ~20.

Dynamically adjusting underlying object representation—One use case involved storing millions of integer sets, and performing various operations on them, including doing set intersection.

The distribution of the sizes of these sets followed the distribution frequency of the use of particular words in particular contexts. Hence, for a vast number of these, the sizes were very small, but for some significant number of these, the size was quite large, in the 10,000 and up range.

To accommodate this usage, we built a special integer set capability that stored sets as a sorted `int[]` for the smaller sizes, and as a special optimized hash map over ints (using the above techniques) for larger sizes, with the representation being automatically converted if the set grew above a threshold. Set intersection operations used different algorithms, depending on the type of arguments. In our use cases, the sets were constructed at load/setup time, once, and then repeatedly referenced, without being further updated; therefore, the cost to convert from the `int[]` representation to the other form happened only infrequently, and was insignificant.

Java garbage collection (GC) with large heaps—Large heap sizes present particular garbage collection issues for real-time systems: long pauses of the application during full GC events, often over a minute. These are not acceptable in a game-playing situation. GC performance can be optimized by adjusting the sizes of the old generation (which stores long-lived objects) and the new generation (which stores recently created objects).

The old generation clearly must be big enough to store Watson's "permanent" data resources such as the read-only data structures described in the previous section. During the processing of a question, many temporary objects are created that pertain only to that question, and we would like these to exist only in the new generation and not "leak" to the old generation where they would accumulate over time. This is important because if the old generation becomes full, the JVM** (Java Virtual Machine) completely blocks the application while performing a full GC.

There are two competing objectives for the sizing of the generations. A larger new generation prevents the temporary objects from leaking into the old generation and eventually causing a full GC. On the other hand, the larger the new generation, the longer the pauses for "minor" GCs within the new generation. We found that these minor collections could take hundreds of milliseconds, which is still unacceptable when competing against champion-level players.

Our solution for the large heap Java processes was to take advantage of the fact that Watson only needed to play one or two Jeopardy! games at a time, after which there was enough idle time to do a full GC. Therefore, we set the new generation size to be small enough that the minor collection time was very short, although a small amount of leakage to the old generation did occur. Between games, we forced a full GC event, ensuring that a full GC would not occur in the middle of a game.

Speeding up Indri Passage search

Key Watson primary search components for finding candidate answers use Indri Passage search to find the most relevant passages (one or two sentences of text) out of the Watson corpus that are most relevant. Using a single CPU to search the Watson corpus could easily take 100 seconds to complete.

Indri Passage search is also used to find additional evidence supporting the most promising candidate answers. Supporting evidence searches are an order of magnitude faster because they include the candidate answer as a required term in the query, which drastically reduces the number of passages that need to be considered by Indri. However, since there are many promising candidate answers per question, overall, these searches are even more computationally intensive.

Indri works by creating inverted indexes from the raw text corpus as a preprocessing step and then using these indexes to retrieve and score passages within documents at run time. Passage queries make extensive use of the Indri term proximity operators that require accessing position information in the index during query processing. An Indri search index is roughly the same size as the original corpus.

To achieve the required high-speed search query performance in DeepQA, the 50-GB corpus of 6.8 million documents was divided into 79 collections of approximately 100,000 documents and each indexed as a separate "chunk." A bank of 79 Indri search daemons, each hosting one chunk and running on eight CPU cores, is distributed across the cluster. Indexes are hosted in RAM by way of file system buffers. To handle the many supporting evidence queries occurring in parallel, we deployed two independent banks of search daemons. Maximum throughput was achieved with 16 clients using each bank; hence, the production system could run up to 32 passage queries at the same time.

The latency of distributed search is only as good as the search time of the slowest chunk. It is critical that the chunks are balanced not only in terms of index size and number of documents but also in terms of frequency distribution. Since some DeepQA corpora have a distinct vocabulary (e.g., movie databases use terms such as "actor" and "played" more frequently than other corpora), we spread the documents from each corpus evenly across all the chunks. Furthermore, it is critical to avoid searching on "stop-words" (very common terms such as "and" or "the"). Because they are useful as part of multiword phrase queries, stop-words were not removed at index time but, instead, were eliminated from the query. To determine which terms to remove, we systematically measured the impact on query speed of the most frequently occurring terms and bigrams and constructed a list of those that were unacceptably slow.

Originally, we used Java code on top of Indri to rerank the passages (giving more weight to passages that cover more of the query terms rather than repeat a single query term multiple times) and to expand the passages to sentence boundaries. We were able to increase query speed by pushing both of these operations into Indri. The reranking was implemented by using Indri's capability of plugging in a custom scoring function, and the sentence boundary expansion by precomputing the sentence boundaries over the entire corpus, indexing them, and extending Indri to return them.

Finally, once the search is complete, the passage text and metadata needed to be retrieved from the Indri index, and we observed that in the distributed search environment, the time to access this data can take longer than the passage search itself. Performance was limited by both network and JNI (Java Native Interface) serialization overhead. Instead, metadata lookup is implemented using the highly optimized Java HashMap described earlier, and passage text is retrieved from the content server described below.

Corpora preprocessing and custom content services

In order to use passage hits, Watson does a deep NLP analysis of the passage text [9]. This analysis includes semantic and structural parses, finding and resolving coreferences, identifying named-entities, among others. After eliminating computation bottlenecks in all other components, passage analysis constituted over half of the remaining CPU utilization.

Given that Watson was mandated to be a closed system, i.e., all of its knowledge self-contained, it made sense to move the deep NLP work from run time to a preprocessing stage. This section details the preprocessing effort and the mechanism for retrieving this data at run time.

Retrieval process for preprocessed data

The preprocessed data was ~5 times bigger than the raw text corpus. Together with the text, this required approximately 300 GB, much more than could fit in any of our 32-GB RAM machines. We took the approach of building a custom "content server" for this data and allocated a set of 14 machines that would each hold a portion.

With 14 machines, each one had to hold approximately 20 GB of data. The machines are connected over the network to hundreds of clients that request specific portions of documents, using standard TCP/IP socket protocols. The server code uses Java NIO in nonblocking mode to support multiple connections operating at once.

In Watson, each document has a unique document identifier, an integer, which is sequential within each corpus. We use this document identifier, plus a range of one or more sentences, as the "key" when looking up data in the content servers. The content is distributed round-robin by document id among the content server machines; the client code does the trivial computation to figure out which server to ask for a particular document. It then sends a TCP/IP request along an earlier established socket to the content server, which then uses the document id plus the range of sentences to index into arrays to find the data to send back.

The data is stored in these arrays in exactly the form needed to send out on the wire; hence, there is no "serialization" cost; a Java NIO nonblocking chained socket write operation is used to send the length plus the data in one request.

The same client-server implementation used by the content server for retrieving preprocessed corpora was also used for accessing PRISMATIC data [10].

Preprocessing the corpora

Prior to competing, Watson "reads" the corpora and sends it through the preprocessing UIMA pipeline. The preprocessing occurs prior to running the system. It can take many days of computing running on 100's of CPUs to process all of the corpora. This process also does the computation of sentence boundaries that are needed by the search indexing processes.

To scale out the processing across all of the documents in the corpora, we employed Apache Hadoop** [11]. We installed Hadoop on approximately 50 machines, each having eight cores, and 16 GB of RAM shared among the eight cores. The data was copied to HDFS (Hadoop Distributed File System), where it was replicated and stored over all machines. This allowed the machines to often have their data locally available on the same machine.

Hadoop supports a Map/Sort/Reduce model. We used the Map step to run a UIMA pipeline, with the unit of work being one document. At the end of the UIMA pipeline, we had an extraction step that created and serialized out a Java object representing the results of all the annotations that the pipeline created. The Sort phase arranged the items for each reducer sorted by corpus and then by unique document identifier. The Reduce phase used one reducer for each target content server machine and stored the data in a semicompressed and easily loadable format into a disk file, one per corpus. These files were then copied to the content server's local disk and used to initialize each content server.

The normal way Hadoop expects to scale up within a machine having multiple cores is to replicate the mappers; thus, for an eight-core machine, you might expect to run eight mappers. However, Hadoop implements this by running eight JVMs, one per mapper. In the case of our UIMA annotations, the footprint of our annotators was approaching the size of the physical memory on the machines; hence, running eight of them would not work; the machine would be constantly "swapping" as the working set size would greatly exceed the physical memory. Since our annotators are implemented to share the large resources within a JVM, we implemented a new Hadoop Map Runner that would instantiate and run multiple instances of the same UIMA pipeline within one JVM. This achieved near 100% CPU utilization.

Conclusion

The Watson QA application, which is based on the inherently parallel DeepQA architecture, effectively scales out across many machines. An average latency under 3 seconds was achieved using 2,300 CPU cores working together to answer each question. UIMA was effective for integrating over 100 different analytic components into a single application and facilitating a flexible development environment. Using UIMA-AS, the same components were deployed without any changes across 71 machines, each having 32 CPU cores, with insignificant scale-out overhead. UIMA CAS Multipliers were key for both organizing analysis results and achieving parallelism. CAS Views, another UIMA data organization facility, also proved useful in eliminating scale-out overhead. The major challenge to making Watson fast was optimizing the analytics themselves. There was no "silver bullet" to making the analytics faster; multiple approaches were needed, and the most important of which are described in this paper: computational parallelism, eliminating disk I/O by loading large data sets into RAM, moving analysis work from run time to preprocessing, and careful tuning of passage search. Finally, building the right performance measurements into the application was essential to driving the optimization work.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Jeopardy Productions, Inc., Apache Software Foundation, or Sun Microsystems in the United States, other countries, or both.

References

1. J. W. Murdock, J. Fan, A. Lally, H. Shima, and B. K. Boguraev, "Textual evidence gathering and analysis," *IBM J. Res. & Dev.*, vol. 56, no. 3/4, Paper 8, pp. 8:1–8:14, May/Jul. 2012.
2. J. Chu-Carroll, J. Prager, C. Welty, K. Czuba, and D. Ferrucci, "A multi-strategy and multi-source approach to question answering," in *Proc. 11th Text Retrieval Conf.*, 2002, pp. 1–8. [Online]. Available: http://trec.nist.gov/pubs/trec11/papers/ibm.prager.pdf
3. D. Moldovan, C. Clark, S. Harabagiu, and S. Maiorano, "COGEX: A logic prover for question answering," in *Proc. HLT-NAACL*, 2003, pp. 87–93.
4. D. A. Ferrucci, "Introduction to 'This is Watson'," *IBM J. Res. & Dev.*, vol. 56, no. 3/4, Paper 1, pp. 1:1–1:15, May/Jul. 2012.
5. Apache UIMA Project. [Online]. Available: http://uima.apache.org
6. Apache UIMA Asynchronous Scaleout. [Online]. Available: http://uima.apache.org/doc-uimaas-what.html
7. Indri search engine. [Online]. Available: http://www.lemurproject.org/indri/
8. Sesame framework for processing RDF data. [Online]. Available: http://www.openRDF.org
9. M. C. McCord, J. W. Murdock, and B. K. Boguraev, "Deep parsing in Watson," *IBM J. Res. & Dev.*, vol. 56, no. 3/4, Paper 3, pp. 3:1–3:15, May/Jul. 2012.
10. J. Fan, A. Kalyanpur, D. C. Gondek, and D. A. Ferrucci, "Automatic knowledge extraction from documents," *IBM J. Res. & Dev.*, vol. 56, no. 3/4, Paper 5, pp. 5:1–5:10, May/Jul. 2012.
11. Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

Received July 18, 2011; accepted for publication December 20, 2011

Edward A. Epstein *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (eae@us.ibm.com).* Mr. Epstein is a Software Development Manager in the Computer Science department at the T. J. Watson Research Center. He received a B.E.E. degree from Georgia Tech in 1971 and subsequently joined Technicon Instruments Corp in Tarrytown, New York, where he worked on the development of the first automated white blood cell differential system based on flow-through cytochemical analysis. In 1984, he joined the IBM Continuous Speech Recognition Group at the T. J. Watson Research Center, where he contributed to the creation of the IBM Tangora System, the world's first large vocabulary automatic speech recognition system, and then he led the group responsible for creating IBM's ViaVoice* speech recognition engine. For the past seven years, he has been Manager of the IBM team performing ongoing development of Apache Unstructured Information Management Architecture, originally an internal IBM Research project, which is now Apache UIMA. Most recently, he led the effort to scale out IBM's Watson question-answering analytic software over thousands of compute cores in order to compete against human Jeopardy! champions.

Marshall I. Schor *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (schor@us.ibm.com).* Mr. Schor is a Senior Technical Staff Member in the Semantic Analysis and Integration Department at the IBM T. J. Watson Research Center. He received a B.S. degree in engineering from the California Institute of Technology in 1968, and he did a special two-week intensive M.B.A. for technical leaders at the Kellogg School, Northwestern University, in 1999. He has held many positions within IBM, including Senior Manager within the Mathematical Sciences department, where he oversaw the initial research in applying machine learning approaches to text understanding, and he served for several years as the IBM Research liaison to the IBM Business Intelligence Solutions unit. He is currently Chair of the IBM Unstructured Information Management Architecture Board and a Vice President in the Apache Software Foundation, where he serves as Chairperson of the Apache UIMA project. He implemented many of the special space-saving algorithms used in Watson, as well as the content server, and set up and ran the Hadoop preprocessing pipelines.

Bhavani S. Iyer *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (bsiyer@us.ibm.com).* Ms. Iyer is a Software Engineer in the Unstructured Information Management Department at the IBM T. J. Watson Research Center. She received an M.S. degree in computer science from Pace University and subsequently joined IBM, where she has worked on developing database applications and distributed systems. She is a contributor and committer to the Apache UIMA project and is the primary developer of the UIMA C++ framework.

Adam Lally *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (alally@us.ibm.com).* Mr. Lally is a Senior Technical Staff Member at the IBM T. J. Watson Research Center. He received a B.S. degree in computer science from Rensselaer Polytechnic Institute in 1998 and an M.S. degree in computer science from Columbia University in 2006. As a member of the IBM DeepQA Algorithms Team, he helped develop the Watson system architecture that gave the machine its speed. He also worked on the natural-language processing algorithms that enable Watson to understand questions and categories and gather and assess evidence in natural language. Before working on Watson, he was the lead software engineer for the Unstructured Information Management Architecture project, an open-source platform for creating, integrating, and deploying unstructured information management solutions.

Eric W. Brown *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (ewb@us.ibm.com).* Dr. Brown is a Research Staff Member in the Semantic Analysis and Integration Department at the IBM T. J. Watson Research Center. He received a B.S. degree in computer science from the University of Vermont in 1989 and M.S. and Ph.D. degrees in computer science from the University of Massachusetts, Amherst, in 1992 and 1996, respectively. He subsequently joined IBM, where he has worked on information retrieval, text analysis, and question answering. Since 2007, he has been a technical lead on the Watson project. He is a member of the Association for Computing Machinery and Sigma Xi.

Jaroslaw Cwiklik *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (cwiklik@us.ibm.com).* Mr. Cwiklik is a Senior Software Engineer in the Unstructured Information department at the T. J. Watson Research Center. He received a B.S. degree in computer science from Albany State University in 1990. He joined IBM at the T. J. Watson Research Center in 1997. He is a member of the Systems Team working on the Watson project. He designed and developed Unstructured Information Management Architecture (UIMA) AS, the key infrastructure component for scaling out Watson technology to meet latency requirements and to allow Watson to scale its analytical computations across thousands of compute cores. Mr. Cwiklik is an Apache Open Source Committer for the UIMA project.