Memory-mapping, Tightly Packed Tries, and Suffix Arrays

Ulrich Germann

October 21, 2009

So much data, so little memory ...

English Gigaword 5-gram LM					
unigrams	8,135,668	4-grams	123,297,762	file size (txt)	14.0GB
bigrams	47,159,160	5-grams	120,416,442	file size (.gz)	3.7GB
trigrams	116,206,275				

	Implementation	File	Virt. Mem.	Real Mem.	ttfr[a]	wall[b]
full model loaded	**SRILM (map)**	5.2 GB	16.3 GB	15.3 GB	940s	1136s
	SRILM (array)	5.2 GB	13.0 GB	12.9 GB	230s	232s
	IRST	5.1 GB	5.5 GB	5.4 GB	614s	615s
	IRST (mmap)	5.1 GB	5.5 GB	1.6 GB	548s	744s
	IRST-Q	3.1 GB	3.5 GB	3.4 GB	588s	589s
	IRST-Q (mmap)	3.1 GB	3.5 GB	1.4 GB	548s	674s
	Portage	8.0 GB	10.5 GB	10.5 GB	120s	**122s**
	TPT	**2.9 GB**	**3.4 GB**	**1.4 GB**	**2s**	127s
filtered	**SRILM**	5.2 GB	6.0 GB	5.9 GB	111s	112.0s
	SRILM-C	5.2 GB	4.6 GB	4.5 GB	112s	113.0s
	Portage	8.0 GB	4.5 GB	4.4 GB	120s	122.0s

[a] Time to First Response

[b] Wall time computing the perplexity of 10,000 lines of text

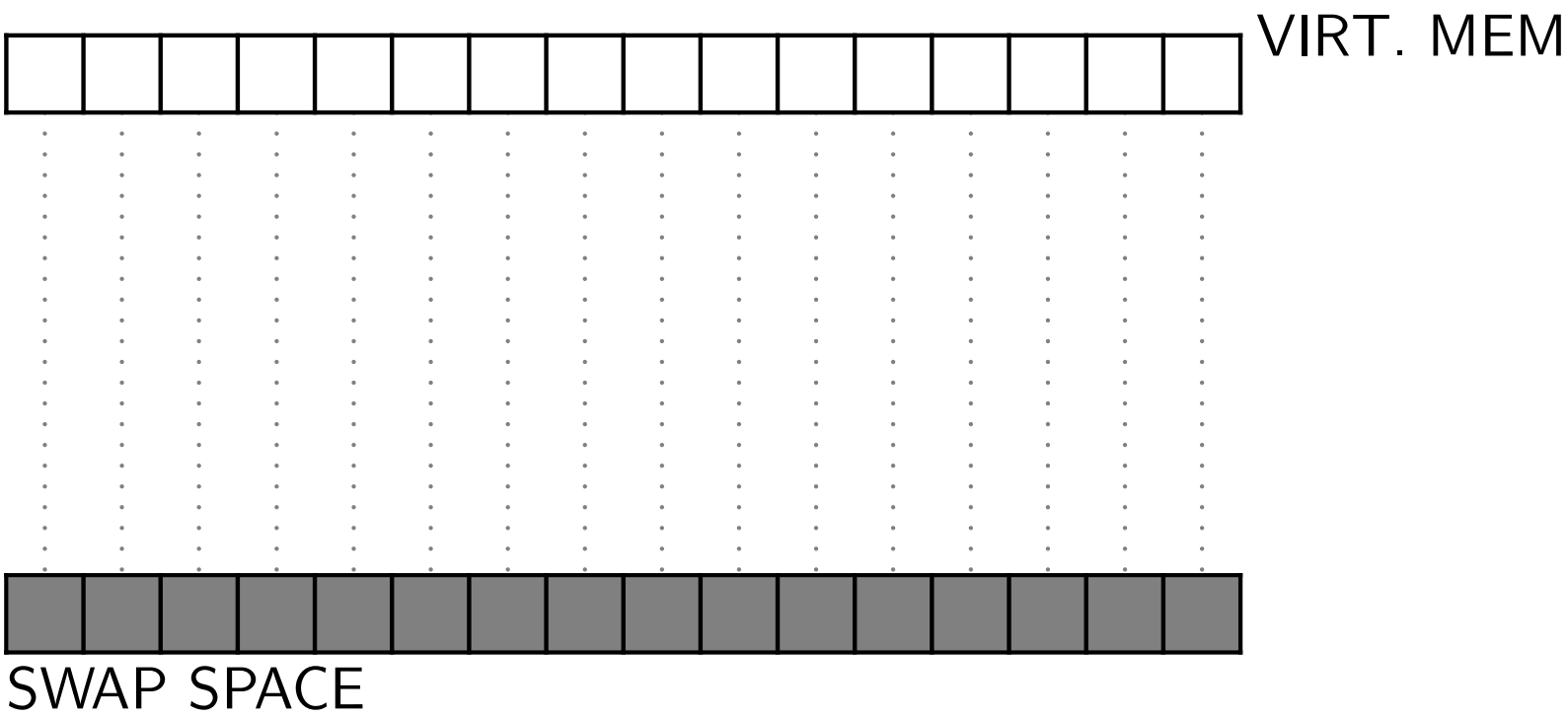
Dealing with large models

- Compact representations
- Distributed databases [Brants et al., 2007]
- Lossy compression or pruning
  [Stolcke, 1998; Whittaker&Raj, 2001, Federico&Bertoldi, 2006, Johnson et al., 2007]
- Hash functions, Bloom filters, Bloomier filters [Talbot&Brants, 2008]
  - Not enumerable
- Lazy loading and memory mapping [Zens&Ney, 2007, Federico&Bertoldi, 2006]

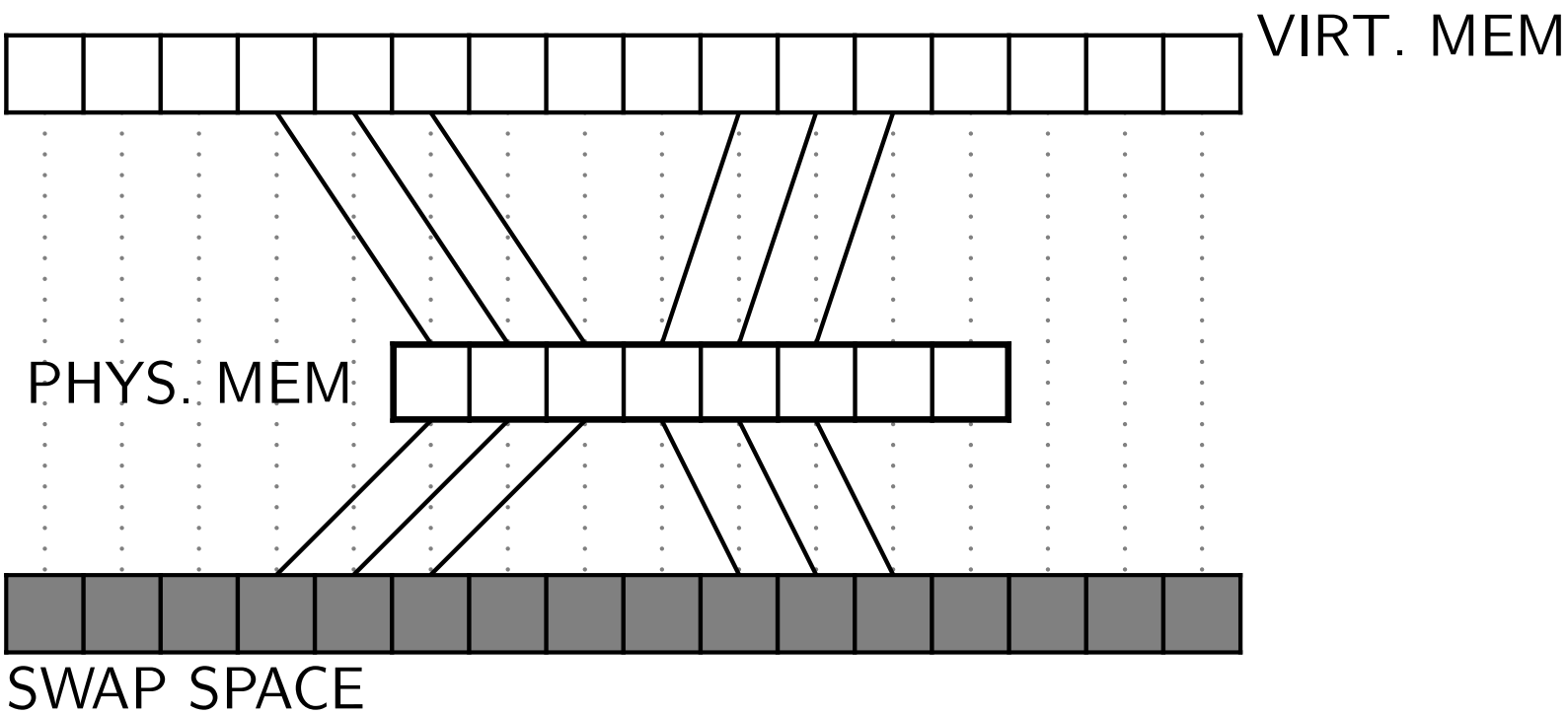
Part I

Memory Mapping

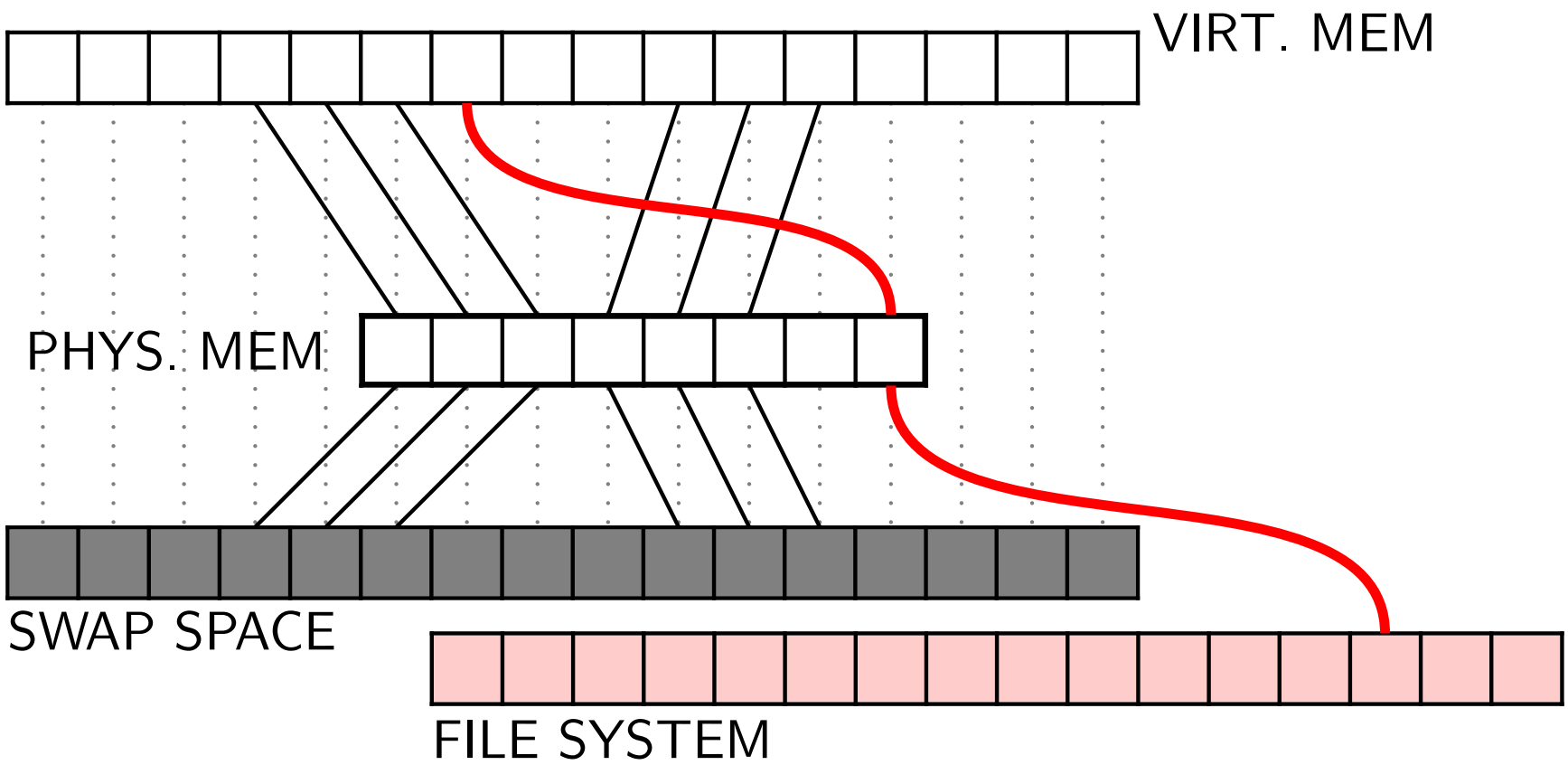
Memory Mapping



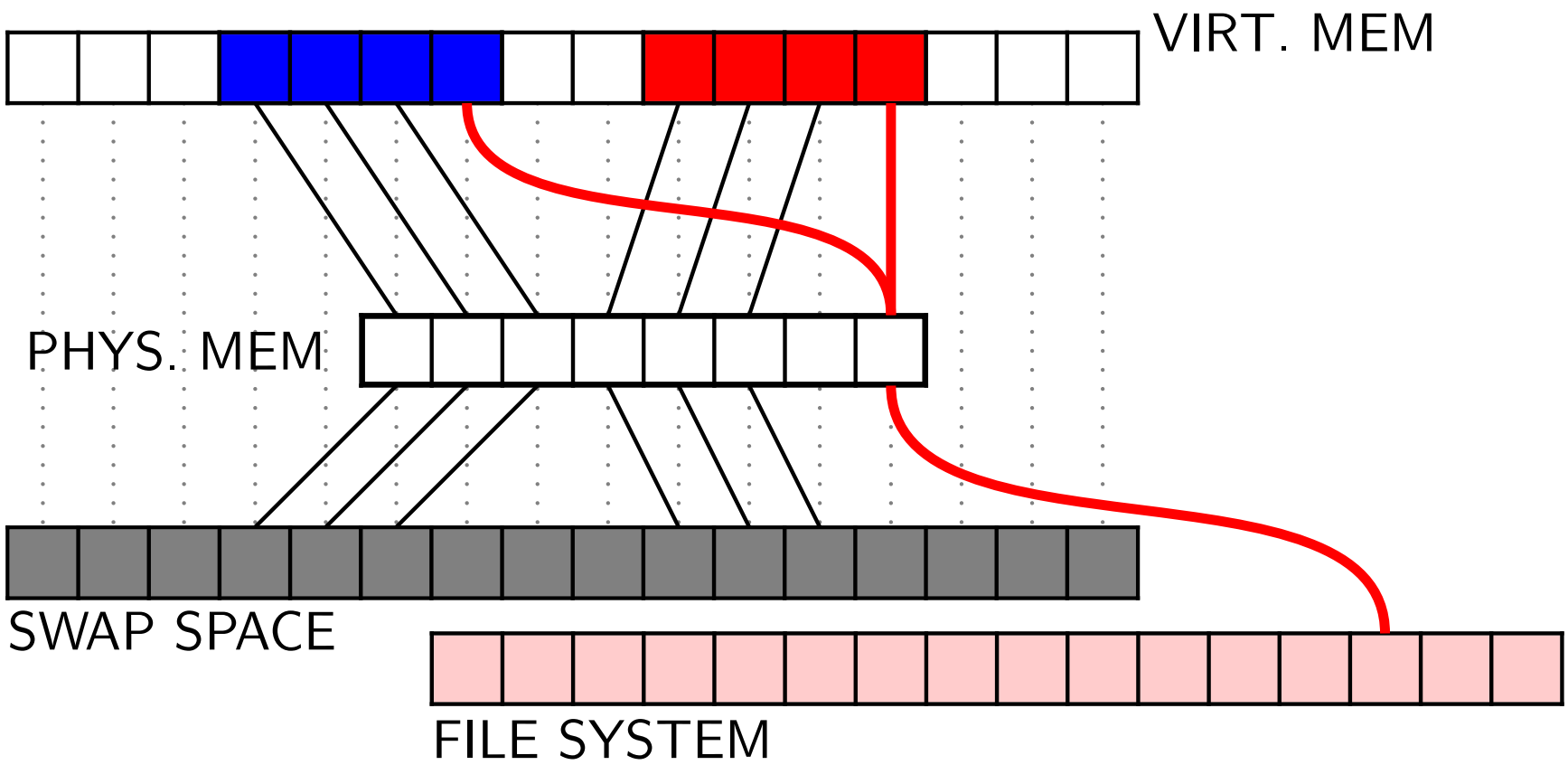
Memory Mapping



Memory Mapping



Memory Mapping



Memory Mapping

Don't do it yourself! Leave it to virtual memory manager.

Perks

- Fastest way of getting data from disk into memory.
- Lazy loading ($\Rightarrow$ short time to first response).
- If data is read-only, swapping is fast (no need to write to disk).
- Caching effecting across multiple runs (pages often stay in memory).
- Multiple processes can share memory.
- Trivial to implement — there are library functions.

Caveats

- For small amounts of data, the conventional file interface is more efficient.

Memory mapping: Java

```
FileChannel ch = new RandomAccessFile("myfile","r")
                        .getChannel();

long fsize = ch.size()

ByteBuffer mapped
= ch.map(FileChannel.MapMode.READ_ONLY,0,fsize);
```

Disclaimer:

- I don't usually program in Java, so I haven't actually tried this.

Source of this example:

`http://www.developer.com/java/other/article.php/10936_1548681_2/Introduction-to-Memory-Mapped-IO-in-Java.htm`

See also:

`http://java.sun.com/javase/6/docs/api/java/nio/channels/FileChannel.html`

Memory mapping: Python

```
import mmap

fileName='myfile'
f = open(fileName,"r+")
mapped = mmap.mmap(f.fileno(),0)
```

mmap objects can be used like

- **files:** e.g., mapped.seek(0), mapped.read(8), etc.)
- **strings:** e.g., mapped[0:4].

See also:
`http://www.python.org/doc/current/lib/module-mmap.html`

Memory mapping: Perl

```
use File::Map ':map';
map_file my $mmap, $filename;
```

Perl does not have a built-in convenient way of accessing individual bytes in strings, but you can tie an array to the string:

```
use Tie::CharArray;
...
tie my @mapped, 'Tie::CharArray', $mmap;
```

See also:

- `http://search.cpan.org/~leont/File-Map-0.16/lib/File/Map.pm`
- `http://search.cpan.org/~iltzu/Tie-CharArray-1.00/CharArray.pm`

Memory mapping: Perl (Continued)

Caveats:

- I haven't actually tried it out yet.
- Still seems a bit home-cooked (listed on CPAN, but not part of the standard distribution; File::Map is v0.16; single-author packages).

Memory mapping: C++

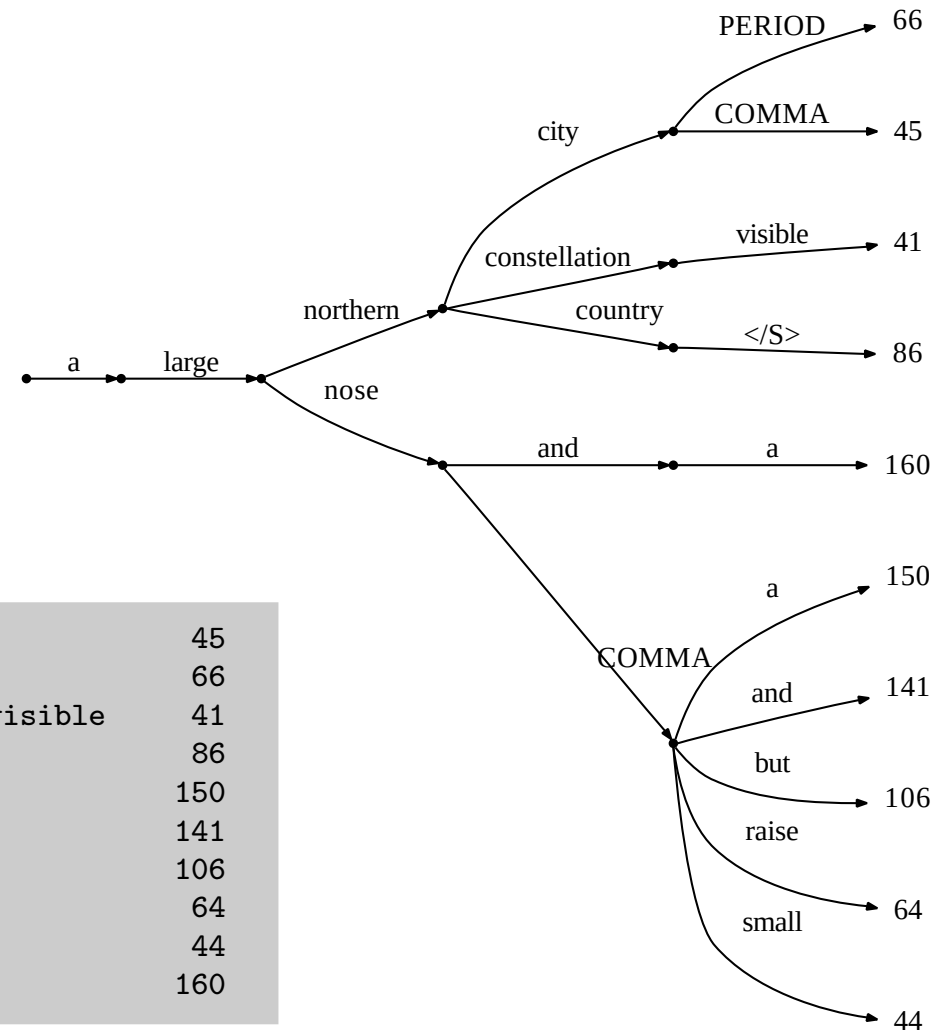
```
#include <boost/iostreams/device/mapped_file.hpp>
...
boost::iostreams::mapped_file_source() mapped;
mapped.open("myfile");
...
char const* startData = mapped.data();
char const* endData = startData+mapped.size();
```

Part II

Tightly Packed Tries

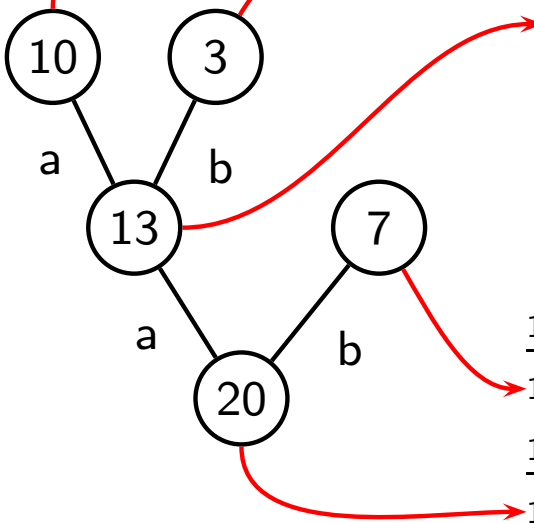
Tries (a.k.a. prefix trees)

Tries offer a compact way of storing token sequences with overlapping prefixes.




```
a large northern city ,                    45
a large northern city .                    66
a large northern constellation visible     41
a large northern country </S>              86
a large nose , a                          150
a large nose , and                        141
a large nose , but                        106
a large nose , raise                       64
a large nose , small                       44
a large nose and a                        160
```

Implementing tries as contiguous byte arrays



0	13	*offset of root node*
1	10	*node value of* ‘aa’
2	0	*size of index to child nodes of* ‘aa’ in bytes
3	3	*node value of* ‘ab’
4	0	*size of index to child nodes of* ‘ab’ in bytes
5	13	*node value of* ‘a’
6	4	*size of index to child nodes of ‘a’ in bytes*
7	a	*index key for* ‘aa’ *coming from* ‘a’
8	4	*relative offset of node* ‘aa’ $(5 - 4 = 1)$
9	b	*index key for* ‘ab’ *coming from* ‘a’
10	2	*relative offset of node* ‘ab’ $(5 - 2 = 3)$
11	7	*node value of* ‘b’
12	0	*size of index to child nodes of* ‘b’ in bytes
13	20	*root node value*
14	4	*size of index to child nodes of root in bytes*
15	a	*index key for* ‘a’ *coming from root*
16	8	*relative offset of node* ‘a’ $(13 - 8 = 5)$
17	b	*index key for* ‘b’ *coming from root*
18	2	*relative offset of node* ‘b’ $(13 - 2 = 11)$

Variable-length encoding of (unsigned) integers

- Represent number in base-128 ($\Rightarrow$ 7 bits per digit).
- Store each digit in a single byte.
- Use 8th bit in the byte to indicate whether more digits need to be read from the stream or file.

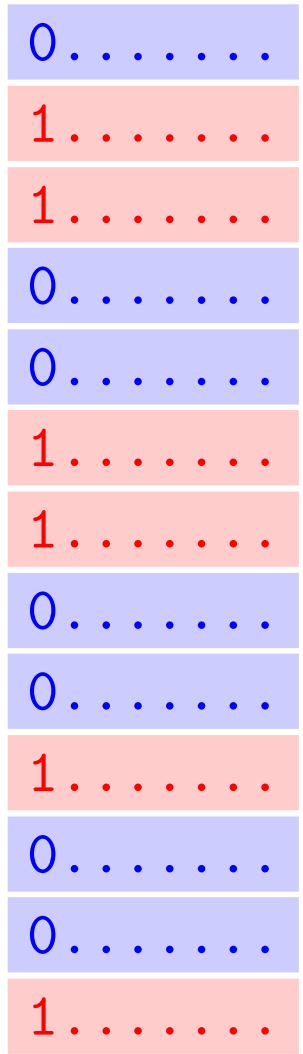
base-10	**base-128**	**bytes**
3	3	10000011
260	24	00000010 10000100
10584073	5609	00000101 00000110 00000000 10001001

$\Rightarrow$ Small numbers are better than large numbers!

- Assign token IDs in order of token frequency ($\Rightarrow$ lowest IDs occur the most often).
- Keep children close to their parent and use relative offsets.

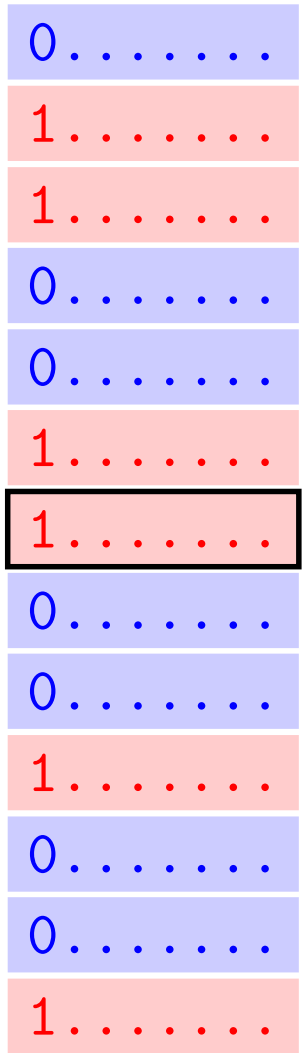
Binary search in compressed (= "tight") indices

- Use the flag bit to indicate for each byte whether it belongs to a key or an offset/value ($\Rightarrow$ stop reading when the flag bit changes or when you reach the end of the search range).



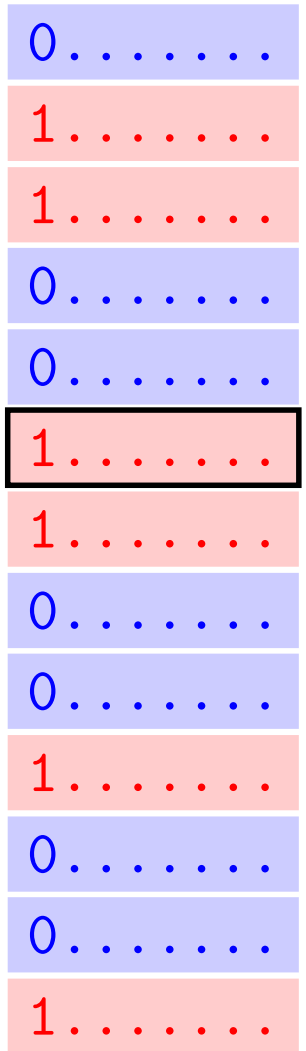
Binary search in compressed (= "tight") indices

- Use the flag bit to indicate for each byte whether it belongs to a key or an offset/value ($\Rightarrow$ stop reading when the flag bit changes or when you reach the end of the search range).
- Jump into the middle of the *byte* range that constitutes the search range.



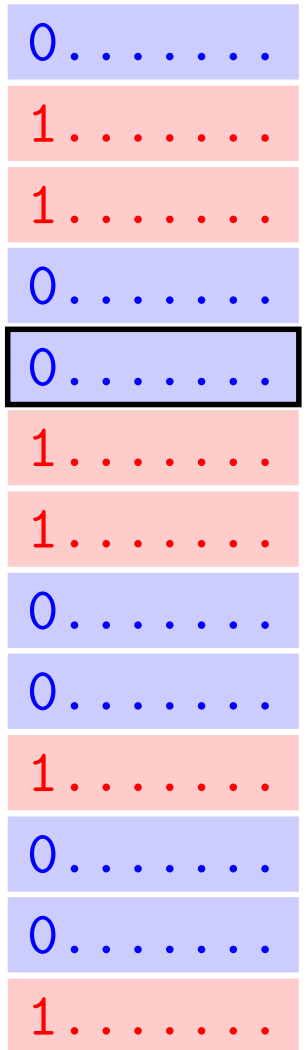
Binary search in compressed (= "tight") indices

- Use the flag bit to indicate for each byte whether it belongs to a key or an offset/value ($\Rightarrow$ stop reading when the flag bit changes or when you reach the end of the search range).
- Jump into the middle of the *byte* range that constitutes the search range.
- Linearly search backwards till you find the left edge of a key-value pair (flag bit changes from 0 to 1).



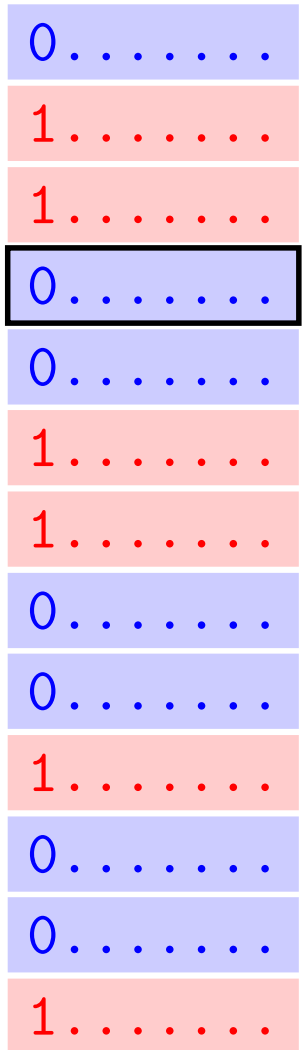
Binary search in compressed (= "tight") indices

- Use the flag bit to indicate for each byte whether it belongs to a key or an offset/value ($\Rightarrow$ stop reading when the flag bit changes or when you reach the end of the search range).
- Jump into the middle of the *byte* range that constitutes the search range.
- Linearly search backwards till you find the left edge of a key-value pair (flag bit changes from 0 to 1).



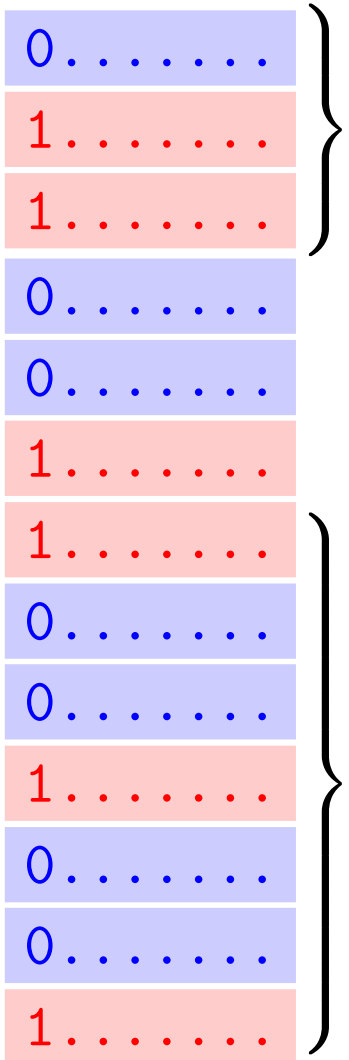
Binary search in compressed (= "tight") indices

- Use the flag bit to indicate for each byte whether it belongs to a key or an offset/value ($\Rightarrow$ stop reading when the flag bit changes or when you reach the end of the search range).
- Jump into the middle of the *byte* range that constitutes the search range.
- Linearly search backwards till you find the left edge of a key-value pair (flag bit changes from 0 to 1).



Binary search in compressed (= "tight") indices

- Use the flag bit to indicate for each byte whether it belongs to a key or an offset/value ($\Rightarrow$ stop reading when the flag bit changes or when you reach the end of the search range).
- Jump into the middle of the *byte* range that constitutes the search range.
- Linearly search backwards till you find the left edge of a key-value pair (flag bit changes from 0 to 1).
- Read key at this point and compare with search key.
- Return value if key == search key or recursively search the lower or upper half of the search range.



Writing unsigned integers

... in variable-width encoding with a stop flag:

```
void binwrite(ostream& out, uint64_t data);
```

... in variable-width encoding with a fixed flag:

```
void tightwrite(ostream& out, uint64_t data, bool flag);
```

... in fixed-width:

```
void numwrite(ostream& out, T& value);}
```

Reading unsigned integers

... from variable-width encoding with a stop flag:
returns pointer to byte after the number read.

```
char const* binread(char const* p, T& dest);
```

... from variable-width encoding with a fixed flag from memory range [p,q):
returns pointer to byte after the number read.

```
char const* tightread(char const* p, char const* q, T& dest);
```

... from fixed-width encoding:
returns pointer to byte after the number read.

```
char const* numread(char const* src, T& dest);
```

Additional tweaks

- Use an array of fixed-width fields to store the byte offsets of top-level nodes ($\Rightarrow \mathrm{O}(1)$ access at the root level).
- Use two binary flags per node to indicate
  - whether the node's value is distinct from a default value (don't store the value if it's the default)
  - whether the node is terminal or not (don't store size of index if terminal)

  Storing flags:
  - Shift each key value in the index two bits to the left.
  - Store the flags in the two freed least significant bits of the number.
  - Shift back prior to comparison during search.
- If node values can be represented as an integer, store them directly in the index instead of storing the offset of the node.

Overall File Structure

header	8 bytes	offset of start of top-level index from start of file
	4 bytes	size of the top level index
	variable	root value
	variable	default value for nodes that don't record a value
data	variable	contiguous byte sequence encoding the trie data
index	fixed	top-level index: array of $\langle$ `uint64_t`, `uchar`$\rangle$ pairs wher `uint64_t` = *file offset* and `uchar` = *flags*

A Node Entry

variable	tightly packed index of child nodes
→ variable	size of index in bytes (unless flag indicates it has no children)
variable	node value (if not default)

The TpTrie API

- template class with two template parameters
  - the value type T stored at each node
  - a functor vReader that reads a value from a char const*
- vReader must be a subclass of

```
template<typename T> class ValReaderBaseClass
{
public:
  virtual
  void
  operator()(istream& in, T& value) const = 0;
  // deprecated, will disappear

  virtual
  char const*
  operator()(char const* p, T& value) const = 0;
};
```

The TpTrie API: Typical Use

```
#include "tptrie.h"
#include "tpt_tokenindex.h"
TpTrie<uint32_t> T;
T.open("myfile");
TokenIndex V("vocabfile.tdx");
vector<id_type> key;
string line,w;
while (getline(cin,line))
{
  istringstream buf(line);
  buf >> w;
  TpTrie<uint32_t>::node_ptr_t n = T.find(V[w]);
  while (n != NULL && buf >> w)
    n = n->find(V[w]);
  if (n != NULL)
    cout << n->value() << endl;
  else
    cout << "[not found]" << endl
}
```

TpTrie<>::Node and TpTrie<>::Iterator

- **TpTrie<>::Node:** small record that contains all the vital information about a node in the trie (byte range of index, start position of value, etc.)
- **TpTrie<>::Iterator:**
  - maintains a vector of `boost::shared_ptr<TpTrie<T>::Node>` from the root of the structure to a particular node in the trie.
  - also maintains a vector of IDs (arc labels) leading to that node.
  - allows iteration over the trie via member functions bool down(), bool over() and bool up().

```
void dump(TpTrie<T>::Iterator& m, TokenIndex const& V) {
  cout << V.toString(m.ids) << " "
       << m.path.back()->value << endl;
  if (m.down()) {
    do { dump(m,V); } while (m.over());
    m.up();
  }
}
```

Encoding back-off language models

$$\mathrm{P}(w_i \mid w_{i-n+1}^{i-1}) = \begin{cases} \bar{\mathrm{P}}(w_i \mid w_{i-n+1}^{i-1}) & \text{if found} \\ \beta(w_{i-n+1}^{i-1}) \cdot \bar{\mathrm{P}}(w_i \mid w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases}$$

found $\bar{\mathrm{P}}(e \mid a\ \ b\ \ c\ \ d)$? $\xrightarrow{\text{yes}}$ *return* $\bar{\mathrm{P}}(e \mid a\ b\ c\ d)$

↓ no

found $\bar{\mathrm{P}}(e \mid \ \ b\ \ c\ \ d)$? $\xrightarrow{\text{yes}}$ *return* $\beta_{abcd}\bar{\mathrm{P}}(e \mid b\ c\ d)$

↓ no

found $\bar{\mathrm{P}}(e \mid \ \ \ \ c\ \ d)$? $\xrightarrow{\text{yes}}$ *return* $\beta_{bcd}\beta_{abcd}\bar{\mathrm{P}}(e \mid c\ d)$

↓ no

found $\bar{\mathrm{P}}(e \mid \ \ \ \ \ \ d)$? $\xrightarrow{\text{yes}}$ *return* $\beta_{cd}\beta_{bcd}\beta_{abcd}\bar{\mathrm{P}}(e \mid d)$

↓ no

found $\bar{\mathrm{P}}(e \ \ \ \ \ \ \ \ )$? $\xrightarrow{\text{yes}}$ *return* $\beta_d\beta_{cd}\beta_{bcd}\beta_{abcd}\bar{\mathrm{P}}(e)$

↓ no

return β_0

Encoding back-off language models

- Use ‘backwards’ contexts for indexing.
- Use codebook for back-off weights and probability valuesmost frequent scores get lowest IDs.
- Node value:
  - ID of the back-off weight for the context.
  - List of possible successor words and IDs of probability valuesstored as compressed index.

Encoding phrase tables

Node values: lists of target language phrases with associated scores.

- Store scores via a code book.
- Store phrases via pointers into a *bottom-up trie:*
  Each node in a bottom-up trie stores
  - a token ID;
  - a pointer to the parent node.

  Phrases can be represented by pointing (via byte offset) to a node in the bottom-up trie.
- Instead of using fixed 7-bit blocks for variable-length encoding of code book IDs and phrase IDs, we adapt the scheme to use bit blocks of optimal size. E.g.: read 5 bits, if flag bit is set, read another 7 bits, etc.

Building TpTries

```
...
uint64_t startIndex=0;
uint32_t idxSize=0;
ValWriter<T> write;
T defaultValue=...;
T rootValue=...;
vector<pair<uint64_t,uchar>> index;
ofstream out("blabla.tpt");

// write the file header
numwrite(out,startIndex); // place holder, will be overwritten at the end
numwrite(out,idxSize);    // place holder, will be overwritten at the end
write(out,rootValue);
write(out,defaultValue);

// write the data block
for (size_t i = 0; i < numTypes; ++i) {
  index.push_back(processNode(TreeIterator(topLevelNode(i))));
}
startIndex = out.tellg();
idxSize = index.size();

// write the index
for (size_t i = 0; i < index.size(); ++i) {
  numwrite(out,index[i].first);
  out.put(index[i].second);
}

// update the file header
out.seek(0);
numwrite(out,startIndex);
numwrite(out,indexSize); // carefull, must be of type uint32_t!
out.close();
```

Building TpTries

```
pair<uint64_t,uchar>
processNode(TreeIterator& m) {
  map<uint32_t,uint64_t> index;
  if (m.down()) {
    do {
      pair<uint64_t,uchar> foo = processNode(m);
      // get the ID of the node currently at the end of m
      uint32_t id = lastID(m);
      id = (id<<FLAGBITS)+foo.second;
      index[id] = foo.first;
    } while (m.over());
    m.up();
  }
  uint64_t startIdx = write_tp_index(out,index);
  uint64_t myPosition = out.tellp();
  uchar flags= index.size() ? HAS_CHILD_MASK : 0;
  if (flags) {
    binwrite(out,myPosition-startIdx);
    if (lastValue(m) != defaultValue) {
      write(out,lastValue(m));
      flags += HAS_VALUE_MASK;
    }
  else {
#if value_can_be_represented_as_integer
    myPosition = toInteger(lastValue(m));
#else
    if (lastValue(m) != defaultValue) {
      write(out,lastValue(m));
      flags += HAS_VALUE_MASK;
    }
#endif
  }
  return pair<uint64_t,uchar>(myPosition,flags);
}
```

Part III

Suffix Arrays

Corpus Tracks

Memory-mapped file with

8 bytes	offset of index start from start of file
4 bytes	number of sentences in corpus
4 bytes	number of tokens in corpus
	data block: all tokens in corpus in linear order in one big array
	index: maps from sentence number to start position in corpus

Interface

```
Token const* sntStart(uint32_t sid) const;
Token const* sntEnd(uint32_t sid) const;
size_t size() const;          // = number of sentences
size_t numTokens() const; // = number of tokens in corpus
void open(string fname);  // open a file
```

Suffix Arrays

Memory-mapped file with

8 bytes	offset of index start from start of file
4 bytes	vocab size
	data block: sorted list of ⟨sentence id, offset⟩ pairs written as tight index for compactness
	index: maps from token ID to byte range in the data block that corresponds to all suffixes starting with that token

Low-level Interface

```
char const* lower_bound(Token* startKey, Token* endKey) const;
char const* upper_bound(Token* startKey, Token* endKey) const;
```

return the positions in the data block corresponding to the Token positions at the first and just beyond the last occurrence of [startKey,endKey).

Suffix Arrays: Interface

- also provides tree-like iterator
- functions to fill dynamic_bitsets or perform counts, given lower and upper bounds
- function for creating a unique phrase ID (e.g., for mapping from phrases to associated information in a way that requires little memory)
- currently working on suffix arrays that sort by upward dependency chains in a dependency-parsed corpus
- **user beware:** currently converting Corpus Track and Suffix Array implementations into template classes.