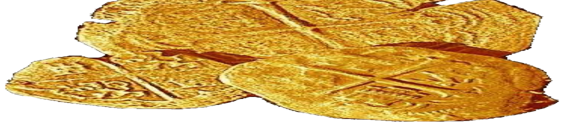
Bzet: A Tree Oriented Compression Technique

Robert Uzgalis
Tigertail Associates

IFIP WG2.1 – 2012 Feb 8 – Rome, Italy



31 January 2012

What’s a Bitstring

A bitstring is a sequential series of bits,
that is 1s and 0s or True and False marks.

Position in a bitstring is important. That is
1010 is different from 0011 even though they both
have two True bits and two False bits.

A good example of a bitstring is a computer register.

Bitstrings are Information

Bits in a bitstring represent information. Sometimes numerical, positional, or just symbolic.

- Numerical ... each bit is a power of 2 and the sum represents the integer.
- Positional ... each bit corresponds to a position in something -- like a record in a data base.
- Symbolic ... each bit means something. Like the first bit means: is a person. The second bit means: is female...

Ballooning Bitstrings

Ballooning bitstrings come from several sources

- In numeric representations, from overflow
- In positional representations, an enlargement of the domain.
- In symbolic representations, more information to record.

Bitstring is a bit of a mouthful, in the future just bitsets, which is shorter and easier to say.

Sparse Bitsets

A generalization: As bitsets balloon they tend to get sparse.

- For example most algorithms use relatively small numbers, far less than the size of register.
- In a large database the number of records that have any one given property tend to be small.

Taking advantage of Sparseness

- The usual way of taking advantage of sparseness in large bitsets is to compress them with run-length encoding.
- Run length encoding is a linear compression technique that replaces a run of zeros or ones with a length and an attribute.
- Ted Glaser patented a method for performing Binary operations on binary run-length encoded bitsets. (US Patent 5,036,457 1991-07-30)

The Bzet* Proposal

Normally bitsets are represented in computers by:

- A fixed length register or memory location,A series of bits in memory,A compressed, run-length encoded string
- New idea: that a positional, varying length, logarithmic encoding, as a general, more useful, representation. Call that representation a Bzet*.

* patent pending.

Binary Bzets

Assume that we want to represent a bitset as a binary tree of bits. It can be done this way:

Use 3 values: call them 1,T, and 0. 1 if values below this position in the tree are all 1, T if they are mixed, and 0 if they are all 0.

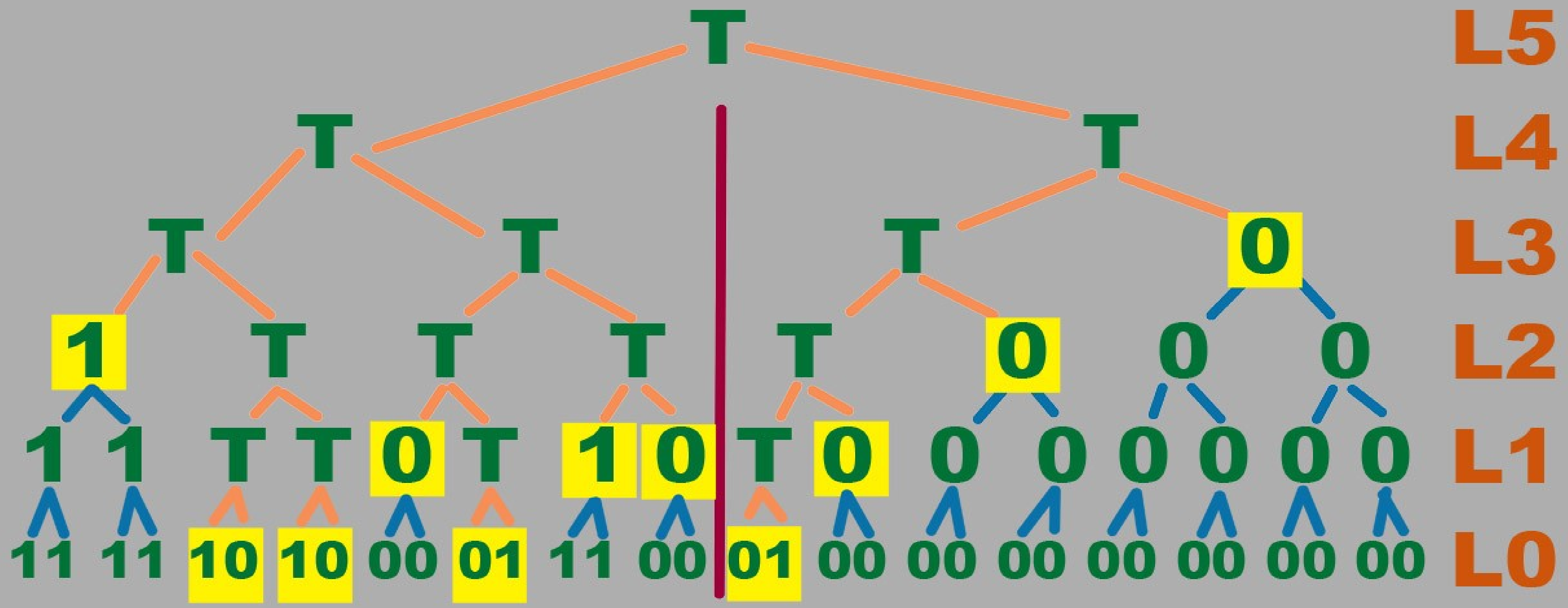
If the size of the bitset is not a power of 2, expand the bitset to the nearest power of 2 greater than its current size and append 0s to the right to fill. Build the tree. Do a top-down, depth-first, traversal of the tree writing down the nodes. A 1 or 0 terminates traversal of a subtree. Prefix the node list with its depth.

Probably the best way to understand this encoding is to see the tree and the encoding at the same time:

Binary BZET -- 5 Level 32 bit

5:TTT1TT10T10TT0T01T10
54322100100321100211

TTTT01000
432100123



Note:

- There can be no node that is either 00 or 11 except the top node of the tree. If one exists it must be collapsed into the node above it.
- This is not a particularly compact representation of the bitset. But it is a useful one.
- Note that binary BZETs can be directly compared without traversal.

64 or 128 bit BZETs

For the same data would look almost the same...

```
64 bit  6:T  TTT1TT10T10TT0T01T10TTTT01000 0
128 bit 7:TT TTT1TT10T10TT0T01T10TTTT01000 00
```

This extends the bitset on the right with 0s, the bit indexes stay the same.

```
64 bit  6:T0   TTT1TT10T10TT0T01T10TTTT01000
128 bit 7:T0T0 TTT1TT10T10TT0T01T10TTTT01000
```

This shifts the bits producing leading zeros, the bit indexes change.

Advantages of BZETS

- Typically bitsets get compressed, although for some bitsets, results can be larger.
- For all Boolean operations Bzet strings can be operated upon serially.
- A Bzet is a canonical representation of the bitset. Thus equal and not equal are easy operations.
- In a numeric binary representation they can be compared directly given that the bits for $0 < T < 1$.
- Operation times depend only on the length of the representation not on the size of the bitset itself.

The Software Implementation

As software, bit handling is a slightly clumsy and a bit inefficient on modern computers. Words are a bit easier to handle. In this case 8-bit words are used.

So the real question is how to take this Bzet technique and do a reasonable software implementation.

Requirements

We need two bits to represent every subtree:

- 00 for an all zero subtree
- 01 for a mixed value subtree
- 10 for an all one subtree
- 11 is unused

The first bit represents data: 1 or 0 value

The second bit represents structure: 1 for a mixed subtree 0 for constant subtree. With the 11 combination unused/bad/illegal.

Collecting Bzet Bits into Bytes Creating Oct-Trees

Assemble 8 data bits to represent the 8 trees into a byte. Assemble 8 structure bits to represent the 8 trees into another byte.

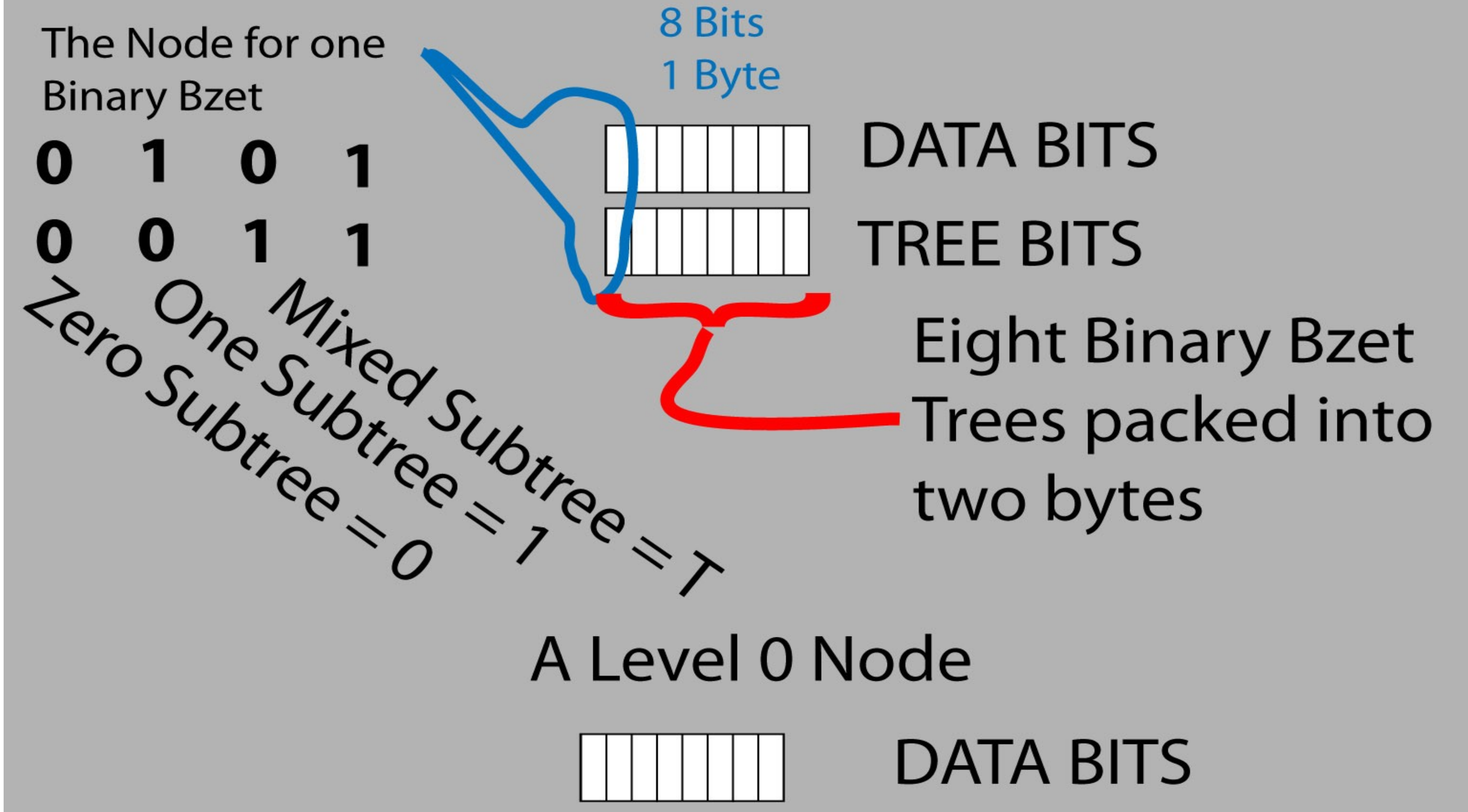
Thus one gets an Oct-tree node in two bytes, represented in hexadecimal as [0xnn,0xtt]

A Level 0 Oct-tree node has only data so it can be represented as D(0xnn)

The serial representation will then look like--

levels: followed by a mixture of nodes.

One Node of an Oct-Tree Bzet

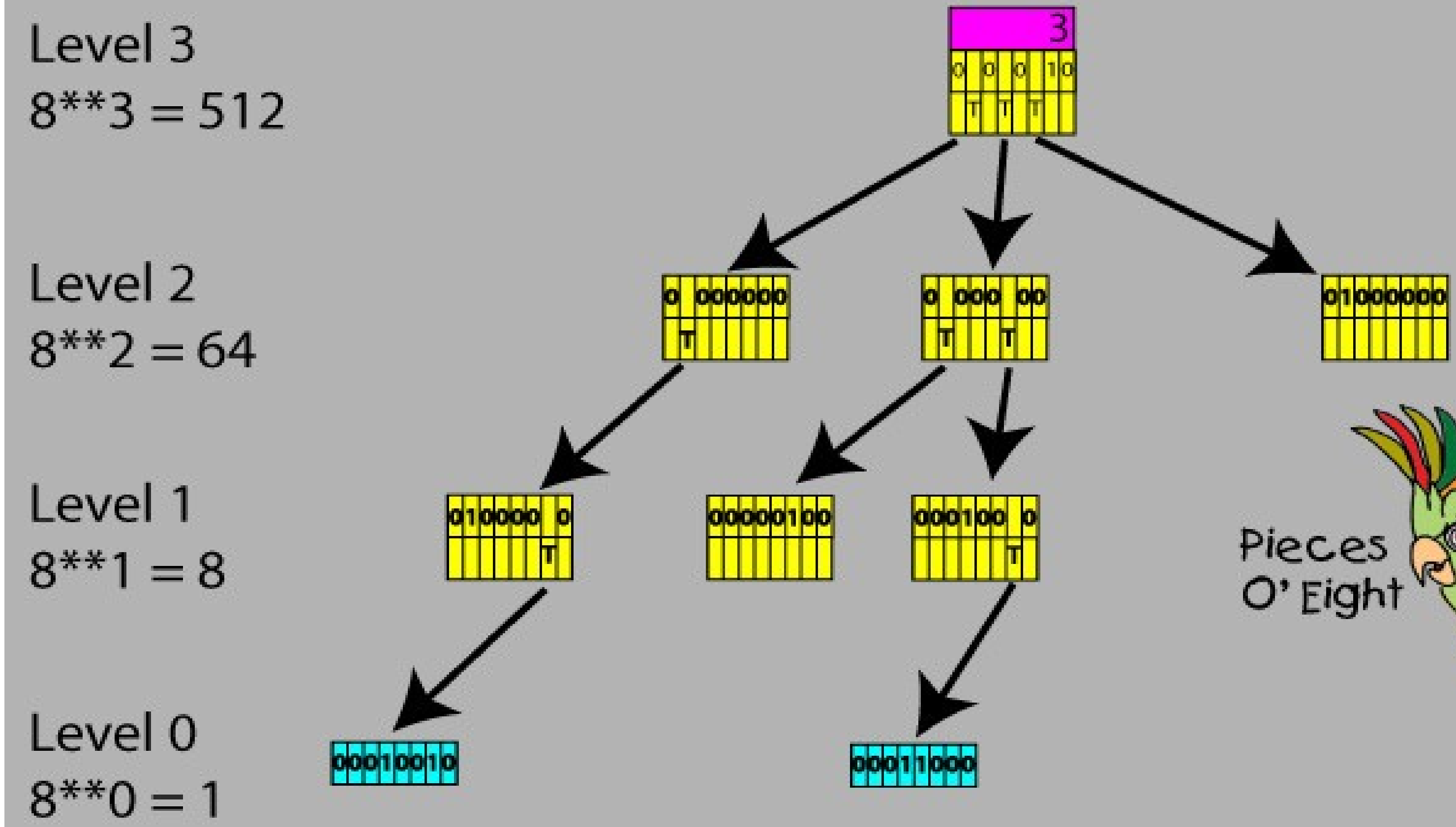


A Note to Avoid Confusion

- Note that because the eight nodes are now grouped together: a zero or a one means all sub-trees that branch terminate in all zeros or ones.
- And likewise a mixed subtree will show up in the proper order subsequently in the linear form, just as it did in the Binary form of the Bzet, only it will be an 8 way node.

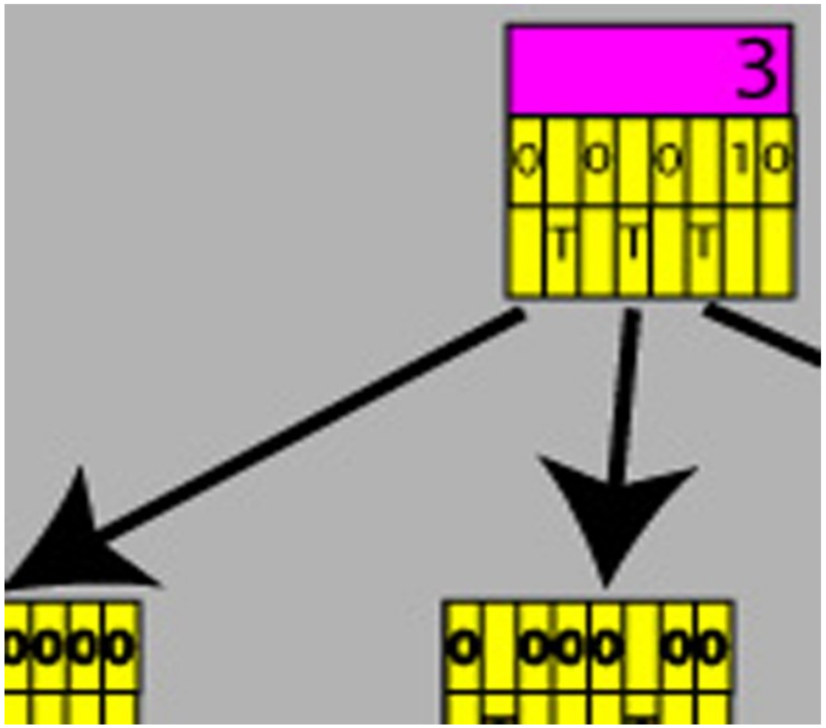
An Level 3 Bzet Oct-Tree

Holds a maximum of 4096 bits



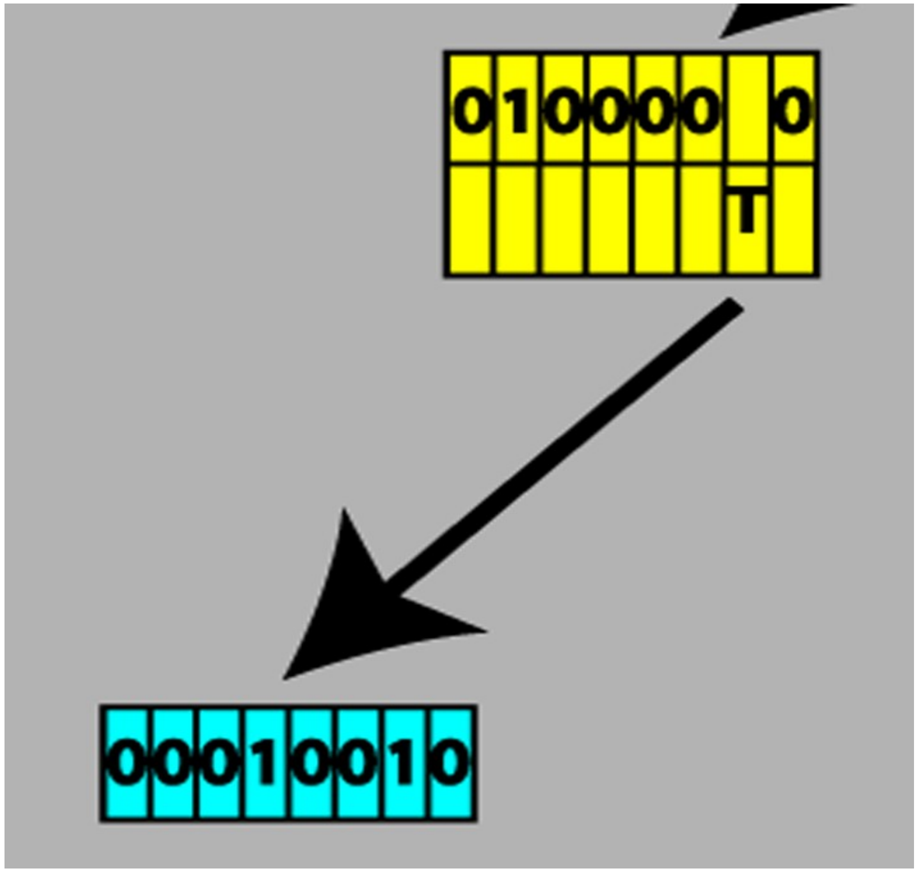
```
3L [02-54][00-40][40-02]D(12)
          [00-44][04-00]
                 [10-02]D(18)
          [40-00]
```

A Closer Look:



- 3 is level of top of tree.
- **0x0x** 0x**10** is the data... a **0** means all 0s below; a **1** means all 1s. The x-s are zero and ignored.
- xTxT xTxx shows where subtrees are: T is a 1. The x-s are zero and ignored.
- The nodes will be expressed in hexadecimal: 3L [02-54] with 02 in hex representing: 0000 0010 and 54 in hex representing the tree bits: 0101 0100
- Note the top level node has 54 as its tree bits so there will be 3 subtrees to process. 5 has two 1s and 4 has one.

A Closer Look



Level 0 never has a tree below it. So it is represented as: D(12)

This part of the tree would then be expressed as [40-02]D(12)

Note that using this method there will be as many D elements as there are Ts at the first level. 02 has one bit on so there is only one data byte.

Bzet Octane Trees

Here are some simple examples:

- MT = 1L [00-00]
- Bit 1 = 0L D(40)
- Bit 24 = 1L [00-10]D(80)
- Bit 300 = 2L [00-08][00-04]D(08)
  Note: 300 in octal is 454. And that
  four 0s then 1 is 08; five 0s then 1 is 04.
  So it is easy to encode a BZET bit from its
  index using octal.

Another Example

```
Bzet0 = 1L [00-05]D(4f)D(f4)
The following bits are true:
    0:     41;    44;    45;    46;    47;
    5:     56;    57;    58;    59;    61.
 10 True bits found
```

More Complex

bzet1 is

```
4L [00-05][00-80][00-40][00-01]D(34)
          [00-40][00-70][00-10]D(1f)
                        [00-0c]D(2e)D(e2)
                        [00-01]D(3b)
```

```
 0:  20,826; 20,827; 20,829; 20,856; 20,857;
 5:  20,858; 20,859; 20,860; 20,861; 20,862;
10:  20,863; 29,275; 29,276; 29,277; 29,278;
15:  29,279; 29,346; 29,348; 29,349; 29,350;
20:  29,352; 29,353; 29,354; 29,358; 29,434;
25:  29,435; 29,436; 29,438; 29,439.
29 True bits found
```

Boolean Operations on Bzets

Take for a simple example NOT.

- NOT each data byte and turn off bits that correspond to Tree bits to maintain the 0-1 structure of Tree elements.

Bzet0 = **`1L [00-05]D(4f)D(f4)`**

~Bzet0 = **`1L [fa-05]D(b0)D(0b)`**

Binary Boolean Operations

There are sixteen possible binary Boolean operations. The three most common are AND, OR, and XOR

If we are to operate on trees not just individual bits we need to characterize what operations are necessary to process trees.

Subtree Operations

Four subtree operations are necessary and sufficient for all Boolean operations:

- Copy a subtree
- Copy a subtree and Invert it
- Delete subtree and compress it to 0s
- Delete subtree and compress it to 1s

Bzet AND, OR and XOR

There are 6 cases when dealing with compressed data, data to data, tree to compressed tree, tree to tree:

		0	1	2	3	4	5
oper		*dd*	*0T*	*T0*	*1T*	*T1*	*TT*
	A	`0011`					
	B	`0101`					
AND		`0001`	DB0	DA0	CB	CA	R
OR		`0111`	CB	CA	DB1	DA1	R
XOR		`0110`	CB	CA	NB	NA	R

Other Binary Boolean Operations

- They can be expressed in the same way.
- The operation table works on any Bzet trees. (i.e. Binary Bzets are the same as Quad-tree, or Oct-tree Bzets)
- None of the fundamental Boolean operations require backing up... so all processing is serial.
- Operations are done on the compressed form, no expansion is necessary.

Advantages

- Bzets provide an attractive way to serialize processing of bitsets.
- Provides logarithmic rather than linear compression of bitsets.
- Provides for processing large bitsets with compression both in storage and processing time.

Disadvantages

As with RLE compression this only works well with sparse bitsets. The rule of thumb for RLE is use it if the bit density is lower than 8%. In that sense Bzets seem similar but there are many parameters one can tune in a Bzet that are not available in RLE compression.

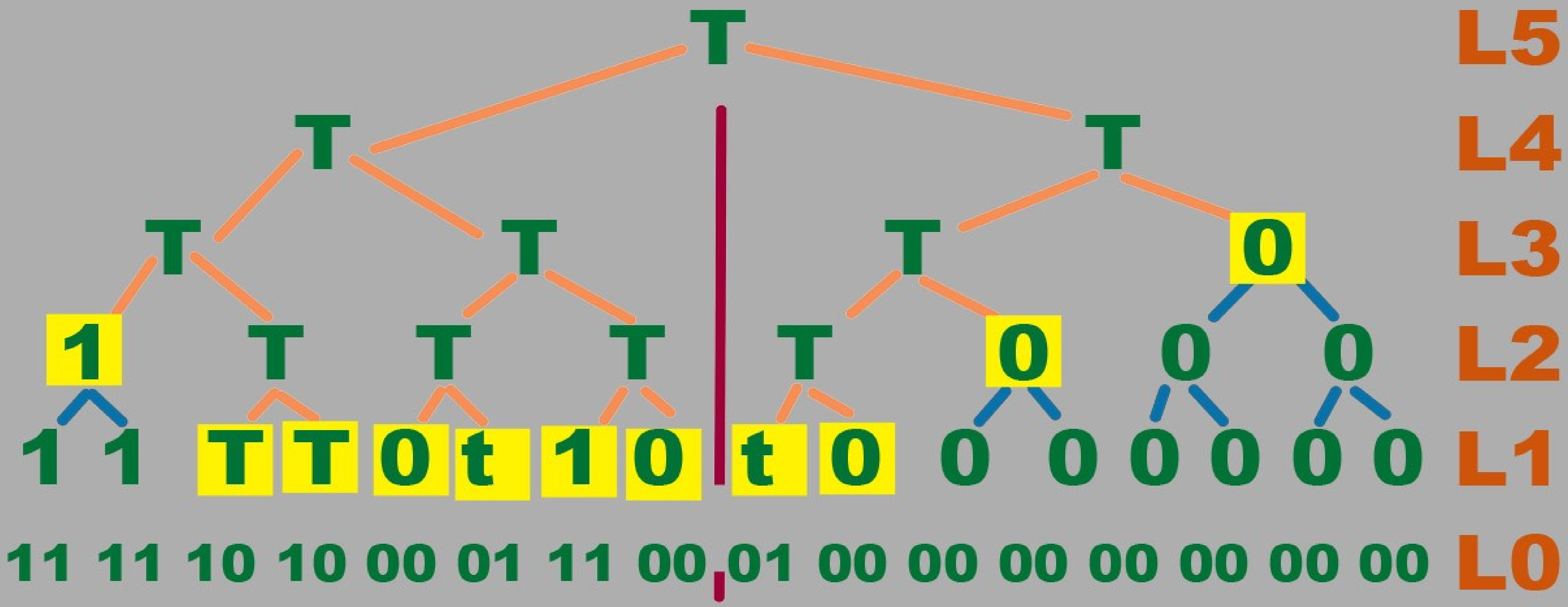
- For example in a Binary Bzet one could easily compress away the bottom level of the tree.

Binary BZET -- 5 Level 32 bit

```
5:TTT1TTTTT0tT10 TTTt000
  54322113211211 4321123
```

T means mixed bits; at L1 it means 1-0.
t means 0-1 below any level.

This encoding has 4 states.



Generalize this Notion

There is no need to restrict ourselves to level 1. We can make any level we want to to be the bottom level. This decreases compression of short strings of all 1s or all 0s, but increases compression overall by eliminating tree structure nodes for local variations.

The question is, of course: where is the optimal level to make the bottom level?

Implementation

Initially a Python 3 Oct-tree implementation exists with level 0 being the bottom level.

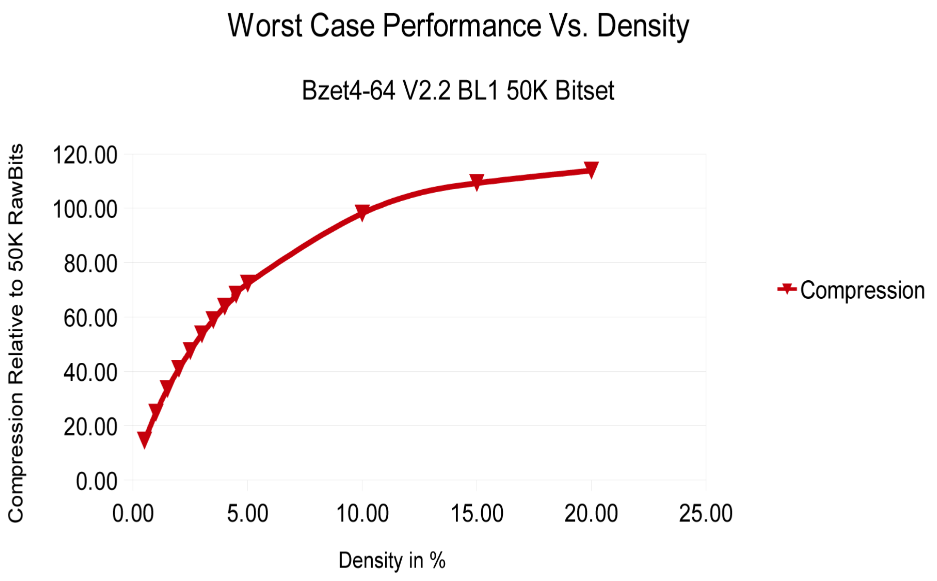
Fifteen UCLA Software Engineering Students were broken into 3 groups to implement Bzets in C or C++ and implement a Python3 interface for their code.

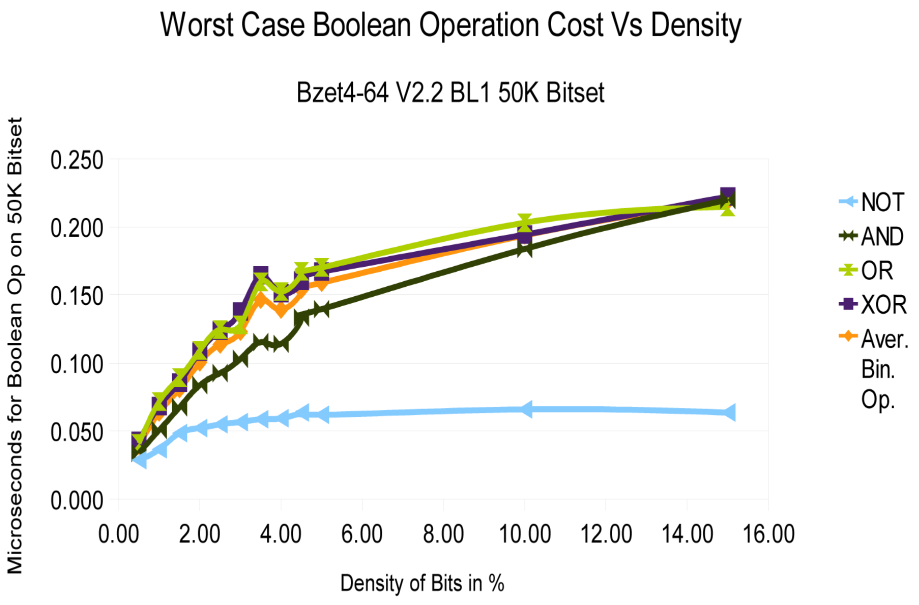
The three groups each with five members did binary, quad, and octal Bzet implementations.

Bzet4

- Has a single byte for each node: 4 data bits, and 4 structure bits.
- Has a bottom level of 1 with 16-bit parallel operations being performed.
- Written in C++, with a ctypes interface to Python3

Bzet4 Performance





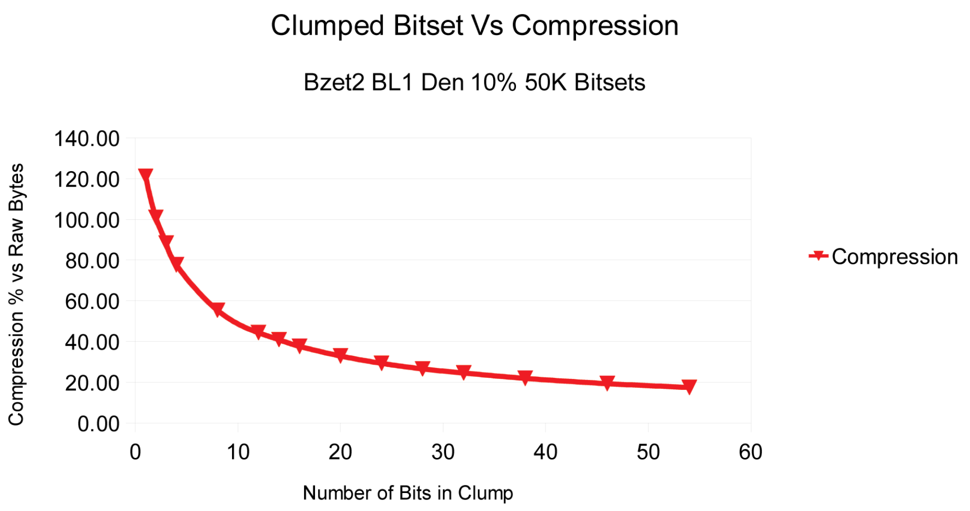
Bzet2

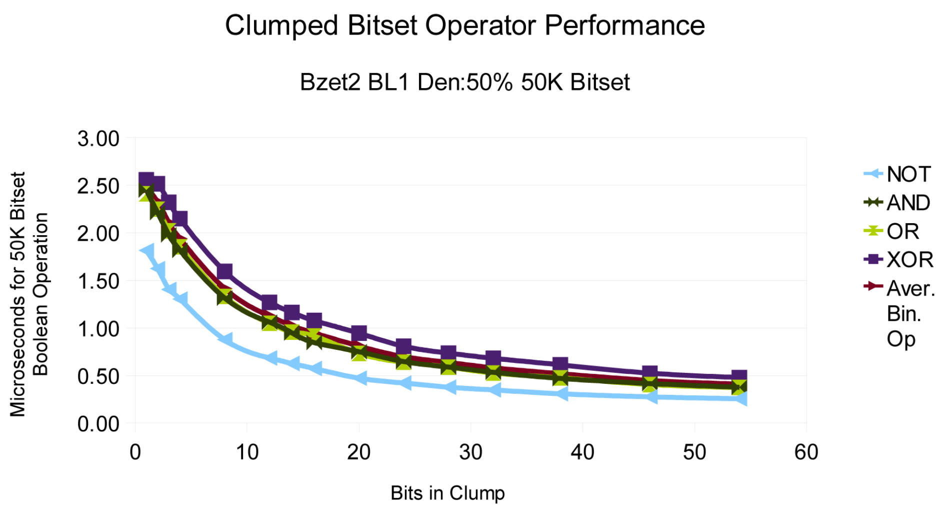
- Does a binary tree with 2 bits for each node.
- Has a parametrized bottom level:
  - 0 – 1 bit bottom level (slow)
  - 1 – 2 bit bottom level ...
  - 6 – 64 bit bottom level (fast)
- Written in C with a ctypes interface to Python3

Let's Look a BL1 with 2-bit processing, but for a change, let's hold density constant and vary bit clumping.

Bzet2 BL1 Performance

X-axis is number of clumped bits





Conclusions

- The C/C++ implementations ran about 200x faster than my Python3 version.
- Performance is awesome in general. Processing a Boolean binary operation on 50K bits (worst case) in 0.2 microseconds is fast, in good cases it drops as low as 0.05 ms.
- Space/Time performance stays good even when the compression technique fails because the bitset isn't suitable for compression. Even worst case compression with bzets is only mildly worse that having raw bitsets, So it probably isn't worth switching the technique when the compression technique begins to fail.

Conclusions

- Bzets clearly meet the need for Database traditional uses and in fact may expand the uses since worst case performance is still pretty good. One could imagine a database that is built primarily out of bitmaps as its primary logical engine.
- Binary bzets offer another possibility. If worst case performance is about equal to raw bit performance, why not encode all bitsets as Bzets?

The Fantasy Future

Thus, Binary Bzets could possibly form the basis for a new computer architecture that has variable length registers and is primarily bit serial in nature.

- Floating point is no longer necessary, although it may still be wanted but it could be far more flexible, with variable sized exponent and mantissa fields.
- Lines or words could be packed into single entities.

Implementation

- Python 3 Module
- Underlying data-structure is a bytes string.
- Provides bitset constructors, operations, and some specialized functions.
- Currently in Beta-test.

Julian Days: Days since 1 Jan 4713 BC

- def julian_day(yr, month, day):
-     a = (14-month)//12
-     y = yr + 4800 - a
-     m = month + 12*a -3
-     return day + (153*m+2)//5 + 365*y + y//4 - y//100 + y//400 – 32045

Julian day algorithm from Wikipedia

Julian Time Bitsets

- So if today is: 7 Jan 2012
  then the Julian day is 2455934
- Suppose my birthday was: 29 Sep 1940
  then its Julian day is 2,429,902
- So I would be 26,033 days old today.
- My birthday, one bit as a bitset:

```
buzbday = BZET( julian_day(1940,9,29) )
7L[00-40][00-40][00-20][00-40][00-40][00-01][00-40]D(02)
```

- My life, 26,033 bits as a bitset:

```
buz = BZET( [(buzday,julian_day(2012,1,7))] )
7L[00-40][00-40][00-20][3e-41][3f-40][00-01][3f-40]D(03)[f0-08]
[f8-04][fe-01]D(fe)
```

BZET Operations

- 1 September 1939 – Invasion of Poland
- 15 August 1945 – Japan agrees to Surrender

- WW II as a bitset:

```
wwii = BZET( [(julian_day(1939,9,1),julian_day(1945,8,15))] )
7L[00-40][00-40][00-20][00-40][38-44][3f-40][7f-80]D(0f)
                                     [e0-10][00-80]D(f0)
wwii.COUNT() = 2,176 days
```

- My life and WW II:

```
buzww2 = buz & wwii
7L[00-40][00-40][00-20][00-40][38-44][00-01][3f-40]D(03)
                                     [e0-10][00-80]D(f0)
buzww2.COUNT() = 1,782 days
```