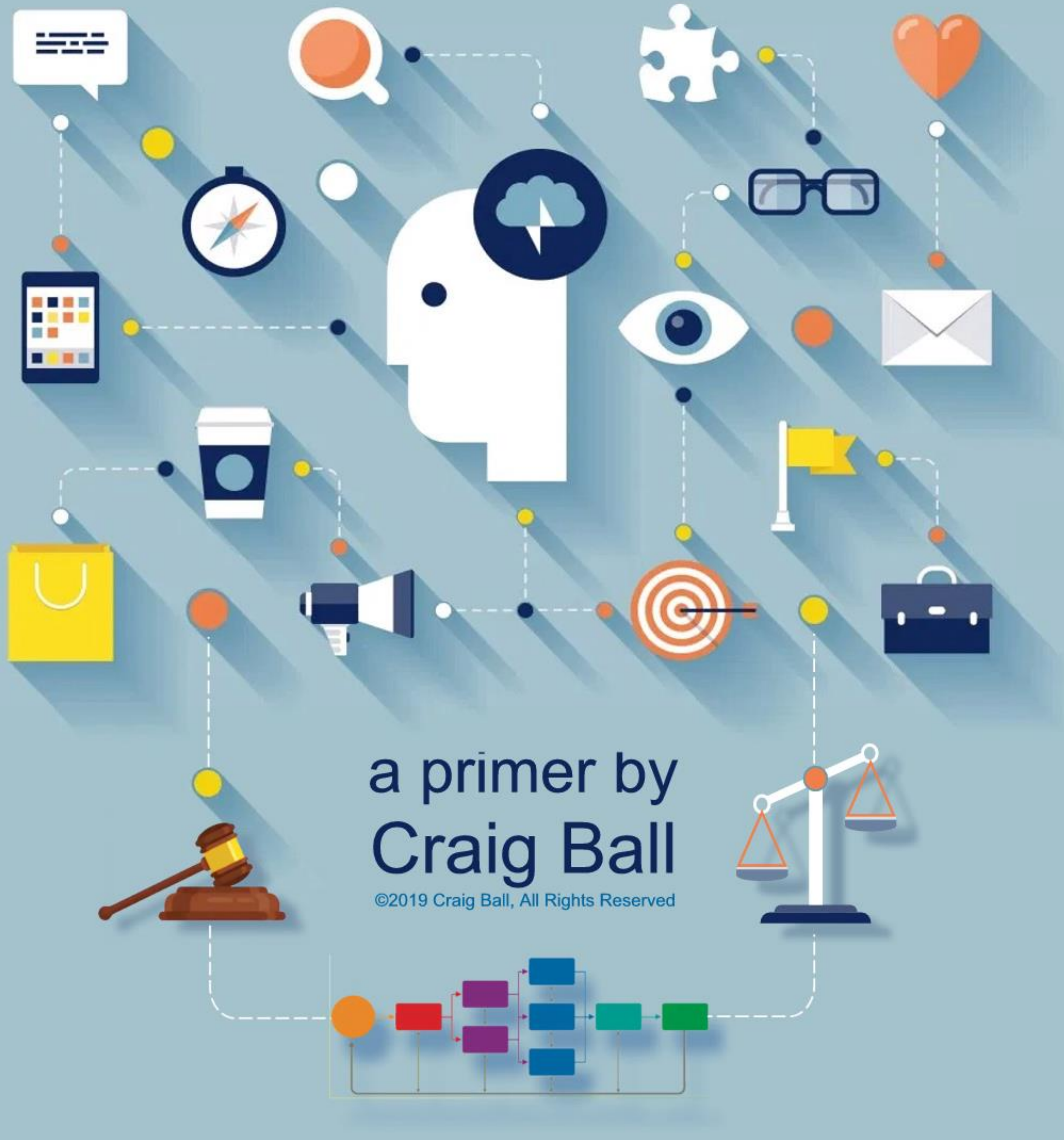
Processing in E-Discovery



a primer by

Craig Ball

Processing in E-Discovery, a Primer

Craig Ball[1]

Table of Contents

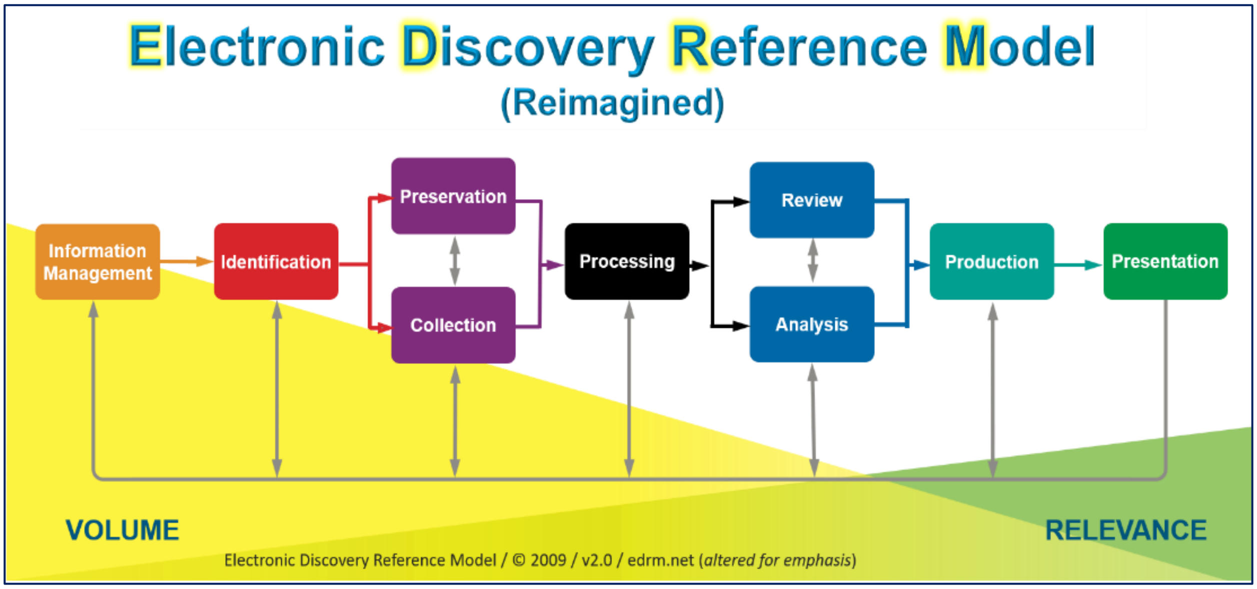
[1] Adjunct Professor, E-Discovery & Digital Evidence, University of Texas School of Law, Austin, Texas. The author acknowledges the contributions of Diana Ball as editor and of David Sitsky of Nuix for his insights into processing.

Introduction

Talk to lawyers about e-discovery processing and you'll likely get a blank stare suggesting no clue what you're talking about. Why would lawyers want anything to do with something so disagreeably technical? Indeed, processing is technical and strikes attorneys as something they need not know. That's lamentable because processing is a stage of e-discovery where things can go terribly awry in terms of cost and outcome. Lawyers who understand the fundamentals of ESI processing are better situated to avoid costly mistakes and resolve them when they happen. This paper looks closely at ESI processing and seeks to unravel its mysteries.

Processing is the "black box" between preservation/collection and review/analysis. Though the iconic Electronic Discovery Reference Model (EDRM) positions Processing, Review and Analysis as parallel paths to Production, processing is an essential prerequisite—"the only road"—to Review, Analysis and Production.[2] Any way you approach e-discovery at scale, you must process ESI before you can review or analyze it. If we recast the EDRM to reflect processing's centrality, the famed schematic would look like this:

[2] That's not a flaw. The EDRM is a conceptual view, not a workflow.



There are hundreds—perhaps thousands—of articles delving into the stages of e-discovery that flank processing in the EDRM. These are the stages where lawyers have had a job to do. But lawyers tend to cede processing decisions to technicians. When it comes to processing, lawyer competency and ken is practically non-existent, little more than "stuff goes in, stuff comes out."

Why process ESI in e-discovery? Isn't it "review ready?"

We process information in e-discovery to catalog and index contents for search and review. Unlike Google, e-discovery is the search for *all* responsive information in a collection, not just one information item deemed responsive. Though all electronically stored information is inherently electronically searchable, computers don't structure or search all ESI in the same way; so, we must process ESI to ***normalize*** it to achieve uniformity for indexing and search.

Thus, "processing" in e-discovery could be called "normalized access," in the sense of extracting content, decoding it and managing and presenting content in consistent ways for access and review. It encompasses the steps required to extract text and metadata from information items and build a searchable index. The closest analogy is the creation of a Google-like capability with respect to a discrete collection of documents, data and metadata.

ESI processing tools perform five common functions.[4] They must:

1) Decompress, unpack and fully explore, *i.e.,* **recurse** ingested items.

[4] While a processing tool may do considerably more than the listed functions, a tool that does less is unlikely to meet litigants' needs in e-discovery.

2) **Identify** and apply templates (filters) to **encoded** data to **parse** (interpret) contents and **extract text, embedded objects**, and **metadata**.

3) **Track** and **hash** items processed, **enumerating** and **unitizing** all items and tracking failures.

4) **Normalize** and **tokenize** text and data and create an **index** and **database** of extracted information.

5) **Cull** data by file type, date, lexical content, hash value and other criteria.

These steps are a more detailed description of processing than I've seen published; but I mean to go deeper, diving into *how* it works and exposing situations where it may fail to meet expectations.[5]

Files

If we polled lawyers asking what to call the electronic items preserved, collected and processed in discovery, most would answer, "documents." Others might opt for "data" or reflect on the initialization "ESI" and say, "information." None are wrong answers, but the ideal response would be the rarest: "*files*." Electronic documents are *files*. Electronically stored information resides in *files*. *Everything* we deal with *digitally* in electronic discovery comes from or goes to physical or logical data storage units called "data files" or just "files." Moreover, all programs run against data files are themselves files comprising instructions for tasks. These are "executable files" or simply "executables."

So, what is it we process in the processing stage of e-discovery? The answer is, "**we process *files***." Let's look at these all-important files and explore what's in them and how are they work.

A Bit About and a Byte Out of Files

A colleague once defended her ignorance of the technical fundamentals of electronically stored information by analogizing that "she didn't need to know how planes stay aloft to fly on one." She had a point, but only for passengers. If you aspire to be a pilot or a rocket scientist—if you want to be at the controls or design the plane—you must understand the fundamentals of flight. If you aspire to understand processing of ESI in e-discovery and manage e-discovery, you must understand the fundamentals of electronically stored information, including such topics as:

- *What's* stored electronically?
- *How* is it stored?

[5] Though I commend the abandoned efforts of the EDRM to educate via their Processing Standards Guide, still in its public comment phase five+ years after publication. https://www.edrm.net/resources/frameworks-and-standards/edrm-model/edrm-stages-standards/edrm-processing-standards-guide-version-1/

- *What forms* does it take?

The next few pages are a crash course in ESI storage, particularly the basics of encoding and recording textual information. If you can tough it out, you'll be undaunted by discussions of "Hex Magic Numbers" and "Unicode Normalization" yet to come.

Digital Encoding

All digital evidence is encoded, and how it's encoded bears upon how it's collected and processed, whether it can be searched and in what reasonably usable forms it can be produced. Understanding that electronically stored information is numerically encoded data helps us see the interplay and interchangeability between different forms of digital evidence. Saying "*it's all ones and zeroes*" means nothing if you don't grasp *how* those ones and zeros underpin the evidence.

Decimal and Binary: Base 10 and Base Two

When we were children starting to count, we had to *learn* the decimal system. We had to *think* about what numbers *meant*. When our first-grade selves tackled big numbers like 9,465, we were acutely aware that each digit represented a decimal multiple. The nine was in the thousands place, the four in the hundreds, the six in the tens place and the five in the ones. We might even have parsed 9,465 as: *(9 x 1000) + (4 x 100) + (6 x 10) + (5 x 1)*.

But soon, it became second nature to us. We'd unconsciously process 9,465 as nine thousand four hundred sixty-five. As we matured, we learned about powers of ten and now saw 9,465 as: $(9 \times 10^3) + (4 \times 10^2) + (6 \times 10^1) + (5 \times 10^0)$. This was exponential or "base ten" notation. We flushed it from our adolescent brains as fast as life (and the SAT) allowed.

Humankind probably uses base ten to count because we evolved with ten fingers. But, had we slithered from the ooze with eight or twelve digits, we'd have gotten on splendidly using a base eight or base twelve number system. It really wouldn't matter because ***you can express any number—and consequently any data—in any number system***. So, it happens that computers use the base two or binary system, and computer programmers are partial to base sixteen or hexadecimal. ***Data is just numbers, and it's all just counting.*** The **radix** or base is the number of unique digits, including the digit zero, used to represent numbers in a positional numeral system. For the decimal system the radix is ten, because it uses the ten digits, 0 through 9.

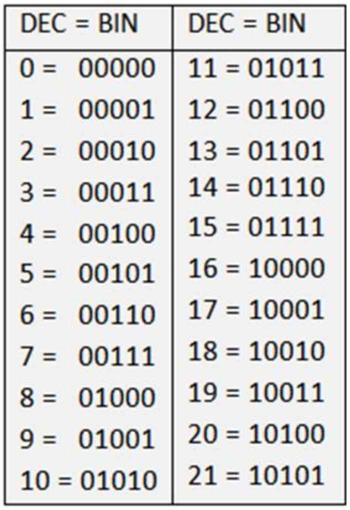
Bits

Computers use **binary digits** in place of decimal digits. The word **bit** is even a shortening of the words "**B**inary dig**IT**." Unlike the decimal system, where we represent any number as a combination of *ten* possible digits (0-9), the binary system uses only *two* possible values: zero or one. This is not as limiting as one might expect when you consider that a digital circuit—

essentially an unfathomably complex array of switches—hasn't got any fingers to count on but is very good and very fast at being "on" or "off."

In the binary system, each binary digit—each "bit"—holds the value of a power of two. Therefore, a binary number is composed of only zeroes and ones, like this: 10101. How do you figure out what the value of the binary number 10101 is? You do it in the same way we did it above for 9,465, but you use a base of 2 instead of a base of 10. Hence: (1 x 2^4) + (0 x 2^3) + (1 x 2^2) + (0 x 2^1) + (1 x 2^0) = 16 + 0 + 4 + 0 + 1 = **21**.

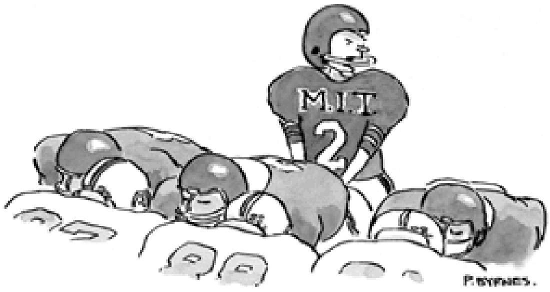
Moving from right to left, each bit you encounter represents the value of increasing powers of 2, standing in for zero, two, four, eight, sixteen, thirty-two, sixty-four and so on. That makes counting in binary easy. Starting at zero and going through 21, decimal and binary equivalents look like the table at right.



DEC = BIN	DEC = BIN
0 = 00000	11 = 01011
1 = 00001	12 = 01100
2 = 00010	13 = 01101
3 = 00011	14 = 01110
4 = 00100	15 = 01111
5 = 00101	16 = 10000
6 = 00110	17 = 10001
7 = 00111	18 = 10010
8 = 01000	19 = 10011
9 = 01001	20 = 10100
10 = 01010	21 = 10101

Bytes

A byte is a string (sequence) of eight bits. The biggest number that can be stored as one byte of information is 11111111, equal to 255 in the decimal system. The smallest number is zero or 00000000. Thus, you can store 256 different numbers as one byte of information. So, what do you do if you need to store a number larger than 256? Simple! You use a second byte. This affords you all the combinations that can be achieved with 16 bits, being the product of all of the variations of the first byte and all of the second byte (256 x 256 or 65,536). So, using bytes to express values, we express any number greater than 256 using at least two bytes (called a "word" in geek speak), and any number above 65,536 requires three bytes or more. A value greater than 16,777,216 (256^3, the same as 2^{24}) needs four bytes (called a "long word") and so on.



"01101001, 00111011, 00011010, but, but!"

Let's try it: Suppose we want to represent the number 51,975. It's 1100101100000111, *viz*:

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8		2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
32768	16384	8192	4096	2048	1024	512	256		128	64	32	16	8	4	2	1
1	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**+**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**
(32768+16384+2048+512+256) or **51,968**								**+**	(4+2+1) or **7**							

Why are eight-bit sequences the fundamental building blocks of computing? It just happened that way. In these times of cheap memory, expansive storage and lightning-fast processors, it's easy to forget how scarce and costly such resources were at the dawn of the computing era. Seven bits (with a leading bit reserved) was the smallest block of data that would suffice to represent the minimum complement of alphabetic characters, decimal digits, punctuation and control instructions needed by the pioneers in computer engineering. It was, in a sense, all the data early processors could bite off at a time, perhaps explaining the name "byte" (coined in 1956 by IBM scientist Dr. Werner Buchholz).

Werner Buchholz

The Magic Decoder Ring called ASCII

Back in 1935, American kids who listened to the Little Orphan Annie radio show and drank lots of Ovaltine could join the Radio Orphan Annie Secret Society and obtain a Magic Decoder Ring, a device with rotating disks that allowed them to read and write numerically-encoded messages.

Similarly, computers encode words as numbers. Binary data stand in for the upper- and lower-case English alphabet, as well as punctuation marks, special characters and machine instructions (like carriage return and line feed). The most widely deployed U.S. encoding mechanism is known as the **ASCII** code (for **American Standard Code for Information Interchange,** pronounced "ask-key"). By limiting the ASCII character set to just 128 characters, we can express any character in just seven bits (2^7 or 128) and so occupy only one byte in the computer's storage and memory. In the Binary-to-ASCII Table below, the columns reflect a binary (byte) value, its decimal equivalent and the corresponding ASCII text value (including some for machine codes and punctuation):

Binary-to-ASCII Table

Binary	Decimal	Character	Binary	Decimal	Character	Binary	Decimal	Character
00000000	000	NUL	00101011	043	+	01010110	086	V
00000001	001	SOH	00101100	044	,	01010111	087	W
00000010	002	STX	00101101	045	-	01011000	088	X
00000011	003	ETX	00101110	046	.	01011001	089	Y
00000100	004	EOT	00101111	047	/	01011010	090	Z

Binary	Decimal	Character
00000101	005	ENQ
00000110	006	ACK
00000111	007	BEL
00001000	008	BS
00001001	009	HT
00001010	010	LF
00001011	011	VT
00001100	012	FF
00001101	013	CR
00001110	014	SO
00001111	015	SI
00010000	016	DLE
00010001	017	DC1
00010010	018	DC2
00010011	019	DC3
00010100	020	DC4
00010101	021	NAK
00010110	022	SYN
00010111	023	ETB
00011000	024	CAN
00011001	025	EM
00011010	026	SUB
00011011	027	ESC
00011100	028	FS
00011101	029	GS
00011110	030	RS
00110000	048	0
00110001	049	1
00110010	050	2
00110011	051	3
00110100	052	4
00110101	053	5
00110110	054	6
00110111	055	7
00111000	056	8
00111001	057	9
00111010	058	:
00111011	059	;
00111100	060	<
00111101	061	=
00111110	062	>
00111111	063	?
01000000	064	@
01000001	065	A
01000010	066	B
01000011	067	C
01000100	068	D
01000101	069	E
01000110	070	F
01000111	071	G
01001000	072	H
01001001	073	I
01011011	091	[
01011100	092	\
01011101	093	]
01011110	094	^
01011111	095	_
01100000	096	`
01100001	097	a
01100010	098	b
01100011	099	c
01100100	100	d
01100101	101	e
01100110	102	f
01100111	103	g
01101000	104	h
01101001	105	i
01101010	106	j
01101011	107	k
01101100	108	l
01101101	109	m
01101110	110	n
01101111	111	o
01110000	112	p
01110001	113	q
01110010	114	r
01110011	115	s
01110100	116	t

Binary	Decimal	Character	Binary	Decimal	Character	Binary	Decimal	Character
00011111	031	US	01001010	074	J	01110101	117	u
00100000	032	SP	01001011	075	K	01110110	118	v
00100001	033	!	01001100	076	L	01110111	119	w
00100010	034	"	01001101	077	M	01111000	120	x
00100011	035	#	01001110	078	N	01111001	121	y
00100100	036	$	01001111	079	O	01111010	122	z
00100101	037	%	01010000	080	P	01111011	123	{
00100110	038	&	01010001	081	Q	01111100	124	|
00100111	039	'	01010010	082	R	01111101	125	}
00101000	040	(	01010011	083	S	01111110	126	~
00101001	041	)	01010100	084	T	01111111	127	DEL
00101010	042	*	01010101	085	U	**Note: 0-127 is 128 values**		

So, “E-Discovery” would be written in a binary ASCII sequence as:

0100010100101101**01000100**01101001**01110011**01100011**01101111**01110110**01100101**0111 0010**01111001**

It would be tough to remember your own name written in this manner! *Hi, I’m Craig, but my computer calls me **01000011**01110010**01100001**01101001**01100111.***

Note that each leading bit of each byte in the table above is a zero. It wasn’t used to convey any encoding information; that is, they are all *7-bit* bytes. In time, the eighth bit (the leading zero) would change to a one and was used to encode another 128 characters (2^8 or 256) , leading to various “extended” (or “high”) ASCII sets that include, *e.g.*, accented characters used in foreign languages and line drawing characters.

Unfortunately, these extra characters weren’t assigned in the same way by all computer systems. The emergence of different sets of characters mapped to the same high byte values prompted a need to identify these various **character encodings** or, as Microsoft calls them in Windows, these “**code pages**.” If an application used the wrong code page, information displayed as gibberish. This is such a familiar phenomenon that it has its own name, **mojibake** (from the Japanese for "character changing"). If you've encountered a bunch of Asian language characters in an e-mail or document you know was written in English, you might have glimpsed mojibake.

Note that we are speaking here of textual information, not typography; so, don't confuse character encodings with fonts. The former tells you whether the character is an A or b, not whether to display the character in Arial or Baskerville.

In the mid-1980s, international standards began to emerge for character encoding, ultimately resulting in various code sets issued by the International Standards Organization (ISO). These retained the first 128 American ASCII values and assigned the upper 128-byte values to characters suited to various languages (*e.g.,* Cyrillic, Greek, Arabic and Hebrew). ISO called these various character sets ISO-8859-*n,* where the *"n"* distinguished the sets for different languages. ISO-8859-1 was the set suited to Latin-derived alphabets (like English) and so the nickname for the most familiar code page to U.S. computer users became "**Latin 1**."

But Microsoft adopted the Windows code page before the ISO standard became final, basing its Latin 1 encoding on an earlier draft promulgated by the American National Standards Institute (ANSI). Thus, the standard Windows Latin-1 code page, called **Windows-1252 (ANSI),** is *mostly* identical to ISO-8859-1, and it's common to see the two referred to interchangeably as "Latin 1."

Unicode

ASCII dawned in the pre-Internet world of 1963—before the world was flat, when the West dominated commerce and personal computing was the stuff of science fiction. Using a single byte (even with various code pages) supported only 256 characters, so was incompatible with Asian languages like Chinese, Japanese and Korean employing thousands of pictograms and ideograms.

Though programmers developed various *ad hoc* approaches to foreign language encodings, we needed a universal, systematic encoding mechanism to serve an increasingly interconnected world. These methods would use **more than one byte** to represent each character. The most widely adopted such system is **Unicode**. In its latest incarnation (version 12.1), Unicode standardizes the encoding of 150 written languages called "scripts" comprising 127,994 characters, plus multiple symbol sets and emoji.

The Unicode Consortium crafted Unicode to co-exist with the longstanding ASCII and ANSI character sets by emulating the ASCII character set in corresponding byte values within the more extensible Unicode counterpart, **UTF-8**. Because of its backward compatibility and multilingual adaptability, UTF-8 has become a popular encoding standard, especially on the Internet and within e-mail systems.

Mind the Gap!

Now, as we talk about all these bytes and encoding standards as a precursor to hexadecimal notation, it will be helpful to revisit how this all fits together. A byte is eight ones or zeroes, which means a byte can represent 256 different decimal numbers from 0-255. So, two bytes can represent a much bigger range of decimal values (256 x 256 or 65,536). Character encodings (aka "code pages") like Latin 1 and UTF-8 are ways to map textual, graphical or machine instructions to numeric values expressed as bytes, enabling machines to store and communicate information in human languages. As we move forward, keep in mind that *hex, like binary and decimal, is just another way to write numbers*. Hex is not a code page, although the numeric values it represents may correspond to values *within* code pages. [6]

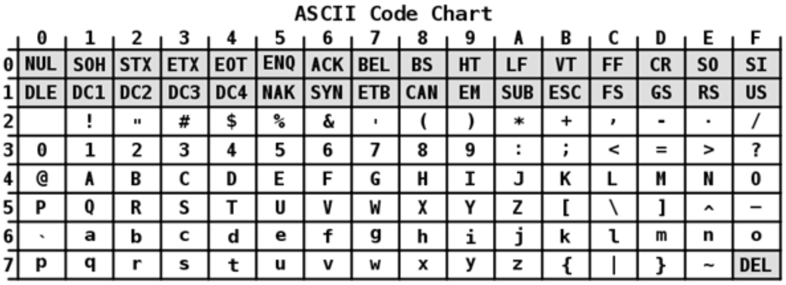
Hex

Long sequences of ones and zeroes are very confusing for people, so **hexadecimal notation** emerged as more accessible shorthand for binary sequences. Considering the prior discussion of base 10 (decimal) and base 2 (binary) notation, it might be enough to say that hexadecimal is base 16. In hexadecimal notation (**hex** for short), each digit can be any value from zero to fifteen. Accordingly, we can replace four binary digits with just a single hexadecimal digit, and more to the point, we can express a byte as just two hex characters.

The decimal system supplies only 10 symbols (0-9) to represent numbers. Hexadecimal notation demands 16 symbols, leaving us without enough single character *numeric* values to stand in for all the values in each column. So, how do we cram 16 values into each column? The solution was to substitute the letters A through F for the numbers 10 through 15. So, we can represent 10110101 (the decimal number 181) as "B5" in hexadecimal notation. Using hex, we can notate values from 0-255 as 00 to FF (using either lower- or upper-case letters; it doesn't matter).

It's hard to tell if a number is decimal or hexadecimal just by looking at it: if you see "37", does that equate to 37 ("37" in decimal) or a decimal 55 ("37" in hexadecimal)? To get around this problem, two common notations are used to indicate hexadecimal numbers. The first is the suffix of a lower-case "h." The second is the prefix of "0x." So "37 in hexadecimal," "37h" and "0x37" all mean the same thing.

The ASCII Code Chart at right can be used to express ASCII characters in hex. The capital letter "G" is encoded



ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

[6] Don't be put off by the math. The biggest impediment to getting through the encoding basics is the voice in your head screaming, "I SHOULDN'T HAVE TO KNOW ANY OF THIS!!" Ignore that voice. It's wrong.

as the hex value of 47 (i.e., row 4, column 7), so "**E-Discovery**" in hex encodes as:

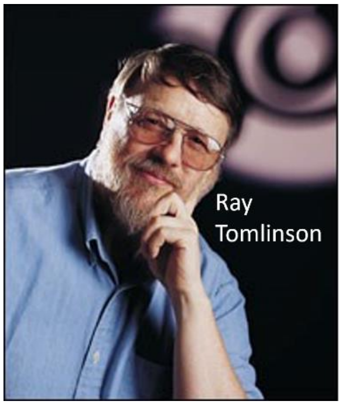
0x45 2D 44 69 73 63 6F 76 65 72 79[7]

That's easier than:

0100010100101101**01000100**01101001**01110011**01100011**01101111**01110110**01100101**01110010**01111001?**

Base64

Internet e-mail was born in 1971, when a researcher named Ray Tomlinson sent a message to himself using the "@" sign to distinguish the addressee from the machine. Tomlinson didn't remember the message transmitted in that historic first e-mail but speculated that it was probably something like "qwertyuiop." So, not exactly, "*Mr. Watson, come here. I need you*," but then, Tomlinson didn't *know* he was changing the world. He was just killing time.



Also, back when the nascent Internet consisted of just four university research computers, UCLA student Stephen Crocker originated the practice of circulating proposed technical standards (or "protocols" in geek speak) as publications called "Requests for Comments" or RFCs. They went via U.S. postal mail because there was no such thing as e-mail. Ever after, proposed standards establishing the format of e-mail were promulgated as numbered RFCs. So, when you hear an e-discovery vendor mention "RFC5322 content," fear not, it just means plain ol' e-mail.

An e-mail is as simple as a postcard. Like the back-left side of a postcard, an e-mail has an area called the message body reserved for the user's text message. Like a postcard's back right side, we devote another area called the message header to information needed to get the card where it's supposed to go and to transmittal data akin to a postmark.

We can liken the picture or drawing on the front of our postcard to an e-mail's attachment. Unlike a postcard, we must convert e-mail attachments to letters and numbers for transmission, enabling an e-mail to carry any type of electronic data — audio, documents, software, video — not just pretty pictures.

[7] The concept of byte order called **Endianness** warrants passing mention here. Computers store data in ways compatible with their digital memory registers. In some systems, the most significant digit (byte) comes first and in others, it's last. This **byte ordering** determines whether a hexadecimal value is written as 37 or 73. A **big-endian** ordering places the most significant byte first and the least significant byte last, while a **little-endian** ordering does the opposite. Consider the hex number 0x1234, which requires at least two bytes to represent. In a big-endian ordering they would be [0x12, 0x34], while in a little-endian ordering, the bytes are arranged [0x34, 0x12].

The key point is that *everything in any e-mail is plain text*, no matter what's attached.

And by plain text, I mean the plainest English text, 7-bit ASCII, lacking even the diacritical characters required for accented words in French or Spanish or any formatting capability. No **bold**. No underline. No *italics*. It is text so simple that you can store a letter as a single byte of data.

The dogged adherence to plain English text stems in part from the universal use of the Simple Mail Transfer Protocol or SMTP to transmit e-mail. SMTP only supports 7-bit ASCII characters, so sticking with SMTP maintained compatibility with older, simpler systems. Because it's just text, it's compatible with any e-mail system invented in the last 50 years. Think about that the next time you come across a floppy disk or CD and wonder how you're going to read it.

How do you encode a world of complex digital content into plain text without losing anything?

The answer is an encoding scheme called Base64, which substitutes 64 printable ASCII characters (A–Z, a–z, 0–9, + and /) for any binary data or for foreign characters, like Cyrillic or Chinese, that can be represented by the Latin alphabet.

Base64 is brilliant and amazingly simple. Since all digital data is stored as bits, and six bits can be arranged in 64 separate ways, you need just 64 alphanumeric characters to stand in for any six bits of data. The 26 lower case letters, 26 upper case letters and the numbers 0-9 give you 62 stand-ins. Throw in a couple of punctuation marks—say the forward slash and plus sign—and you have all the printable ASCII characters you need to represent any binary content in six bit chunks. Though the encoded data takes up roughly a third more space than its binary source, now any mail system can hand off any attachment. Once again, *it's all just numbers.*

Why Care about Encoding?

All this code page and Unicode stuff matters because of the vital role that electronic search plays in e-discovery. If we index an information item for search using the wrong character set, the information in the item won't be extracted, won't become part of the index and, accordingly, won't be found even when the correct search terms are run. Many older search tools and electronic records are not Unicode-compliant. Worse, because the content may include a smattering of ASCII text, an older search tool may conclude it's encountered a document encoded with Latin-1 and may fail to flag the item as having failed to index. In short, encoding issues carry real-world consequences, particularly when foreign language content is in the collection. If you aren't aware of the limits of the processing tools you, your opponents or service providers use, you can't negotiate successful protocols. What you don't know *can* hurt you.

The many ways in which data is encoded—and the diverse ways in which collection, processing and search tools identify the multifarious encoding schemes—lie at the heart of significant challenges and costly errors in e-discovery. It's easy to dismiss these fundamentals of information technology as too removed from litigation to be worth the effort to explore them; but, understanding encoding and the role it plays play in processing, indexing and search will help you realize the capabilities and limits of the tools you, your clients, vendors and opponents use.

Let's look at these issues and file formats through the lens of a programmer making decisions about how a file should function.

Hypothetical: TaggedyAnn

Ann, a programmer, must create a tool to generate nametags for attendees at an upcoming continuing legal education program. Ann wants her tool to support different label stock and multiple printers, fonts, font sizes and salutations. It will accept lists of pre-registrants as well as create name tags for those who register onsite. Ann will call her tool TaggedyAnn. TaggedyAnn has never existed, so no computer operating system "knows" what to do with TaggedyAnn data. Ann must design the necessary operations and associations.

Remembering our mantra, "ESI is just numbers," Ann must lay out those numbers in the data files so that the correct program—her TaggedyAnn program—will be executed against TaggedyAnn data files, and she must structure the data files so that the proper series of numbers will be pulled from the proper locations and interpreted in the intended way. Ann must develop the **file format**.

A file format establishes the way to encode and order data for storage in the file. Ideally, Ann will document her file format as a published specification, a blueprint for the file's construction. That way, anyone trying to develop applications to work with Ann's files will know what goes where and how to interpret the data. But Ann may never get around to writing a formal file specification or, if she believes her file format to be a trade secret, Ann may decide not to reveal its structure publically. In that event, others seeking to read TaggedyAnn files must reverse engineer the files to deduce their structure and fashion a document filter that can accurately parse and view the data.

In terms of complexity, file format specifications run the gamut. Ann's format will likely be simple—perhaps just a page or two—as TaggedyAnn supports few features and functions. By contrast, the published file specifications for the binary forms of Microsoft Excel, PowerPoint and Word files run to 1,124, 649 and 575 pages, respectively. Microsoft's latest file format specification for the .PST file that stores Outlook e-mail occupies 192 pages. Imagine trying to reverse engineer such complexity without a published specification! For a peek into the hidden structure of a Word .DOCX file, see Illustration 2: Anatomy of a Word DOCX File.

File Type Identification

In the context of the operating system, Ann must link her program and its data files through various means of **file type identification**.

File type identification using binary file signatures and file extensions is an essential early step in e-discovery processing. Determining file types is a necessary precursor to applying the appropriate document filter to extract contents and metadata for populating a database and indexing files for use in an e-discovery review platform.

File Extensions

Because Ann is coding her tool from scratch and her data files need only be intelligible to her program, she can structure the files any way she wishes, but she must nevertheless supply a way that computer operating systems can pair her data files with only her executable program. Else, there's no telling what program might run. Word can open a PDF file, but the result may be unintelligible.

Filename extensions or just "file extensions" are a means to identify the contents and purpose of a data file. Though executable files must carry the file extension .EXE, any ancillary files Ann creates can employ almost any three-letter or -number file extension Ann desires *so long as her choice doesn't conflict with one of the thousands of file extensions already in use*. That means that Ann can't use .TGA because it's already associated with Targa graphics files, and she can't use .TAG because that signals a DataFlex data file. So, Ann settles on .TGN as the file extension for TaggedyAnn files. That way, when a user loads a data file with the .TGN extension, Windows can direct its contents to the TaggedyAnn application and not to another program that won't correctly parse the data.

Binary File Signatures

But Ann needs to do more. Not all operating systems employ file extensions, and because extensions can be absent or altered, file extensions aren't the most reliable way to denote the purpose or content of a file. Too, file names and extensions are *system metadata*, so they are not stored inside the file. Instead, computers store file extensions within the *file table* of the storage medium's file system. Accordingly, Ann must fashion a unique **binary header signature** that precedes the contents of each TaggedyAnn data file in order that the contents are identifiable as TaggedyAnn data and directed solely to the TaggedyAnn program for execution.

A binary file signature (also called a **magic number**) will typically occupy the first few bytes of a file's contents. It will always be hexadecimal values; however, the hex values chosen may correspond to an intelligible alphanumeric (ASCII) sequence. For example, PDF document files begin 25 50 44 46 2D, which is **%PDF-** in ASCII. Files holding Microsoft tape archive data begin 54 41 50 45, translating to **TAPE** in ASCII. Sometimes, it's a convoluted correlation, *e.g.,* Adobe Shockwave Flash files have a file extension of SWF, but their file signature is 46 57 53, corresponding to **FWS** in ASCII. In other instances, there is no intent to generate a meaningful ASCII word or phrase using the binary header signatures; they are simply numbers in hex. For example, the header signature for a JPG image is FF D8 FF E0 which translates to the gibberish **ÿØÿà** in ASCII.

Binary file signatures often append characters that signal variants of the file type or versions of the associated software. For example, JPG images in the JPEG File Interchange Format (JFIF) use the binary signature FF D8 FF E0 00 10 4A 46 49 46 or **ÿØÿà JFIF** in ASCII. A JPG image in the JPEG Exchangeable Image File Format (Exif, commonly used by digital cameras) will start with the binary signature FF D8 FF E1 00 10 45 78 69 66 or **ÿØÿá Exif** in ASCII.

Ann peruses the lists of file signatures available online and initially thinks she will use Hex 54 41 47 21 as her binary header signature because no one else appears to be using that signature and it corresponds to **TAG!** in ASCII.

But, after reflecting on the volume of highly-compressible text that will comprise the program's data files, Ann instead decides to store the contents in a ZIP-compressed format, necessitating that the binary header signature for TaggedyAnn's data files be 50 4B 03 04, **PK..** in ASCII. All files compressed with ZIP use the **PK..** header signature because Phil Katz, the programmer who wrote the ZIP compression tool, chose to flag his file format with his initials.

How can Ann ensure that only TaggedyAnn opens these files and they are not misdirected to an unzipping (decompression) program? Simple. Ann will use the .TGN extension to prompt Windows to load the file in TaggedyAnn and TaggedyAnn will then read the **PK..** signature and unzip the contents to access the compressed contents.

But, what about when an e-discovery service provider processes the TaggedyAnn files with the mismatched file extensions and binary signatures? Won't that throw things off? It could. We'll return to that issue when we cover compound files and recursion. First, let's touch on file structures.

File Structure

Now, Ann must decide how she will structure the data within her files. Once more, Ann's files need only be intelligible to her application, so she is unconstrained in her architecture. This point becomes important as the differences in file structure are what make processing and indexing essential to electronic search. There are thousands of different file types, each structured in idiosyncratic ways. Processing normalizes their contents to a common searchable format.

Some programmers dump data into files in so consistent a way that programs retrieve particular data by offset addressing; that is, by beginning retrieval at a specified number of bytes from the start of the file (*offset* from the start) and retrieving a specified extent of data from that offset (*i.e.,* grabbing *X* bytes following the specified offset).

Offset addressing could make it hard for Ann to add new options and features, so she may prefer to implement a chunk- or directory-based structure. In the first approach, data is labeled within the file to indicate its beginning and end or it may be tagged ("marked up") for identification. The program accessing the data simply traverses the file seeking the tagged data it requires and grabs the data between tags. There are many ways to implement a chunk-based structure, and it's probably the most common file structure. A directory-based approach constructs a file as a small operating environment. The directory keeps track of what's in the file, what it's called and where it begins and ends. Examples of directory-based formats are ZIP archive files and Microsoft Office files after Office 2007. Ann elects to use a mix of both approaches. Using ZIP entails a directory and folder structure, and she will use tagged, chunked data within the compressed folder and file hierarchy.

Data Compression

Many common file formats and containers are compressed, necessitating that e-discovery processing tools be able to identify compressed files and apply the correct decompression algorithm to extract contents.

Compression is miraculous. It makes modern digital life possible. Without compression, we wouldn't have smart phones, digitized music, streaming video or digital photography. Without compression, the web would be a different, duller place.

Compression uses algorithms to reduce the space required to store and the bandwidth required to transmit electronic information. If the algorithm preserves all compressed data, it's termed "**lossless compression**." If the algorithm jettisons data deemed expendable, it's termed "**lossy compression**."

JPEG image compression is lossy compression, executing a tradeoff between image quality and file size. Not all the original photo's graphical information survives JPEG compression. Sharpness and color depth are sacrificed, and compression introduces distortion in the form of rough margins called "jaggies." We likewise see a loss of fidelity when audio or video data is compressed for storage or transmission (*e.g.*, as MPEG or MP3 files). The offsetting benefit is that the smaller file sizes facilitate streaming video and storing thousands of songs in your pocket.

In contrast, file compression algorithms like ZIP are lossless; decompressed files perfectly match their counterparts before compression. Let's explore how that's done.

One simple approach is **Run-Length Encoding**. It works especially well for images containing consecutive, identical data elements, like the ample white space of a fax transmission. Consider a black and white graphic where each pixel is either B or W; for example, the image of the uppercase letter "E," below left:

```
WWBBBBBWW
WWBBWWWWW
WWBBBBWWW
WWBBWWWWW
WWBBBBBWW
```

The image at left requires 45 characters but we can write it in 15 fewer characters by adding a number describing each sequence or "run,“ *i.e.,* 2 white pixels, 5 black, 2 white.

```
2W5B2W
2W2B5W
2W4B3W
2W2B5W
2W5B2W
```

We've compressed the data by a third. Refining our run-length compression, we substitute a symbol (|) for each 2W and now need just 23 characters to describe the graphic, like so:

```
|5B|
|2B5W
|4B3W
|2B5W
|5B|
```

We've compressed the data by almost half but added overhead: we must now supply a dictionary defining |=2W.

Going a step further, we swap in symbols for 5B (\), 2B (/) and 5W (~), like so:

```
|\|
|/~
|4B3W
|/~
|\|
```

It takes just 17 characters to serve as a compressed version of the original 45 characters by adding three more symbols to our growing dictionary.

As we apply this run-length encoding to more and more data, we see improved compression ratios because we can apply symbols already in use and don't need to keep adding new symbols to our dictionary for each swap.

ZIP employs a lossless compression algorithm called DEFLATE, which came into use in 1993. DEFLATE underpins both ZIP archives and the PNG image format; and thanks to its efficiency and being free to use without license fees, DEFLATE remains the most widely used compression algorithm in the world.

DEFLATE combines two compression techniques, Huffman Coding and a dictionary technique called Lempel–Ziv–Storer–Szymanski (LZSS). Huffman Coding analyzes data to determine frequency of occurrence, generating a probability tree supporting assignment of the shortest symbols to the most recurrences. LZSS implements pointers to prior occurrences (back-references). Accordingly, DEFLATE achieves compression by the assignment of symbols based on frequency of use and by the matching and replacement of duplicate strings with pointers.

Tools for processing files in e-discovery must identify compressed files and apply the correct algorithm to unpack the contents. The decompression algorithm locates the tree and symbols library and retrieves and parses the directory structure and stored metadata for the contents.

Identification on Ingestion

Remember Programmer Ann and her struggle to select a TaggedyAnn file extension and signature? Now, those decisions play a vital role in how an e-discovery processing tool extracts text and metadata. If we hope to pull intelligible data from a file or container, we must first reliably ascertain the file's structure and encoding. For compressed files, we must apply the proper decompression algorithm. If it's an e-mail container format, we must apply the proper encoding schema to segregate messages and decode all manner of attachments. We must treat image files as images, sound files as sounds and so on. Misidentification of file types guarantees failure.

The e-discovery industry relies upon various open source and commercial file identifier tools. These tend to look first to the file's binary header for file identification and then to the file's extension and name. If the file type cannot be determined from the signature and metadata, the identification tool may either flag the file as unknown (an "exception") or pursue other identification methods as varied as byte frequency analysis (BFA) or the use of specialty filetype detection tools designed to suss out characteristics unique to certain file types, especially container files. Identifiers will typically report both the apparent file type (from metadata, *i.e.*, the file's name and extension) and the actual file type. Inconsistencies between these may prompt special handling or signal spoofing with malicious intent.

The table below sets out header signatures aka "magic numbers" for common file types:

File Type	Extension	Hex Signature	ASCII	Notes
ZIP Archive	ZIP	50 4B 03 04	PK..	
MS Office	DOCX XLSX PPTX	50 4B 03 04	PK..	Compressed XML files
Outlook mail	PST	21 42 44 4E 42	!BDN	
Outlook message	MSG	D0 CF 11 E0 A1 B1 1A E1		
Executable file	EXE	4D 5A	MZ	For Mark Zbikowski, who said, "when you're writing the linker, you get to make the rules."

File Type	Extension	Hex Signature	ASCII	Notes
Adobe PDF	PDF	25 50 44 46	%PDF	
PNG Graphic	PNG	89 50 4E 47	.PNG	
VMWare Disk file	VMDK	4B 44 4D 56	KDMV	
WAV audio file	WAV	52 49 46 46	RIFF	
Plain Text file	TXT	none	none	Only binary files have signatures

Media (MIME) Type Detection

File extensions are largely unique to Microsoft operating systems. Systems like Linux and Mac OS X don't rely on file extensions to identify file types. Instead, they employ a file identification mechanism called **Media (MIME) Type Detection**. MIME, which stands for **M**ultipurpose **I**nternet **M**ail **E**xtensions, is a seminal Internet standard that enables the grafting of text enhancements, foreign language character sets (Unicode) and multimedia content (*e.g.,* photos, video, sounds and machine code) onto plain text e-mails. Virtually all e-mail travels in MIME format.

The ability to transmit multiple file types via e-mail created a need to identify the content type transmitted. The **I**nternet **A**ssigned **N**umbers **A**uthority (IANA) oversees global Internet addressing and defines the hierarchy of media type designation. These hierarchical designations for e-mail attachments conforming to MIME encoding came to be known as **MIME types**. Though the use of MIME types started with e-mail, operating systems, tools and web browsers now employ MIME types to identify media, prompting the IANA to change the official name from MIME Types to **Media Types**.

Media types serve two important roles in e-discovery processing. First, they facilitate the identification of content based upon a media type declaration found in, *e.g.,* e-mail attachments and Internet publications. Second, media types serve as standard classifiers for files after identification of content. Classification of files within a media type taxonomy simplifies culling and filtering data in ways useful to e-discovery. While it's enough to specify "document," "spreadsheet" or "picture" in a Request for Production, e-discovery tools require a more granular breakdown of content. Tools must be able to distinguish a Word binary .DOC format from a Word XML .DOCX format, a Windows PowerPoint file from a Mac Keynote file and a GIF from a TIFF.

Media Type Tree Structure

Media types follow a path-like tree structure under one of the following standard types: **application, audio, image, text** and **video** (collectively called **discrete** media types) and **message** and **multipart** (called **composite** media types). These top-level media types are further defined by subtype and, optionally, by a suffix and parameter(s), written in lowercase.

Examples of file type declarations for common file formats:
Note: File types prefixed by x- are not IANA. Those prefixed by vnd. are vendor-specific formats.

Application

Word .DOC:	application/msword
Word .DOCX:	application/vnd.openxmlformats-officedocument.wordprocessingml.document
Adobe PDF:	application/pdf (.pdf)
PowerPoint .PPT:	application/vnd.ms-powerpoint
PowerPoint .PPTX:	application/vnd.openxmlformats-officedocument. presentationml. presentation
Slack file:	application/x-atomist-slack-file+json
.TAR archive:	application/x-tar
Excel .XLS:	application/vnd.ms-excel
Excel .XLSX:	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
ZIP archive .ZIP:	application/zip

Audio

.MID:	audio/x-midi
.MP3:	audio/mpeg
.MP4:	audio/mp4
.WAV:	audio/x-wav

Image

.BMP:	image/bmp
.GIF:	image/gif
.JPG:	image/jpeg
.PNG:	image/png
.TIF:	image/tiff

Text (Typically accompanied by a **charset** parameter identifying the character encoding)

.CSS:	text/css
.CSV:	text/csv
.HTML:	text/html
.ICS:	text/calendar

.RTF:	text/richtext
.TXT	text/plain

Video

.AVI	video/x-msvideo
.MOV:	video/quicktime
.MP4:	video/mp4
.MPG:	video/mpeg

When All Else Fails: Octet Streams and Text Stripping

When a processing tool cannot identify a file, it may flag the file as an exception and discontinue processing contents; but, the greater likelihood is that the tool will treat the unidentifiable file as an **octet stream** and harvest or "strip out" whatever text or metadata it *can* identify within the stream. An octet stream is simply an arbitrary sequence or "stream" of data presumed to be binary data stored as eight-bit bytes or "octets." So, an octet stream is anything the processor fails to recognize as a known file type.

In e-discovery, the risk of treating a file as an octet stream and stripping identifiable text is that the file's content is likely encoded, such that whatever plain text is stripped and indexed doesn't fairly mirror relevant content. However, because *some* text was stripped, the file may not be flagged as an exception requiring special handling; instead, the processing tool records the file as successfully processed notwithstanding the missing content.

Data Extraction and Document Filters

If ESI were like paper, you could open each item in its associated program (its *native application*), review the contents and decide whether the item is relevant or privileged. But, ESI is much different than paper documents in crucial ways:

- ESI collections tend to be exponentially more voluminous than paper collections;
- ESI is stored digitally, rendering it unintelligible absent electronic processing;
- ESI carries metainformation that is always of practical use and may be probative evidence;
- ESI is electronically searchable while paper documents require laborious human scrutiny;
- ESI is readily culled, filtered and deduplicated, and inexpensively stored and transmitted;
- ESI and associated metadata change when opened in native applications;

These and other differences make it impractical and risky to approach e-discovery via the piecemeal use of native applications as viewers. Search would be inconsistent and slow, and deduplication impossible. Too, you'd surrender all the benefits of mass tagging, filtering and production. Lawyers who learn that *native productions* are superior to other forms of production may mistakenly conclude that native production suggests use of native applications for review. Absolutely not! Native applications are not suited to e-discovery, and you shouldn't use them for review. E-discovery review tools are the only way to go.

To secure the greatest benefit of ESI in search, culling and review, we process ingested files to extract their text, embedded objects, and metadata. In turn, we normalize and tokenize extracted contents, add them to a database and index them for efficient search. These processing operations promote efficiency but impose penalties in terms of reduced precision and accuracy. It's a tradeoff demanding an informed and purposeful balancing of benefits and costs.

Returning to Programmer Ann and her efforts to fashion a new file format, Ann had a free hand in establishing the structural layout of her TaggedyAnn data files because she was also writing the software to read them. The ability to edit data easily is a hallmark of computing; so, programmers design files to be able to grow and shrink without impairing updating and retrieval of their contents. Files hold text, rich media (like graphics and video), formatting information, configuration instructions, metadata and more. All that disparate content exists as a sequence of hexadecimal characters. Some of it may reside at fixed offset addresses measured in a static number of bytes from the start of the file. But because files must be able to grow and shrink, fixed offset addressing alone won't cut it. Instead, files must supply dynamic directories of their contents or incorporate tags that serve as signposts for navigation.

When navigating files to extract contents, it's not enough to know where the data starts and ends, you must also know how the data's encoded. Is it ASCII text? Unicode? JPEG? Is it a date and time value or perhaps a bit flag where the numeric value serves to signal a characteristic or configuration to the program?

There are two broad approaches used by processing tools to extract content from files. One is to use the Application Programming Interface (API) of the application that created the file. The other is to turn to a published file specification or reverse engineer the file to determine where the data sought to be extracted resides and how it's encoded.

A software API allows "client" applications (*i.e.,* other software) to make requests or "calls" to the API "server" to obtain specific information and to ask the server to perform specific functions. Much like a restaurant, the client can "order" from a menu of supported API offerings without knowing what goes on in the kitchen, where the client generally isn't welcome to enter. Like a restaurant with off-menu items, the API may support undocumented calls intended for a limited pool of users.

For online data reposing in sites like Office 365, Dropbox or OneDrive, there's little choice but to use an API to get to the data; but for data in local files, using a native application's API is something of a last resort because APIs tend to be slow and constraining. Not all applications offer open APIs, and those that do won't necessarily hand off all data needed for e-discovery. For many years, a leading e-discovery processing tool required purchasers to obtain a "bootleg" copy of the IBM/Lotus Notes mail program because the secure structure of Notes files frustrated efforts to extract messages and attachments by any means but the native API.

An alternative to the native application API is the use of data extraction templates called **Document Filters.** Document filters lay out where content is stored within each filetype and how that content is encoded and interpreted. Think of them as an extraction template. Document filters can be based on a published file specification or they can be painstakingly reverse engineered from examples of the data—a terrifically complex process that produces outcomes of questionable accuracy and consistency. Because document filters are challenging to construct and keep up to date for each of the hundreds of file types seen in e-discovery, few e-discovery processors build their own library of document filters. Instead, they turn to a handful of commercial and open source filters.

The leading commercial collection of document filters is Oracle's *Outside In*, which its publisher describes as "a suite of software development kits (SDKs) that provides developers with a comprehensive solution to extract, normalize, scrub, convert and view the contents of 600 unstructured file formats." *Outside In* quietly serves as the extraction and viewer engine behind many e-discovery review tools, a fact the sellers of those tools are often reluctant to concede; but, I suppose sellers of the Lexus ES aren't keen to note it shares its engine, chassis and most parts with the cheaper Toyota Avalon.

Aspose Pty. Ltd., an Australian concern, licenses libraries of commercial APIs, enabling software developers to read and write to, *e.g.*, Word documents, Excel spreadsheets, PowerPoint presentations, PDF files and multiple e-mail container formats. Aspose tools can both read from and write to the various formats, the latter considerably more challenging.

Hyland Software's *Document Filters* is another developer's toolkit that facilitates file identification and content extraction for 500+ file formats, as well as support for OCR, redaction and image rendering. Per Hyland's website, its extraction tools power e-discovery products from Catalyst and Reveal Software.

A fourth commercial product that lies at the heart of several e-discovery and computer forensic tools (*e.g.*, Relativity, LAW, Ringtail aka Nuix Discover and Access Data's FTK) is *dtSearch*, which serves as both content extractor and indexing engine.

On the open source side, Apache's *Tika* is a free toolkit for extracting text and metadata from over a thousand file types, including most encountered in e-discovery. *Tika* was a subproject of the open source Apache Lucene project, Lucene being an indexing and search tool at the core of several commercial e-discovery tools.

Beyond these five toolsets, the wellspring of document filters and text extractors starts to dry up, which means a broad swath of commercial e-discovery tools relies upon a tiny complement of text and metadata extraction tools to build their indices and front-end their advanced analytics.

In fact, most e-discovery tools I've seen in the last 15 years are proprietary wrappers around code borrowed or licensed from common sources for file identifiers, text extractors, OCR, normalizers, indexers, viewers, image generators and databases. Bolting these off-the-shelf parts together to deliver an efficient workflow and user-friendly interface is no mean feat.

But as we admire the winsome wrappers, we must remember that these products share the same DNA in spite of marketing efforts suggesting "secret sauces" and differentiation. More to the point, products built on the same text and metadata extractor share the same limitations and vulnerabilities as that extractor.

By way of example, in 2018, Cisco's Talos Intelligence Group reported a serious security vulnerability in Hyland Software's *Document Filters*. Presumably, this vulnerability impacted implementations by e-discovery platforms Catalyst and Reveal Software. This isn't an isolated example, as security vulnerabilities have been reported in the text extraction products from, *inter alia*, Apache *Tika*, Aspose, and Oracle *Outside In*.

Security vulnerabilities are common across the software industry, and these reports are no more cause for alarm than any other circumstance where highly confidential and privileged client data may be compromised. But they underscore that the integrity of e-discovery products isn't simply a function of the skill of the e-discovery toolmaker and service provider. Developers build e-discovery software upon a handful of processing and indexing products that, like the ingredients in the food we eat and the origins of the products we purchase, ought to be revealed, not concealed. Why? Because all these components introduce variables contributing to different, and sometimes untoward, outcomes affecting the evidence.

Recursion and Embedded Object Extraction

Just as an essential task in processing is to correctly identify content and apply the right decoding schema, a processing tool must extract and account for all the components of a file that carry potentially responsive information.

Modern productivity files like Microsoft Office documents are rich, layered containers called **Compound Files**. Objects like images and the contents of other file formats may be embedded and linked within a compound file. Think of an Excel spreadsheet appearing as a diagram within a Word document. Microsoft promulgated a mechanism supporting this functionality called **OLE** (pronounced "o-lay" and short for **O**bject **L**inking and **E**mbedding). OLE supports dragging and dropping content between applications and the dynamic updating of embedded content, so the Excel spreadsheet embedded in a Word documents updates to reflect changes in the source spreadsheet file. A processing tool must be able to recognize an OLE object and extract and enumerate all of the embedded and linked content.

A MIME e-mail is also a compound document to the extent it transmits multipart content, particularly encoded attachments. A processing tool must account for and extract every item in the e-mail's informational payload, recognizing that such content may nest like a Russian doll. An e-mail attachment could be a ZIP container holding multiple Outlook .PST mail containers holding e-mail collections that, in turn, hold attachments of OLE documents and other ZIP containers! The mechanism by which a processing tool explores, identifies, unpacks and extracts all embedded content from a file is called **recursion**. It's crucial that a data extraction tool be able to recurse through a file and loop itself to extract embedded content until there is nothing else to be found.

Tika, the open source extraction toolset, classifies file structures in the following ways for metadata extraction:

- Simple
- Structured
- Compound
- Simple Container
- Container with Text

Simple Document

A simple document is a single document contained in a single file. Some examples of simple documents include text files and xml files. Any metadata associated with a simple document is for the entire document.

Structured Document

Like simple documents, structured documents are single documents in single files. What makes a structured document different is that the document has internal structure, and there is metadata associated with specific parts of a document. For example, a PDF document has an internal structure for representing each page of the PDF, and there may be metadata associated with individual pages.

Compound Document

A compound document is a single document made up of many separate files, usually stored inside of a single container. Examples of compound documents include the following:

- .doc (several named streams inside of an OLE file)
- .xlsx (several named xml files inside of a ZIP file)

Simple Container

A simple container is a container file that contains other files. The container itself does not contain text but instead contains files that could be any document type. Examples of a simple container include ZIP, tar, tar.gz, and tar.bz2.

Container with Text

Some container files have text of their own and also contain other files. Examples include the following:

- an e-mail with attachments
- A .doc with embedded spreadsheets

Family Tracking and Unitization: Keeping Up with the Parts

As a processing tool unpacks the embedded components of compound and container files, it must update the database with information about what data came from what file, a relationship called **unitization**. In the context of e-mail, recording the relationship between a transmitting message and its attachments is called **family tracking:** the transmitting message is the **parent object** and the attachments are **child objects**. The processing tool must identify and preserve metadata values applicable to the entire contents of the compound or container file (like system metadata for the parent object) and embedded metadata applicable to each child object. One of the most important metadata values to preserve and pair with each object is the object's custodian or source. Post-processing, every item in an e-discovery collection must be capable of being tied back to an originating file at time of ingestion, including its prior unitization and any parent-child relationship to other objects.

Exceptions Reporting: Keeping Track of Failures

It's rare that a sizable collection of data will process flawlessly. There will almost always be encrypted files that cannot be read, corrupt files, files in unrecognized formats or languages and files requiring optical character recognition (OCR) to extract text. A great many documents are not amenable to text search without special handling. Common examples of non-searchable documents are faxes and scans, as well as TIFF images and Adobe PDF documents lacking a text layer. A processing tool must track all exceptions and be capable of generating an **exceptions report** to enable counsel and others with oversight responsibility to act to rectify exceptions by, *e.g.*, securing passwords, repairing or replacing corrupt files and running OCR against the files. Exceptions resolution is key to a defensible e-discovery process.

Counsel and others processing ESI in discovery should broadly understand the exceptions handling characteristics of their processing tools and be competent to make necessary disclosures and answer questions about exceptions reporting and resolution. *Exceptions are* *exclusions—the evidence is missing; so, exceptions should be disclosed and must be defended.* As noted earlier, it's particularly perilous when a processing tool defaults to text stripping an unrecognized or misrecognized file because the tool may fail to flag a text-stripped file as an exception requiring resolution. Just because a tool succeeds in stripping some text from a file doesn't mean that all discoverable content was extracted.

Lexical Preprocessing of Extracted Text

Computers are excruciatingly literal. Computers cannot read. Computers cannot understand language in the way humans do. Instead, computers apply rules assigned by programmers to normalize, tokenize, and segment natural language, all instances of **lexical preprocessing**—steps to prepare text for parsing by other tools.

Normalization

ESI is numbers; numbers are precise. Variations in those numbers—however subtle to humans—hinder a computer's ability to equate information as humans do. Before a machine can distinguish words or build an index, we must massage the streams of text spit out by the document filters to ultimately increase recall; that is, to insure that more documents are retrieved by search, even the documents we seek that don't exactly match our queries.

Variations in characters that human beings readily overlook pose big challenges to machines. So, we seek to minimize the impact of these variations through **normalization**. How we normalize data and even the order in which steps occur affect our ability to query the data and return correct results.

Character Normalization

Consider three characteristics of characters that demand normalization: **Unicode equivalency**, **diacriticals** (accents) and **case** (capitalization).

Unicode Normalization

In our discussion of ASCII encoding, we established that each ASCII character has an assigned, corresponding numeric value (*e.g.,* a capital "E" is 0100 0101 in binary, 69 in decimal and Ox45 in hexadecimal). But linguistically identical characters encoded in Unicode may be represented by *different* numeric values by virtue of accented letters having both precomposed and composite references. That means that you can use an encoding specific to the accented letter (a **precomposed** character) or you can fashion the character as a **composite** by pairing the encoding for the base letter with the encoding for the diacritical. For example, the Latin capital "E" with an acute accent (É) may be encoded as either **U+00C9** (a precomposed Latin capital letter E with acute accent) or as **U+0045** (Latin capital letter E) + **U+0301** (combining acute accent). Both will appear as É.

Easy enough to grasp, but there are a lot of accented variants of Latin letters, *e.g.*:

Ĕ Ḝ Ê Ê Ē̂ Ê̌ Ề Ế Ể Ễ Ệ Ẻ Ḙ Ě Ɇ Ė É Ē̃ Ẹ Ë È E̩ Ȅ É É̩ Ē Ḕ Ḗ Ẽ Ḛ Ę Ę́ Ę̃ Ȩ E̩ ę[

Surely, we don’t want to have to account for *every* variation in which a character can be encoded in Unicode when searching in e-discovery! To obviate that burden, the Unicode Consortium promulgates normalization algorithms that produce a consistent (“normalized”) encoding for each identical character. One version of the algorithm reduces all identical characters to a composed version and another reduces all to a decomposed (composite) version. In e-discovery, we often seek to strip accents so see more of the latter.

Diacritical Normalization

Unicode normalization serves to ensure that the same canonical character is encoded in a consistent way. But often—especially in the United States—we want accented characters to be searchable whether a diacritical is employed or not. This requires normalizing the data to forge a false equivalency between accented characters and their non-accented ASCII counterparts. So, if you search for “resume” or “cafe,” you will pick up instances of “resumé” and “café.” As well, we must normalize ligatures like the German Eszett (ß) seen in the word “straße,” or “street.”

The major processing tools offer filters that convert alphabetic, numeric, and symbolic Unicode characters which are not in the first 127 ASCII characters (the "Basic Latin" Unicode block) into their reasonable ASCII equivalents, if any. Lucene and Elasticsearch offer “ASCIIFolding” filters for this. *dtSearch* strips diacritics by default when it creates an index, but indices can optionally be made accent-sensitive by re-indexing.

Case Normalization

The Latin alphabet is bicameral, meaning it employs upper- and lower-case letters to enhance meaning. *[The terms upper- and lower-case derive from the customary juxtaposition of the shallow drawers or “cases” that held movable type for printing presses.]* By contrast, languages such as Chinese, Arabic and Hebrew are unicameral and use no capitalization. Because people capitalize indiscriminately—particularly in e-mail and messaging—most often we want search queries to be case-insensitive such that DOE, Doe and doe all return the same hits. Other times, the ability to limit a search to a case-specific query is advantageous, such as by searching just DOE when your interest is the Department of Energy and search precision is more important than recall.

Just as processing tools can be configured to "fold" Unicode characters to ASCII equivalents, they can fold all letters to their upper- or lower-case counterparts, rendering an index that is case-insensitive. Customarily, normalization of case will require specification of a default language because of different capitalization conventions attendant to diverse cultures.

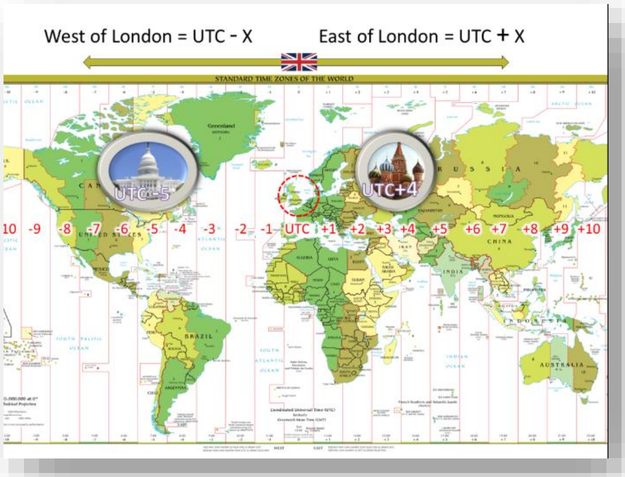
Impact of Normalization on Discovery Outcomes

Although all processing tools draw on a handful of filters and algorithms for these and other normalization processes, processors don't implement normalization in the same sequence or with identical default settings. Accordingly, it's routine to see tools produce varying outcomes in culling and search because of differences in character normalization. Whether these differences are material or not depends upon the nature of the data and the inquiry; but any service provider and case manager should know how their tool of choice normalizes data.

In the sweep of a multi-million document project, the impact of normalization might seem trivial. Yet character normalization affects the whole collection and plays an outsize role in what's filtered and found. It's an apt reminder that a good working knowledge of processing equips e-discovery professionals to "normalize" *expectations,* especially expectations as to what data will be seen and searchable going forward. The most advanced techniques in analytics and artificial intelligence are no better than what emerges from processing. If the processing is off, it's fancy joinery applied to rotten wood.

Lawyers must fight for quality before review. Sure, review is the part of e-discovery most lawyers see and understand, so the part many fixate on. As well, review is the costliest component of e-discovery and the one with cool tools. But here's the bottom line: *The most sophisticated MRI scanner won't save those who don't survive the trip to the hospital.* It's more important to have triage that gets people to the hospital alive than the best-equipped emergency room. Collection and processing are the EMTs of e-discovery. If we don't pay close attention to quality, completeness and process *before* review, review won't save us.

Time Zone Normalization


West of London = UTC - X
East of London = UTC + X
STANDARD TIME ZONES OF THE WORLD
UTC-5
UTC+4
10 -9 -8 -7 -6 -5 -4 -3 -2 -1 UTC +1 +2 +3 +4 +5 +6 +7 +8 +9 +10

You needn't be an Einstein of e-discovery to appreciate that time is relative. Parsing a message thread, it's common to see e-mails from Europe to the U.S. prompt replies that, at least according to embedded metadata, appear to precede by hours the messages they answer. Time zones and daylight savings time both work to make it difficult to correctly order documents and communications on a consistent timeline. So, a common processing task is to normalize date and time values according to a single temporal baseline, often **Coordinated Universal Time** (UTC)—essentially Greenwich Mean Time—or to any other time zone the parties choose. The differential between the source time and **UTC offset** may then be expressed as plus or minus the numbers of hours separating the two (*e.g.,* UTC-0500 to denote five hours earlier than UTC).

Parsing and Tokenization

To this point, we've focused on efforts to identify a file's format, then extract its content and metadata—important tasks, because if you don't get the file's content out and properly decoded, you've nearly nothing to work with. But, getting data out is just the first step. Now, we must distill the extracted content into the linguistic components that serve to convey the file's informational payload; that is, we need to isolate the words within the documents and construct an index of those words to allow us to instantly identify or exclude files based on their lexical content.

There's a saying that anything a human being can do after age five is easy for a computer, but mastering skills humans acquire earlier is hard. Calculate pi to 31 trillion digits? Done! Read a Dr. Seuss book? Sorry, no can do.

Humans are good at spotting linguistic units like words and sentences from an early age, but computers must identify lexical units or "**tokens**" within extracted and normalized character strings, a process called "**tokenization**." When machines search collections of documents and data for keywords, they don't search the extracted text of the documents or data for matches; instead, they consult an index of words built from extracted text. Machines cannot read; instead, computers identify "words" in documents because their appearance and juxtaposition meet certain **tokenization rules.** These rules aren't uniform across systems or software. Many indices simply don't index short words (*e.g.,* two-letter words, acronyms and initializations). None index single letters or numbers.

Tokenization rules also govern such things as the handling of punctuated terms (as in a compound word like "wind-driven"), capitalization/case (will a search for "roof" also find "Roof?"), diacritical marks (will a search for "Rene" also find "René?") and numbers and single letters (will a search for "Clause 4.3" work? What about a search for "Plan B?"). Most people simply *assume* these searches will work. Yet, in many e-discovery search tools, they don't work as expected or don't work at all.

So, how do you train a computer to spot sentences and words? What makes a word a word and a sentence a sentence?

Languages based on Latin-, Cyrillic-, or Greek-based writing systems, such as English and European languages, are "segmented;" that is, they tend to set off ("delimit") words by white space and punctuation. Consequently, most tokenizers for segmented languages base token boundaries on spacing and punctuation. That seems a simple solution at first blush, but one quickly complicated by hyphenated words, contractions, dates, phone numbers and abbreviations. How does the machine distinguish a word-break hyphen from a true or lexical hyphen? How does the machine distinguish the periods in the salutation "Mr." or the initialization "G.D.P.R" from periods which signal the ends of sentences? In the realm of medicine and pharmacology, many words contain numbers, dashes and parentheses as integral parts. How could you defend a search for Ibuprofen if you failed to also seek instances of ***(RS)-2-(4-(2-methylpropyl)phenyl)propanoic acid***?

Again, tokenization rules aren't uniform across systems, software or languages. Some tools are simply incapable of indexing and searching certain characters. These exclusions impact discovery in material ways. Several years ago, after contentious motion practice, a court ordered the parties to search a dataset using queries incorporating the term "20%." No one was pleased to learn their e-discovery tools were incapable of searching for the percentage sign.

You cannot run a query in Relativity including the percentage sign (%) because Relativity uses *dtSearch* as an indexing tool and *dtSearch* has reserved the character "%" for another purpose. This is true no matter how you tweak the settings because the % sign simply cannot be added to the index and made searchable.[8] When you run a search, you won't be warned that the search is impossible; you'll simply get no hits on any query requiring the % sign be found.

[8] There's a clumsy workaround to this described in the Help section of Relativity's website: https://help.relativity.com/RelativityOne/Content/Recipes/Searching__Filtering__and_Sorting/Searching_for_symbols.htm; however, it doesn't serve to make a percentage value searchable so much as permit a user to find a

Using *dtSearch*/Relativity as another example, you can specify the way to process hyphens at the time an index is created, but you cannot change how hyphens are handled without re-indexing the collection. The default setting is to treat hyphens as spaces, but there are four alternative treatments.

From the *dtSearch* Help pages:

> The dtSearch Engine supports four options for the treatment of hyphens when indexing documents: spaces, searchable text, ignored, and "all three."
>
> *For most applications, treating hyphens as spaces is the best option.* Hyphens are translated to spaces during indexing and during searches. For example, if you index "first-class mail" and search for "first class mail", "first-class-mail", or "first-class mail", you will find the phrase correctly.
>
> **Values**

HyphenSettings Value	**Meaning**
dtsoHyphenAsIgnore	index "first-class" as "firstclass"
dtsoHyphenAsHyphen	index "first-class" as "first-class"
dtsoHyphenAsSpace	index "first-class" as "first" and "class"
dtsoHyphenAll	index "first-class" all three ways

…

The "all three" option has one advantage over treating hyphens as spaces: it will return a document containing "first-class" in a search for "firstclass". Otherwise, it provides no benefit over treating hyphens as spaces, and it has some significant disadvantages:

1. The "all three" option generates many extra words during indexing. For each pair of words separated by a hyphen, six words are generated in the index.
2. If hyphens are treated as significant at search time, it can produce unexpected results in searches involving longer phrases or words with multiple hyphens.

By default, *dtSearch* and Relativity treat all the following characters as spaces:

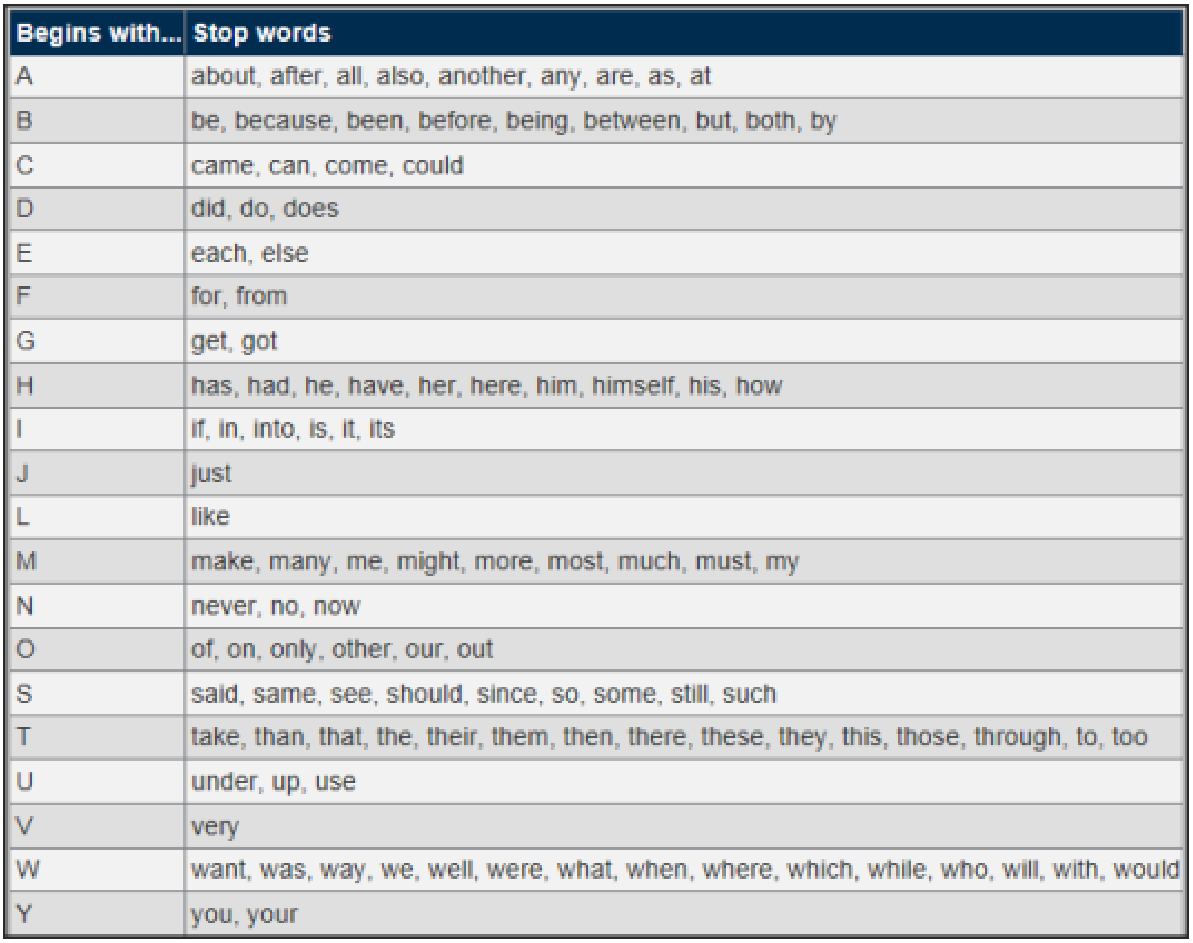
!"#$&'()*+,./:;<=>?@[\5c]^`{|}~

numeric value followed by any symbol; that is, you'd hit on 75%, but also 75! and 75#. Even this kludge requires rebuilding all indices.

Although several of the characters above can be made searchable by altering the default setting and reindexing the collection, the following characters CANNOT be made searchable in *dtSearch* and Relativity: ( ) * ? % @ ~ & : =

Stop Words

Some common "stop words" or "noise words" are excluded from an index when it's compiled. E-discovery tools typically exclude *dozens or hundreds* of stop words from indices. The table below lists 123 English stop words excluded by default in *dtSearch* and Relativity:



Begins with...	Stop words
A	about, after, all, also, another, any, are, as, at
B	be, because, been, before, being, between, but, both, by
C	came, can, come, could
D	did, do, does
E	each, else
F	for, from
G	get, got
H	has, had, he, have, her, here, him, himself, his, how
I	if, in, into, is, it, its
J	just
L	like
M	make, many, me, might, more, most, much, must, my
N	never, no, now
O	of, on, only, other, our, out
S	said, same, see, should, since, so, some, still, such
T	take, than, that, the, their, them, then, there, these, they, this, those, through, to, too
U	under, up, use
V	very
W	want, was, way, we, well, were, what, when, where, which, while, who, will, with, would
Y	you, your

Source: Relativity website, November 3, 2019

Relativity won't index punctuation marks, single letters or numbers. Nuix Discovery (formerly Ringtail) uses a similar English stop word list, except Nuix indexes the words "between," "does," "else," "from," "his," "make," "no," "so," "to," "use" and "want" and "does" where Relativity won't. Relativity indexes the words "an," "even," "further," "furthermore," "hi," "however," "indeed," "made," "moreover," "not" "or," "over," "she" and "thus" where Nuix won't. ***Does it make sense that both tools exclude "he" and "her," and both include "hers," but only Relativity excludes "his?"***

Tools built on the open source Natural Language Tool Kit won't index 179 English stop words. In other products, I've seen between 500 and 700 English stop words excluded. In one notorious instance, the two words that made up the company's own name were both stop words in their e-discovery system. They literally could not find their own name (or other stop words in queries they'd agreed to run)!

Remember, ***if it's not indexed, it's not searched***. Putting a query in quotes won't make any difference. No warning messages appear when you run a query including stop words, so it's the obligation of those running searches to acknowledge the incapability of the search. "No hits" is not the same thing as "no documents." If a party or counsel knows that the systems or searches used in e-discovery will fail to perform as expected, they should affirmatively disclose such shortcomings. If a party or counsel is uncertain whether systems or searches work as expected, they should find out by*, e.g.,* running tests to be reasonably certain.

No system is perfect, and perfect isn't the e-discovery standard. Often, we must adapt to the limitations of systems or software. But we must know what a system can't do before we can find ways to work around its limitations or set expectations consistent with actual capabilities, not magical thinking and unfounded expectations.

Building a Database and Concordance Index

This primer is about processing, and the database (and viewer) belong to the realm of review tools and their features. However, a brief consideration of their interrelationship is useful.

The Database

At every step of processing, information derived from and about the items processed is continually handed off to a database. As the system ingests each file, a record of its name, size and system metadata values becomes part of the database. Sources—called "**custodians**" when they are individuals—are identified and comprise database fields. The processor calculates hash values and contributes them to the database. The tool identifies and extracts application metadata values from the processed information items, including, *inter alia*, authoring data for documents and subject, sender and recipient data for e-mail messages. The database also holds pointers to TIFF or PDF page images and to extracted text. The database is where items are **enumerated**, that is, assigned an item number that will uniquely identify each item in the processed collection. This is an identifier distinct from any Bates numbers subsequently assigned to items when produced.

The database lies at the heart of all e-discovery review tools. It's the recipient of much of the information derived from processing. But note, the database is <u>not</u> the index of text extracted from the processed items. The **concordance index**, the **database** and a third component, the **document viewer,** operate in so tightly coupled a manner that they seem like one.

A query of the index customarily triggers a return of information from the database about items "hit" by the query, and the contents of those items are, in turn, depicted in the viewer, often with hits highlighted. The perceived quality of commercial e-discovery review tools is a function of how seamlessly and efficiently these discrete functional components integrate to form a robust and intuitive user interface and experience.

Much like the file identification and content extraction tools discussed, e-discovery tool developers tend not to code databases from scratch but build atop a handful of open source or commercial database platforms. Notable examples are SQL Server and SQLite. Notwithstanding Herculean efforts of marketers to suggest differences, e-discovery tools tend to share the same or similar "database DNA." Users are none the wiser to the common foundations because the "back end" of discovery tools (the program's code and database operations layer) tends to be hidden from users and wrapped in an attractive interface.

The Concordance Index

Though there was a venerable commercial discovery tool called Concordance (now, Cloudnine), the small-c term "concordance" describes an alphabetical listing, particularly a mapping, of the important words in a text. Historically, scholars spent years painstakingly constructing concordances (or "full-text" indices) of Shakespeare's works or the Bible by hand. In e-discovery, software builds concordance indices to speed lexical search. While it's technically feasible to keyword search all documents in a collection, one after another (so-called "serial search"), it's terribly inefficient.[9]

Instead, the universal practice in e-discovery is to employ software to extract the text from information items, tokenize the contents to identify words and then construct a list of each token's associated document and location. Accordingly, *text searches in e-discovery don't search the evidence; they only search a concordance index of tokenized text*.

This is a crucial distinction because it means the quality of search in e-discovery is only as effective as the index is complete.

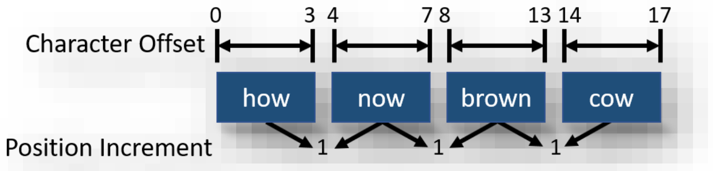
Indexing describes the process by which the data being is processed to form a highly efficient cross-reference lookup to facilitate rapid searching

Notable examples of concordance indexing tools are *dtSearch*, MarkLogic, Hyland Enterprise Search and Apache Lucene along with the Lucene-related products, Apache SOLR and Elasticsearch.

[9] Computer forensic examiners still use serial searches when the corpus is modest and when employing Global Regular Expressions (GREP searches) to identify patterns conforming to, *e.g.*, social security or credit card numbers.

Lucene is an open source, inverted, full-text index. It takes the tokenized and normalized word data from the processing engine and constructs an index for each token (word). It's termed an "inverted index" because it inverts a page-centric data structure (page>words) to a keyword-centric data structure (words>pages)

The full-text index consists of a token identifier (the word), a listing of documents containing the token and the position of the token within those documents (offset start- and end-positions, plus the increment between tokens). Querying the index returns a listing of tokens and positions for each. It may also supply an algorithmic ranking of the results. Multiword queries return a listing of documents where there's an intersection of the returns for each token searched and found.



An advantage of tools like Lucene is their ability to be updated incrementally (*i.e.*, new documents can be added without the need to recreate a single, large index).

Culling and Selecting the Dataset

Processing is not an end but a means by which potentially responsive information is exposed, enumerated, normalized and passed on for search, review and production. Although much culling and selection occurs in the search and review phase, the processing phase is an opportunity to reduce data volumes by culling and selecting by defensible criteria.

Now that the metadata is in a database and the collection has been made text searchable by creation of a concordance index, it's feasible to filter the collection by, *e.g.,* date ranges, file types, Internet domains, file size, custodian and other objective characteristics. We can also cull the dataset by **immaterial item suppression**, **de-NISTing** and **deduplication**, all discussed *infra*.

The crudest but most common culling method is keyword and query filtering; that is, lexical search. Lexical search and its shortcomings are beyond the scope of this processing primer, but it should be clear by now that the quality of the processing bears materially on the ability to find what we seek through lexical search. No search and review process can assess the content of items missed or malformed in processing.

Immaterial Item Suppression

E-discovery processing tools must be able to track back to the originating source file for any information extracted and indexed. Every item must be catalogued and enumerated, including each container file and all contents of each container. Still, in e-discovery, we've little need to search or produce the wrappers if we've properly expanded and extracted the contents. The wrappers are immaterial items.

Immaterial items are those extracted for forensic completeness but having little or no intrinsic value as discoverable evidence. Common examples of immaterial items include the folder structure in which files are stored and the various container files (like ZIP, RAR files and other containers, *e.g.,* mailbox files like Outlook PST and MBOX, and forensic disk image wrapper files like .E0x or .AFF) that tend to have no relevance apart from their contents.

Accordingly, it's handy to be able to suppress immaterial items once we extract and enumerate their contents. It's pointless to produce a ZIP container if its contents are produced, and it's perilous to do so if some contents are non-responsive or privileged.

De-NISTing

De-NISTing is a technique used in e-discovery and computer forensics to reduce the number of files requiring review by excluding standard components of the computer's operating system and off-the-shelf software applications like Word, Excel and other parts of Microsoft Office. Everyone has this digital detritus on their systems—things like Windows screen saver images, document templates, clip art, system sound files and so forth. It's the stuff that comes straight off the installation disks, and it's just noise to a document review.

Eliminating this noise is called "de-NISTing" because those noise files are identified by matching their cryptographic hash values (*i.e.,* digital fingerprints, explanation to follow) to a huge list of software hash values maintained and published by the National Software Reference Library, a branch of the National Institute for Standards and Technology (**NIST**). The NIST list is free to download, and pretty much everyone who processes data for e-discovery and computer forensic examination uses it.

The value of de-NISTing varies according to the makeup of the collection. It's very effective when ESI has been collected indiscriminately or by forensic imaging of entire hard drives (including operating system and executable files). De-NISTing is of limited value when the collection is composed primarily of user-created files and messages as distinguished from system files and executable applications. As a rule, the better focused the e-discovery collection effort (*i.e.,* the more targeted the collection), the smaller the volume of data culled via de-NISTing.

Cryptographic Hashing:

My students at the University of Texas School of Law and the Georgetown E-Discovery Training Academy spend considerable time learning that all ESI is just a bunch of numbers. They muddle through readings and exercises about Base2 (binary), Base10 (decimal), Base16 (hexadecimal) and Base64 and also about the difference between single-byte encoding schemes (like ASCIII) and double-byte encoding schemes (like Unicode). It may seem like a wonky walk in the weeds; but it's time well spent when the students snap to the crucial connection between numeric encoding and our ability to use math to cull, filter and cluster data. It's a necessary precursor to gaining Proustian "new eyes" for ESI.

Because ESI is just a bunch of numbers, we can use algorithms (mathematical formulas) to distill and compare those numbers. Every student of electronic discovery learns about cryptographic hash functions and their usefulness as tools to digitally fingerprint files in support of identification, authentication, exclusion and deduplication. When I teach law students about hashing, I tell them that hash functions are published, standard mathematical algorithms into which we input digital data of arbitrary size and the hash algorithm spits out a bit string (again, just a sequence of numbers) of fixed length called a "hash value." Hash values *almost* exclusively correspond to the digital data fed into the algorithm (termed "the message") such that the chance of two different messages sharing the same hash value (called a "hash collision") is exceptionally remote. But because it's *possible*, we can't say each hash value is truly "unique."

Using hash algorithms, any volume of data—from the tiniest file to the contents of entire hard drives and beyond—can be *almost* uniquely expressed as an alphanumeric sequence. In the case of the MD5 hash function, data is distilled to a value written as 32 hexadecimal characters (0-9 and A-F). It's hard to understand until you've figured out Base16; but, those 32 characters represent 340 *trillion, trillion, trillion* different possible values (2^{128} or 16^{32}).

Hash functions are one-way calculations, meaning you can't reverse ("invert") a hash value and ascertain the data corresponding to the hash value in the same way that you can't decode a human fingerprint to deduce an individual's eye color or IQ*. It identifies, but it doesn't reveal.* Another key feature of hashing is that, due to the so-called "avalanche effect" characteristic of a well-constructed cryptographic algorithm, when the data input changes even slightly, the hash value changes dramatically, meaning there's no discernable relationship between inputs and outputs. Similarity between hash values doesn't signal any similarity in the data hashed.

There are lots of different hash algorithms, and different hash algorithms generate different hash values for the same data. That is, the hash value for the phrase "Mary had a little lamb" will be the following in each of the following hash algorithms:

MD5:	e946adb45d4299def2071880d30136d4
SHA-1:	bac9388d0498fb378e528d35abd05792291af182
SHA-256:	efe473564cb63a7bf025dd691ef0ae0ac906c03ab408375b9094e326c2ad9a76

It's identical data, *but it prompts different hashes using different algorithms*. Conversely, identical data will generate identical hash values when using the *same* hash function. Freely published hash functions are available to all, so if two people (or machines) anywhere use the same hash function against data and generate matching hash values, their data is identical. If they get different hash values, they can be confident the data is different. The differences may be trivial in practical terms, but any difference suffices to produce markedly different hash values.

I don't dare go deeper in the classroom, but let's dig down a bit here and explore the operations behind calculating an MD5 hash value. If you really don't care, just skip ahead to deduplication.

A widely used hash function is the Message Digest 5 (MD5) hash algorithm circulated in 1992 by MIT professor Ron Rivest as Requests for Comments 1321. Requests for Comments or RFCs are a way the technology community circulates proposed standards and innovations generally relating to the Internet. MD5 has been compromised in terms of its immunity to hash collisions in that's it's feasible to generate different inputs that generate matching MD5 hashes; however, MD5's flaws minimally impact its use in e-discovery where it remains a practical and efficient way to identify, deduplicate and cull datasets.

When I earlier spoke of a hash algorithm generating a hash value of "fixed length," that fixed length for MD5 hashes is 128 bits (16 bytes) or 128 ones and zeroes in binary or Base2 notation. That's a vast number space. It's 340,282,366,920,938,463,463,374,607,431,768,211,455 possibilities in our familiar decimal or Base10 notation. It's also unwieldy, so we shorten MD5 hash values to a 32-character Base16 or hexadecimal ("hex") notation. It's the same numeric value conveniently expressed in a different base or "radix," so it requires only one-fourth as many characters to write the number in hex notation as in binary notation.

That 32-*character* MD5 hash value is built from four 32-*bit* calculated values that are concatenated, that is, positioned end to end to form a 128-bit sequence or "string." Since we can write a 32-bit number as eight hexadecimal characters, we can write a 128-bit number as four concatenated 8-character hex values forming a single 32-character hexadecimal hash. Each of those four 32-bit values are the product of 64 calculations (four rounds of 16 operations) performed on each 512-bit chunk of the data being hashed while applying various specified constants. After each round of calculations, the data shifts within the array in a practice called left bit rotation, and a new round of calculations begins. The entire hashing process starts by padding the message data to insure it neatly comprises 512-bit chunks and by initializing the four 32-bit variables to four default values.

For those wanting to see how this is done programmatically, here's the notated MD5 pseudocode (from Wikipedia):

```
//Note: All variables are unsigned 32 bit and wrap modulo 2^32 when calculating
var int[64] s, K
var int i
```

```
//s specifies the per-round shift amounts
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22 }
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20 }
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21 }

//Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63
    K[i] := floor($2^{32}$ × abs(sin(i + 1)))
end for
//(Or just use the following precomputed table):
K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
K[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
K[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
K[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
K[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
K[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
K[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
K[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
K[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
K[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }

//Initialize variables:
var int a0 := 0x67452301   //A
var int b0 := 0xefcdab89   //B
var int c0 := 0x98badcfe   //C
var int d0 := 0x10325476   //D

//Pre-processing: adding a single 1 bit
append "1" bit to message
// Notice: the input bytes are considered as bits strings,
//  where the first bit is the most significant bit of the byte.[50]
```

```
//Pre-processing: padding with zeros
append "0" bit until message length in bits ≡ 448 (mod 512)
append original length in bits mod $2^{64}$ to message

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of padded message
    break chunk into sixteen 32-bit words M[j], 0 ≤ j ≤ 15
    //Initialize hash value for this chunk:
    var int A := a0
    var int B := b0
    var int C := c0
    var int D := d0
    //Main loop:
    for i from 0 to 63
        var int F, g
        if 0 ≤ i ≤ 15 then
            F := (B and C) or ((not B) and D)
            g := i
        else if 16 ≤ i ≤ 31 then
            F := (D and B) or ((not D) and C)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
            F := B xor C xor D
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
            F := C xor (B or (not D))
            g := (7×i) mod 16
        //Be wary of the below definitions of a,b,c,d
        F := F + A + K[i] + M[g]
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
    end for
    //Add this chunk's hash to result so far:
    a0 := a0 + A
    b0 := b0 + B
    c0 := c0 + C
    d0 := d0 + D
end for

var char digest[16] := a0 append b0 append c0 append d0 //(Output is in little-endian)

//leftrotate function definition
leftrotate (x, c)
    return (x << c) binary or (x >> (32-c));
```

Despite the complexity of these calculations, it's possible to contrive hash collisions where different data generate matching MD5 hash values. Accordingly, the cybersecurity community have moved away from MD5 in applications requiring collision resistance, such as digital signatures.

You may wonder why MD5 remains in wide use if it's "broken" by engineered hash collisions. Why not simply turn to more secure algorithms like SHA-256? Some tools and vendors have done so, but a justification for MD5's survival is that the additional calculations required to make alternate hash functions more secure consume more time and computing resources. Too, most tasks in e-discovery built around hashing—*e.g.,* deduplication and De-NISTing—don't demand strict protection from engineered hash collisions. For e-discovery, MD5 "ain't broke," so there's little cause to fix it.

Deduplication

Processing information items to calculate hash values supports several capabilities, but probably none more useful than deduplication.

Near-Deduplication

A modern hard drive holds trillions of bytes, and even a single Outlook e-mail container file typically comprises billions of bytes. Accordingly, it's easier and faster to compare 32-character/16 byte "fingerprints" of voluminous data than to compare the data itself, particularly as the comparisons must be made repeatedly when information is collected and processed in e-discovery. In practice, each file ingested and item extracted is hashed, and its hash value is compared to the hash values of items previously ingested and extracted to determine if the file or item has been seen before. The first file is sometimes called the "pivot file," and subsequent files with matching hashes are suppressed as duplicates, and the instances of each duplicate and certain metadata is typically noted in a deduplication or "occurrence" log.

When the data is comprised of loose files and attachments, a hash algorithm tends to be applied to the full contents of the files. Notice that I said to "*contents*." Some data we associate with files is not actually stored inside the file but must be gathered from the file system of the device storing the data. Such "system metadata" is not contained within the file and, thus, is not included in the calculation when the file's content is hashed. A file's name is perhaps the best example of this. Recall that even slight differences in files cause them to generate different hash values. But, since a file's name is not typically housed within the file, you can change a file's name without altering its hash value.

So, the ability of hash algorithms to deduplicate depends upon whether the numeric values that serve as building blocks for the data differ from file to file. Keep that firmly in mind as we consider the many forms in which the informational payload of a document may manifest.

A Word .DOCX document is constructed of a mix of text and rich media encoded in Extensible Markup Language (XML), then compressed using the ubiquitous ZIP compression algorithm. It's a file designed to be read by Microsoft Word.

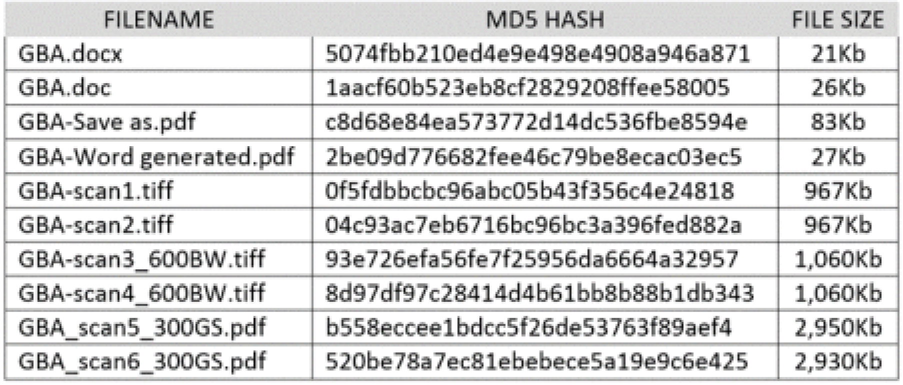
When you print the "same" Word document to an Adobe PDF format, it's reconstructed in a *page description language* specifically designed to work with Adobe Acrobat. It's structured, encoded and compressed in an entirely different way than the Word file and, as a different format, carries a different binary header signature, too.

When you take the printed version of the document and scan it to a Tagged Image File Format (TIFF), you've taken a picture of the document, now constructed in still another different format—one designed for TIFF viewer applications.

To the uninitiated, they are all the "same" document and might look pretty much the same printed to paper; but as ESI, their structures and encoding schemes are radically different. Moreover, even files generated in the same format may not be *digitally* identical when made at separate times. For example, no two optical scans of a document will produce identical hash values because there will always be some variation in the data acquired from scan to scan. Slight differences perhaps; but, any difference at all in content is going to frustrate the ability to generate matching hash values.

Opinions are cheap. Testing is truth. To illustrate this, I created a Word document of the text of Lincoln's Gettysburg Address. First, I saved it in the latest .DOCX Word format. Then, I saved a copy in the older .DOC format. Next, I saved the Word document to a .PDF format, using both the Save as PDF and Print to PDF methods. Finally, I printed and scanned the document to TIFF and PDF. Without shifting the document on the scanner, I scanned it several times at matching and differing resolutions.

I then hashed all the iterations of the “same” document. As the table below demonstrates, none of them matched hash-wise, not even the successive scans of the paper document:



FILENAME	MD5 HASH	FILE SIZE
GBA.docx	5074fbb210ed4e9e498e4908a946a871	21Kb
GBA.doc	1aacf60b523eb8cf2829208ffee58005	26Kb
GBA-Save as.pdf	c8d68e84ea573772d14dc536fbe8594e	83Kb
GBA-Word generated.pdf	2be09d776682fee46c79be8ecac03ec5	27Kb
GBA-scan1.tiff	0f5fdbbcbc96abc05b43f356c4e24818	967Kb
GBA-scan2.tiff	04c93ac7eb6716bc96bc3a396fed882a	967Kb
GBA-scan3_600BW.tiff	93e726efa56fe7f25956da6664a32957	1,060Kb
GBA-scan4_600BW.tiff	8d97df97c28414d4b61bb8b88b1db343	1,060Kb
GBA_scan5_300GS.pdf	b558eccee1bdcc5f26de53763f89aef4	2,950Kb
GBA_scan6_300GS.pdf	520be78a7ec81ebebece5a19e9c6e425	2,930Kb

Thus, file hash matching—the simplest and most defensible approach to deduplication—won’t serve to deduplicate the “same” document when it takes different forms or is made optically at separate times.

Now, here’s where it can get confusing. If you copied any of the electronic *files* listed above, the duplicate files would hash match the source originals and would handily deduplicate by hash. Consequently, multiple copies of the same electronic files will deduplicate, but that is because the files being compared have the same *digital* content. But we must be careful to distinguish the identicality seen in multiple iterations of the same file from the pronounced differences seen when we generate different electronic versions at different times from the same content. One notable exception seen in my testing was that successively saving the same Word document to a PDF format in the same manner sometimes generated identical PDF files. It didn’t occur consistently (*i.e.,* if enough time passed, changes in metadata in the source document triggered differences prompting the calculation of different hash values); but it happened, so is worth mentioning.

Consider hash matching in this real-life scenario: The source data were Outlook .PSTs from various custodians, each under 2GB in size. The form of production was single messages as .MSGs. Reportedly, the new review platform (actually a rather old concept search tool) was incapable of accepting an overlay load file that could simply tag the items already produced, so the messages already produced would have to be culled from the .PSTs before they were loaded. Screwy, to be sure; but we take our cases as they come, right?

It’s important to know that a somewhat obscure quirk of the .MSG message format is that when the same Outlook message is exported as an .MSG at various times, each exported message generates a different hash value because of embedded time of creation values. The differing hash values make it impossible to use hashes of .MSGs for deduplication without processing (*i.e.,* normalizing) the data to a format better suited to the task.

Here, a quick primer on deduplication of e-mail might be useful.

Mechanized deduplication of e-mail data can be grounded on three basic approaches:

1. Hashing the entire message as a file (*i.e.,* a defined block of data) containing the e-mail messages and comparing the resulting hash value for each individual message file. If they match, the files hold the same data. This tends not to work for e-mail messages exported as files because, when an e-mail message is stored as a file, messages that we regard as identical in common parlance (such as identical message bodies sent to multiple recipients) are not identical in terms of their byte content. The differences tend to reflect either variations in transmission seen in the message header data (the messages having traversed different paths to reach different recipients) or variations in time (the same message containing embedded time data when exported to single message storage formats as discussed above with respect to the .MSG format).
2. Hashing segments of the message using the same hash algorithm and comparing the hash values for each corresponding segment to determine relative identicality. With this approach, a hash value is calculated for the various parts of a message (*e.g.,* Subject, To, From, CC, Message Body, and Attachments) and these values are compared to the hash values calculated against corresponding parts of other messages to determine if they match. This method requires exclusion of those parts of a message that are certain to differ (such as portions of message headers containing server paths and unique message IDs) and normalization of segments, so that contents of those segments are presented to the hash algorithm in a consistent way.
3. Textual comparison of segments of the message to determine if certain segments of the message match to such an extent that the messages may be deemed sufficiently “identical” to allow them to be treated as the same for purposes of review and exclusion. This is much the same approach as (2) above, but without the use of hashing to compare the segments.

Arguably, a fourth approach entails a mix of these methods.

All these approaches can be frustrated by working from differing forms of the “same” data because, from the standpoint of the tools which compare the information, the forms are significantly different. Thus, if a message has been “printed” to a TIFF image, the bytes that make up the TIFF image bear no digital resemblance to the bytes comprising the corresponding e-mail message, any more than a photo of a rose smells or feels like the rose.

In short, changing forms of ESI changes data, and changing data changes hash values. Deduplication by hashing requires the same source data and the same, consistent application of algorithms. This is easy and inexpensive to accomplish, but it requires a compatible workflow to insure that evidence is not altered in processing so as to prevent the application of simple and inexpensive mechanized deduplication.

When parties cannot deduplicate e-mail, the reasons will likely be one or more of the following:

1. They are working from different forms of the ESI;
2. They are failing to consistently exclude inherently non-identical data (like message headers and IDs) from the hash calculation;
3. They are not properly normalizing the message data (such as by ordering all addresses alphabetically without aliases);
4. They are using different hash algorithms;
5. They are not preserving the hash values throughout the process; or
6. They are changing the data.

The excerpts that follow are from Microsoft and Relativity support publications, each describes the methodology employed to deduplicate e-mail messages:

Office 365 Deduplication

> Office 365 eDiscovery tools use a combination of the following e-mail properties to determine whether a message is a duplicate:
>
> - **InternetMessageId** - This property specifies the Internet message identifier of an e-mail message, which is a globally unique identifier that refers to a specific version of a specific message. This ID is generated by the sender's e-mail client program or host e-mail system that sends the message. If a person sends a message to more than one recipient, the Internet message ID will be the same for each instance of the message. Subsequent revisions to the original message will receive a different message identifier.
> - **ConversationTopic** - his property specifies the subject of the conversation thread of a message. The value of the **ConversationTopic** property is the string that describes the overall topic of the conversation. A conservation consists of an initial message and all messages sent in reply to the initial message. Messages within the same conversation have the same value for the **ConversationTopic** property. The value of this property is typically the Subject line from the initial message that spawned the conversation.

- **BodyTagInfo** - This is an internal Exchange store property. The value of this property is calculated by checking various attributes in the body of the message. This property is used to identify differences in the body of messages. https://docs.microsoft.com/en-us/microsoft-365/compliance/de-duplication-in-ediscovery-search-results

Relativity E-Mail Deduplication

The Relativity Processing Console (RPC) generates four different hashes for e-mails and keeps each hash value separate, which allows users to de-duplicate in the RPC based on individual hashes and not an all-inclusive hash string.

For example, if you're using the RPC, you have the ability to de-duplicate one custodian's files against those of another custodian based only on the body hash and not the attachment or recipient hashes.

Note that the following information is relevant to the RPC only and not to web processing in Relativity:

- **Body hash** - takes the text of the body of the e-mail and generates a hash.
- **Header hash** - takes the message time, subject, author's name and e-mail, and generates a hash.
- **Recipient hash** - takes the recipient's name and e-mails and generates a hash.
- **Attachment hash** - takes each SHA256 hash of each attachment and hashes the SHA256 hashes together.

MessageBodyHash

To calculate an e-mail's MessageBodyHash, the RPC:

1. Removes all carriage returns, line feeds, spaces, and tabs from the body of the e-mail to account for formatting variations. An example of this is when Outlook changes the formatting of an e-mail and displays a message stating, "Extra Line breaks in this message were removed."
2. Captures the PR_BODY tag from the MSG (if it's present) and converts it into a Unicode string.
3. Gets the native body from the PR_RTF_COMPRESSED tag (if the PR_BODY tag isn't present) and either converts the HTML or the RTF to a Unicode string.
4. Constructs a SHA256 hash from the Unicode string derived in step 2 or 3 above.

HeaderHash

To calculate an e-mail's HeaderHash, the RPC:

1. Constructs a Unicode string containing Subject<crlf>SenderName<crlf>SenderEMail<crlf>ClientSubmitTime.

2. Derives the SHA256 hash from the header string. The ClientSubmitTime is formatted with the following: m/d/yyyy hh:mm:ss AM/PM. The following is an example of a constructed string:

RE: Your last email
Robert Simpson
robert@relativity.com
10/4/2010 05:42:01 PM

RecipientHash

The RPC calculates an e-mail’s RecipientHash through the following steps:

1. Constructs a Unicode string by looping through each recipient in the e-mail and inserting each recipient into the string. Note that BCC is included in the Recipients element of the hash.
2. Derives the SHA256 hash from the recipient string RecipientName<space>RecipientEMail<crlf>. The following is an example of a constructed recipient string of two recipients:

Russell Scarcella
rscarcella@relativity.com
Kristen Vercellino
kvercellino@relativity.com

AttachmentHash

To calculate an e-mail’s AttachmentHash, the RPC:

1. Derive a SHA256 hash for each attachment.
    - If the attachment is a loose file (not an e-mail) the normal standard SHA256 file hash is computed for the attachment.
    - If the attachment is an e-mail, we use the e-mail hashing algorithm described in section 1.2 to generate all four de-dupe hashes. Then, these hashes are combined using the algorithm described in section 1.3 to generate a singular SHA256 attachment hash.
2. Encodes the hash in a Unicode string as a string of hexadecimal numbers without <crlf> separators.
3. Construct a SHA256 hash from the bytes of the composed string in Unicode format. The following is an example of constructed string of two attachments: 80D03318867DB05E40E20CE10B7C8F511B1D0B9F336EF2C787CC3D51B9E26BC9974C9D2C0EEC0F515C770B8282C87C1E8F957FAF34654504520A7ADC2E0E23EA

Calculating the Relativity deduplication hash

To derive the Relativity deduplication hash, the system:

1. Constructs a string that includes the hashes of all four e-mail components described above. See Calculating RPC deduplication hashes for e-mails.
2. Converts that string to a byte array of UTF8 values.
3. Feeds the bytes into a standard MD5/SHA1/SHA256 subroutine, which then computes the hash of the UTF8 byte array.

Note: If two e-mails or loose files have an identical body, attachment, recipient, and header hash, they are duplicates.
https://help.relativity.com/9.2/Content/Relativity/Processing/deduplication_considerations.htm#Technica

Other Processing Tasks

This primer addresses the core functions of ESI processing in e-discovery, but there are many other processing tasks that make up today's powerful processing tools. Some examples include:

Foreign Language Detection

Several commercial and open source processing tools support the ability to recognize and identify foreign language content, enabling selection of the right filters for text extraction, character set selection and diacritical management. Language detection also facilitates assigning content to native speakers for review.

Entropy Testing

Entropy testing is a statistical method by which to identify encrypted files and flag them for special handling.

Decryption

Some processing tools support use of customizable password lists to automate decryption of password-protected items when credentials are known.

Bad Extension Flagging

Most processing tools warn of a mismatch between a file's binary signature and its extension, potentially useful to resolve exceptions and detect data hiding.

Color Detection

When color conveys information, it's useful to detect such usage and direct color-enhanced items to production formats other than grayscale TIFF imaging.

Hidden Content Flagging

It's common for evidence, especially Microsoft Office content, to incorporate relevant content (like collaborative comments in Word documents and PowerPoint speaker notes) that won't appear in the production set. Flagging such items for special handing is a useful way to avoid missing that discoverable (and potentially privileged) content.

N-Gram and Shingle Generation

Increasingly, advanced analytics like predictive coding aid the review process and depend upon the ability to map document content in ways that support algorithmic analysis. N-gram generation and text shingling are text sampling techniques that support latent-semantic analytics.

Optical Character Recognition (OCR)

OCR is the primary means by which text stored as imagery, thus lacking a searchable text layer (*e.g.,* TIFF images, JPGs and PDFs) can be made text searchable. Some processing tools natively support optical character recognition, and others require users to run OCR against exception files in a separate workflow then re-ingest the content accompanied by its OCR text.

Virus Scanning

Files collected in e-discovery may be plagued by malware, so processing may include methods to quarantine afflicted content via virus scanning.

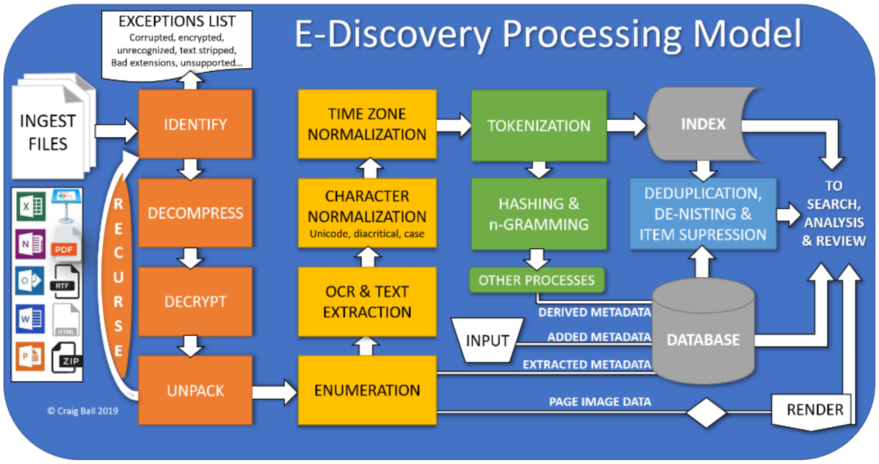
Production Processing

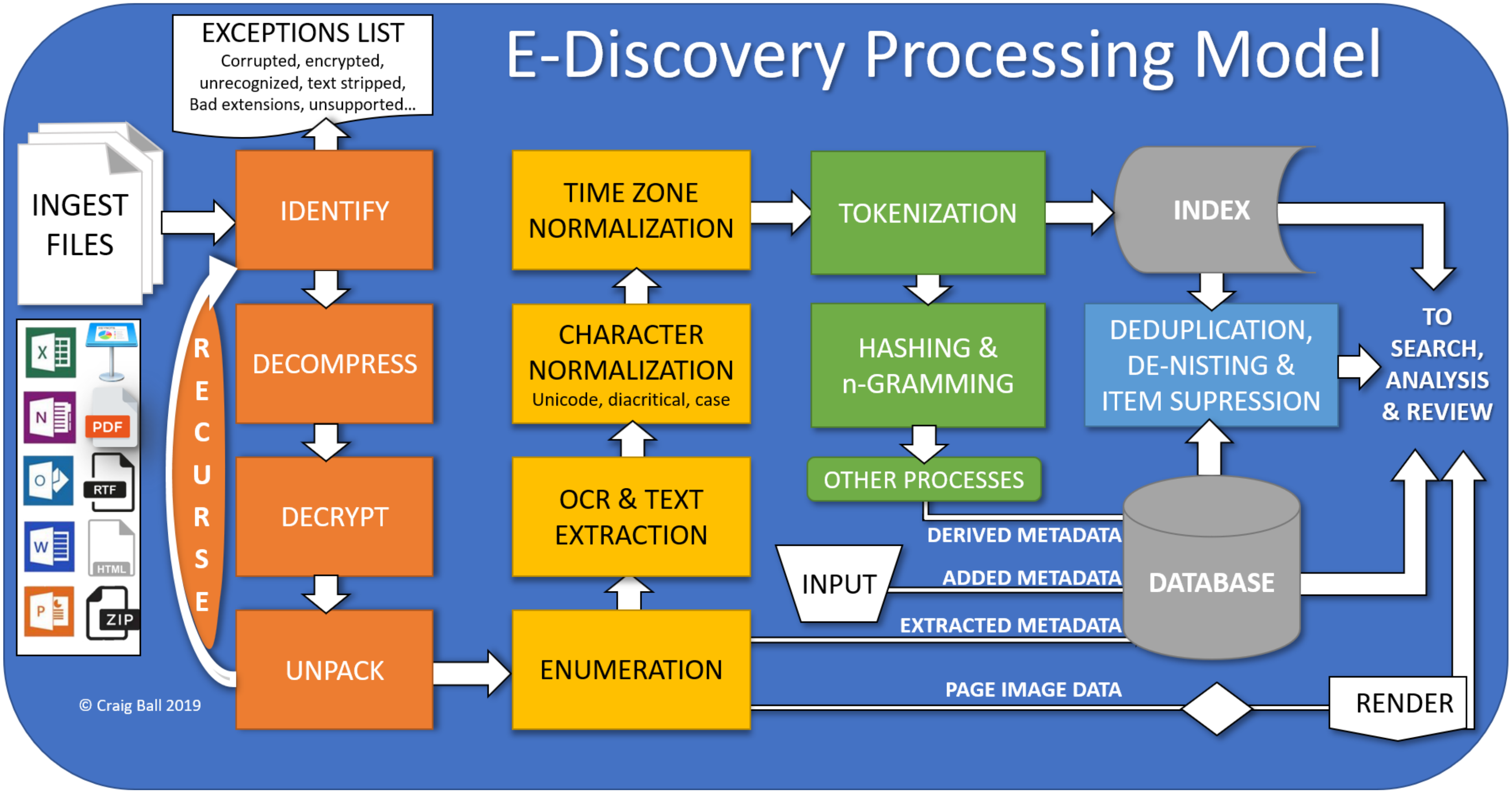
Heretofore, we've concentrated on processing before search and review, but there's typically a processing workflow that follows search and review: **Production Processing**. Some tools and workflows convert items in the collection to imaged formats (TIFF or PDF) before review; in others, imaging is obviated by use of a viewer component of the review tool. If not imaged before review, the e-discovery tool may need to process items selected for production and redaction to imaged formats suited to production and redaction.

Further in conjunction with the production process, the tool will generate a load file to transmit extracted metadata and data describing the organization of the production, such as pointers to TIFF images and text extractions. Production processing will also entail assignment of Bates numbers to the items produced and embossing of Bates numbers and restrictive language (*i.e.*, "Produced Subject to Protective Order").

Illustration 1: E-Discovery Processing Model

Larger version next page.





E-Discovery Processing Model

EXCEPTIONS LIST
Corrupted, encrypted, unrecognized, text stripped, Bad extensions, unsupported...

INGEST FILES

IDENTIFY

DECOMPRESS

DECRYPT

UNPACK

RECURSE

TIME ZONE NORMALIZATION

CHARACTER NORMALIZATION
Unicode, diacritical, case

OCR & TEXT EXTRACTION

ENUMERATION

TOKENIZATION

HASHING & n-GRAMMING

OTHER PROCESSES

INDEX

DEDUPLICATION, DE-NISTING & ITEM SUPRESSION

DATABASE

INPUT

DERIVED METADATA

ADDED METADATA

EXTRACTED METADATA

PAGE IMAGE DATA

RENDER

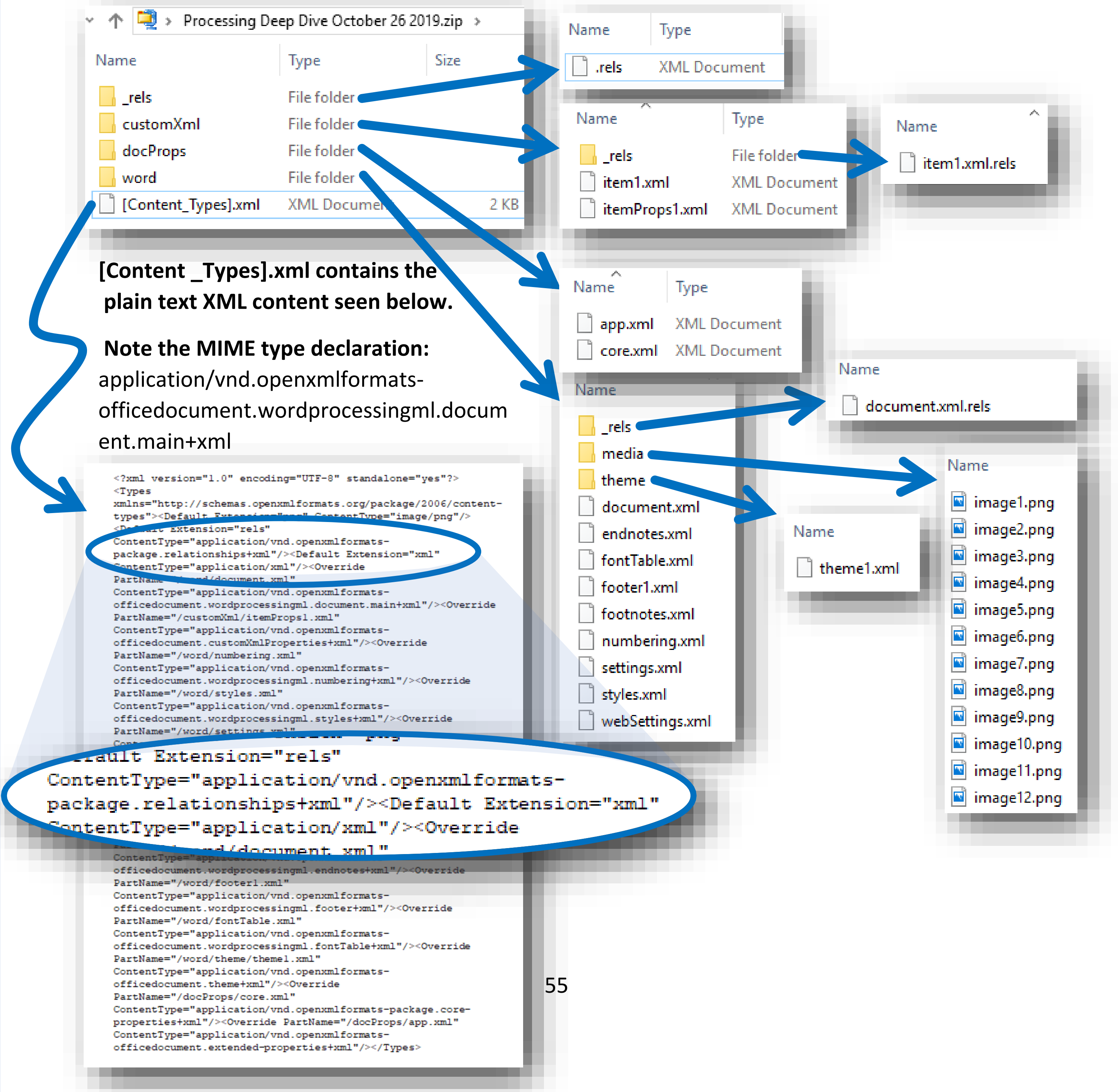
TO SEARCH, ANALYSIS & REVIEW

© Craig Ball 2019

Illustration 2: Anatomy of a Word DOCX File

By changing a Word document's extension from .DOCX to .ZIP, you can "trick" Windows into decompressing the file and sneak a peek into its internal structure and contents. Those contents little resemble the document you'd see in Word; but, as you peruse the various folders and explore their contents, you'll find the text, embedded images, formatting instructions and other components Word assembles to compose the document.

The root level of the decompressed file (below left) contains four folders and one XML file.



Name	Type	Size
_rels	File folder	
customXml	File folder	
docProps	File folder	
word	File folder	
[Content_Types].xml	XML Documen	2 KB

Name	Type
.rels	XML Document

Name	Type
_rels	File folder
item1.xml	XML Document
itemProps1.xml	XML Document

Name
item1.xml.rels

Name	Type
app.xml	XML Document
core.xml	XML Document

Name
_rels
media
theme
document.xml
endnotes.xml
fontTable.xml
footer1.xml
footnotes.xml
numbering.xml
settings.xml
styles.xml
webSettings.xml

Name
document.xml.rels

Name
theme1.xml

Name
image1.png
image2.png
image3.png
image4.png
image5.png
image6.png
image7.png
image8.png
image9.png
image10.png
image11.png
image12.png

[Content _Types].xml contains the plain text XML content seen below.

Note the MIME type declaration: application/vnd.openxmlformats-officedocument.wordprocessingml.document.main+xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Types
xmlns="http://schemas.openxmlformats.org/package/2006/content-
types"><Default Extension="png" ContentType="image/png"/>
<Default Extension="rels"
ContentType="application/vnd.openxmlformats-
package.relationships+xml"/><Default Extension="xml"
ContentType="application/xml"/><Override
PartName="/word/document.xml"
ContentType="application/vnd.openxmlformats-
officedocument.wordprocessingml.document.main+xml"/><Override
PartName="/customXml/itemProps1.xml"
ContentType="application/vnd.openxmlformats-
officedocument.customXmlProperties+xml"/><Override
PartName="/word/numbering.xml"
ContentType="application/vnd.openxmlformats-
officedocument.wordprocessingml.numbering+xml"/><Override
PartName="/word/styles.xml"
ContentType="application/vnd.openxmlformats-
officedocument.wordprocessingml.styles+xml"/><Override
PartName="/word/settings.xml"
Cont
```

```
efault Extension="rels"
ContentType="application/vnd.openxmlformats-
package.relationships+xml"/><Default Extension="xml"
ContentType="application/xml"/><Override
```

```
ContentType="appl
officedocument.wordprocessingml.endnotes+xml"/><Override
PartName="/word/footer1.xml"
ContentType="application/vnd.openxmlformats-
officedocument.wordprocessingml.footer+xml"/><Override
PartName="/word/fontTable.xml"
ContentType="application/vnd.openxmlformats-
officedocument.wordprocessingml.fontTable+xml"/><Override
PartName="/word/theme/theme1.xml"
ContentType="application/vnd.openxmlformats-
officedocument.theme+xml"/><Override
PartName="/docProps/core.xml"
ContentType="application/vnd.openxmlformats-package.core-
properties+xml"/><Override PartName="/docProps/app.xml"
ContentType="application/vnd.openxmlformats-
officedocument.extended-properties+xml"/></Types>
```