EECS 4408 System Design of a Search Engine

Winter 2021

Lecture 10: The frontier, hashing and the index

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Course details.
2. RAII.
3. The frontier.
4. Hashing.
5. The index.

Agenda

1. Course details.
2. RAII.
3. The frontier.
4. Hashing
5. The index.

details

1. Two-day extensions on the HtmlParser HW available on request.
2. The Hashing homeworks published. These are critical to understanding how you will build your inverted word index as a huge memory-mapped data structure and hash table on disk.
3. At this point, you should be able to:
   a. Read pages over HTTP and HTTPS.
   b. Extract words and links. Not crashing is more important than getting every page, word or link.
4. Most teams are probably not yet dealing with redirects, chunking or gzip content, the frontier, robots.txt, or page content (e.g., blacklisting porn).
5. I am reading your plans now and will open meeting slots RSN.
6. Today starts the shift from OS stuff to search engine internals. Will return later to do the web server.

LinuxGetUrl/Ssl

The HW ask that you strip off the entire HTTP header (different than the version I demo).

Can you assume that if you do a recv() with a large buffer, that the \r\n\r\n marking the blank line marking the end of the header will be there?

Can you assume that those 4 characters will appear contiguously in single recv()? Could they be split across multiple recv( ) calls?

Why is that?

recv( ) can only return whatever's arrived. The server can write data in whatever size chunks it likes (a socket looks like a stream to it also) and the data is being packetized and possibly sent over different paths on the internet.

As a great philosopher once observed: You can't always get what you want. (But if you try sometimes, well, you might find you get what you need.)

So, how do you solve this?

Simple state machine

```
const char *endofHeader = "\r\n\r\n";
const char *p, *nextmatch = endofHeader;
bool skipping = true;
while ( ( bytes = recv( s, buffer, sizeof( buffer ), 0 ) ) > 0 )
   if ( skipping )
      {
      for ( p = buffer;  p < buffer + bytes;  p++ )
         if ( *p == *nextmatch )
            {
            // Advance to the next char to match.
            // If at the end, stop skipping and
            // write out the rest of the buffer.
            }
         else
            // start over if not a match.
            nextmatch = endofHeader;
      }
   else
      write( 1, buffer, bytes );
```

Agenda

1. Course details.
2. RAII.
3. The frontier.
4. Hashing
5. The index.

RAII

RAII = Resource acquisition is initialization.

Basic idea: Define a class with a constructor that acquires any lock you need. The destructor frees the lock.

Takes advantage of the C++ guarantee that the destructor will always run, even if a C++ exception is thrown that causes the block to exit.

```
#include <pthread.h>
#include <cassert>

class CriticalSection
   {
   private:
      bool locked;
      pthread_mutex_t *mutex;

   public:

      // Take the lock.
      void Take( )
         {
         assert( !locked );
         pthread_mutex_lock( mutex );
         locked = true;
         }

      // Release the lock.
      void Release( )
         {
         assert( locked );
         pthread_mutex_unlock( mutex );
         locked = false;
         }

      // Constructor takes the lock.
      CriticalSection( pthread_mutex_t *mutex ) :
            locked( false ), mutex( mutex )
         {
         Take( );
         }

      // Destructor releases the lock.
      ~CriticalSection( pthread_mutex_t *mutex )
         {
         if ( locked )
            Release( );
         }

};


      // Usage.
      // Create the mutex.
      pthread_mutex_t mutex;
      pthread_mutex_init( &mutex, nullptr );

      // Lock when needed.
      if ( needLock )
         {   CriticalSection lock( &mutex );
         // do something
         :
         // Lock released automatically
         }
```

Agenda

1. Course details.
2. RAII.
3. The frontier.
4. Hashing.
5. The index.

Crawler

1. Manage a frontier of new links to be crawled.
2. Decide what will or will not be crawled and in what order.
3. Keep track of what's already been crawled.
4. Read pages over HTTP and HTTPS.
5. Obey robots.txt files.
6. Deal with redirects.

All of this has to be highly multithreaded so you don't wait on slow sites, instead overlapping them.

You may also want to spread it across multiple machines.

Crawler

Typically maintains pool of worker processes or threads to read and parse webpages.

Each worker retrieves the file and queues it for the HTML parser which creates an object.

The resulting stream of objects are then passed to the Index builder.

The frontier

1. Basic problem: A very large list of URLs that you have not yet crawled. What should you crawl next?
2. Some links are obviously better than others, e.g., .edu vs. .biz.
3. You don't want to DOS anyone.
4. Some pages are prohibited by robots.txt, meaning you will need to cache this for each domain.
5. If you are crawling on multiple machines, you will need to decide how to split up the work to avoid duplicate crawling but combine the results.
6. Just because you crawled a page doesn't mean you want to index it or follow any of its links. You may want to blacklist it.

Managing the frontier

1. Should crawling be a depth or breadth-first search of the web?
2. How would you decide whether to add a link to frontier in the first place?
3. How would you decide which links already on the frontier should be crawled next?
4. What do you need to cache per domain?
5. Will you have a blacklist?
6. Will you do language or porn detection? How?

The frontier

Often described as a priority queue problem.

But you have a choice when to do the priority calculations.

You can do the calculation as you add it your queue or as you pull things off.

But what happens if you go in strict priority order?

Mercator

The Mercator system in 2001 by Marc Najork and Allan Heydon used a complex system of queues.

http://www.cs.cornell.edu/courses/cs685/2002fa/mercator.pdf

(Please read this article.)

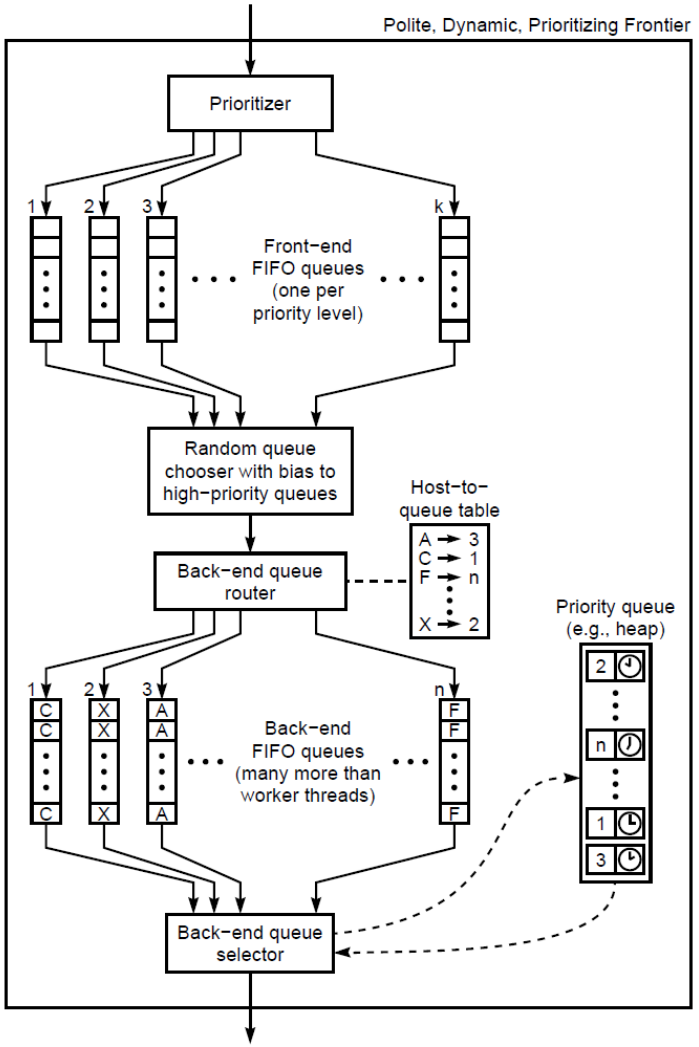


Figure 3: Our best URL frontier implementation

A simpler strategy

Assume there are a very large number of links on your frontier.

1. Randomly choose n links and rank them using whatever information you have available.
2. Crawl the top k links.
3. Repeat.

Distributing your crawler

If you have n crawlers, consider a hash(URL or domain) % n to decide what machine should crawl it.

If a given machine is crawling a URL, it should probably index it.

When a query is presented, it should be broadcast to all your machines, which respond with results, that are merged before reporting.

Agenda

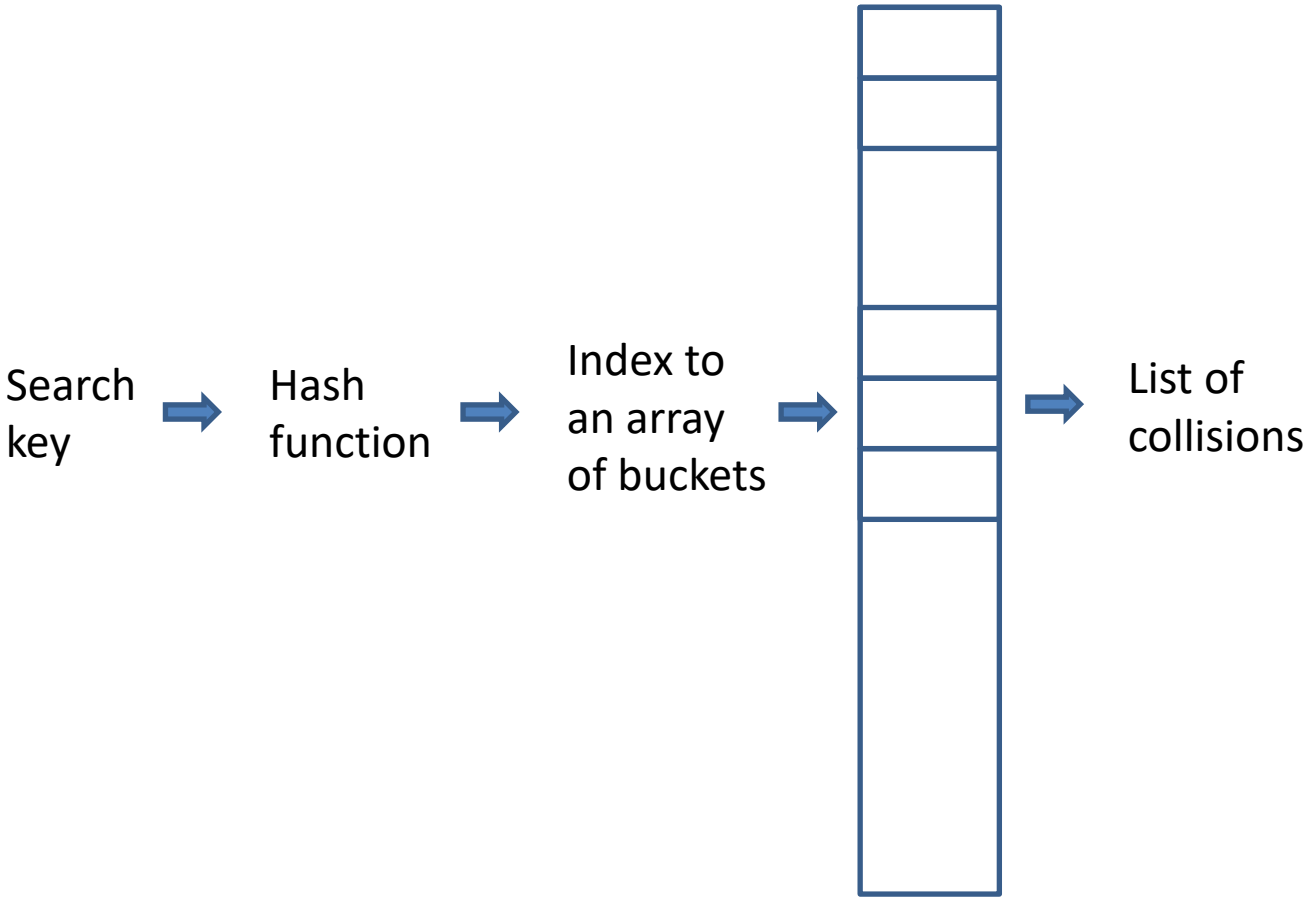
1. Course details.
2. RAII.
3. The frontier.
4. Hashing.
5. The index.

Hashing

Create an array of buckets to hold the items you wish to index based on a key field.

Create a simple function that can map the key to a bucket index.

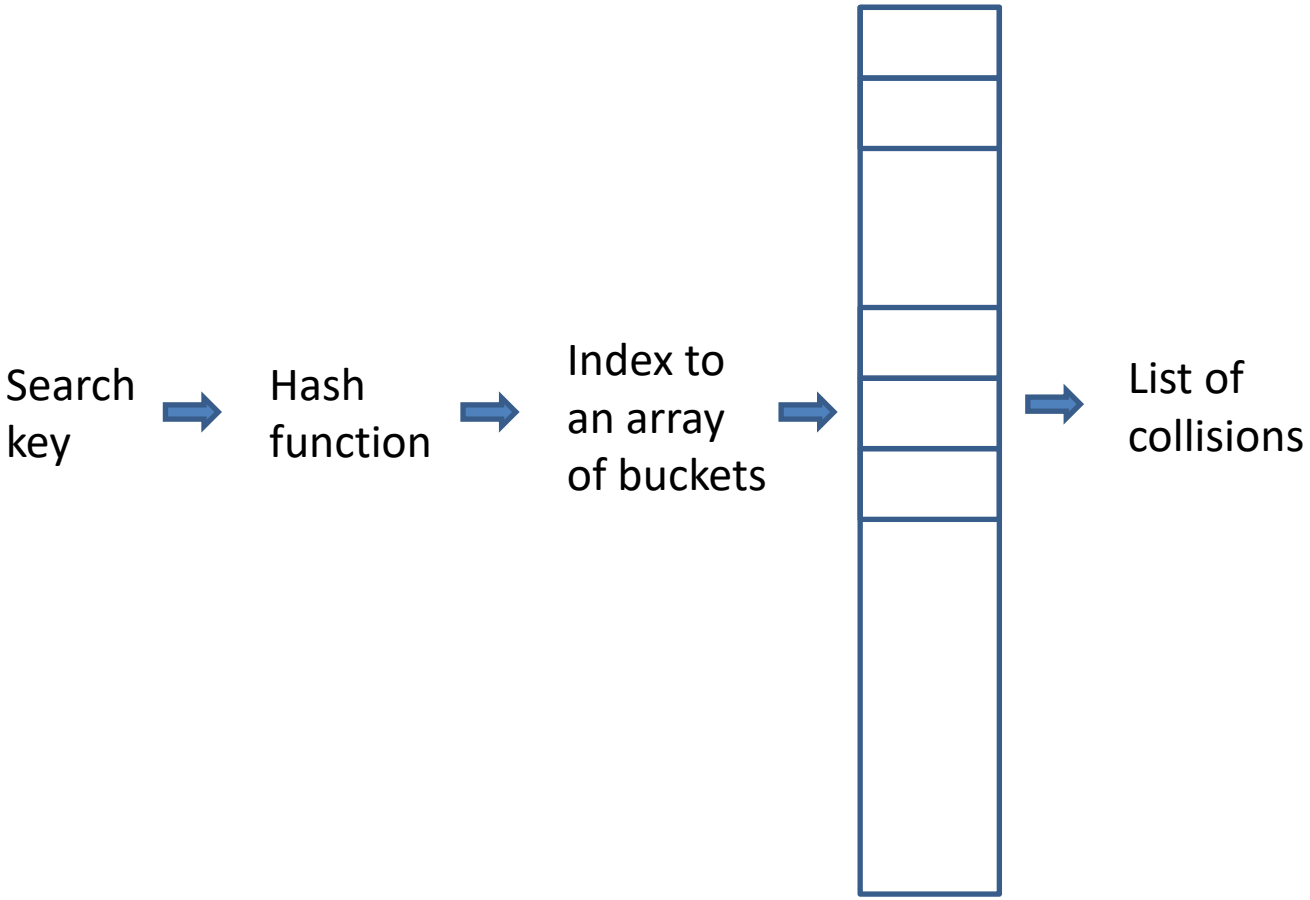
If multiple different keys map to the same bucket, that's called a collision.



Hashing

Expensive to grow the table or deal with collisions, so we make the table pretty big.

No serious developer would use hashing except when they expect *lots* of entries that don't have to be kept in sort order, so there's no point in starting with only two buckets.

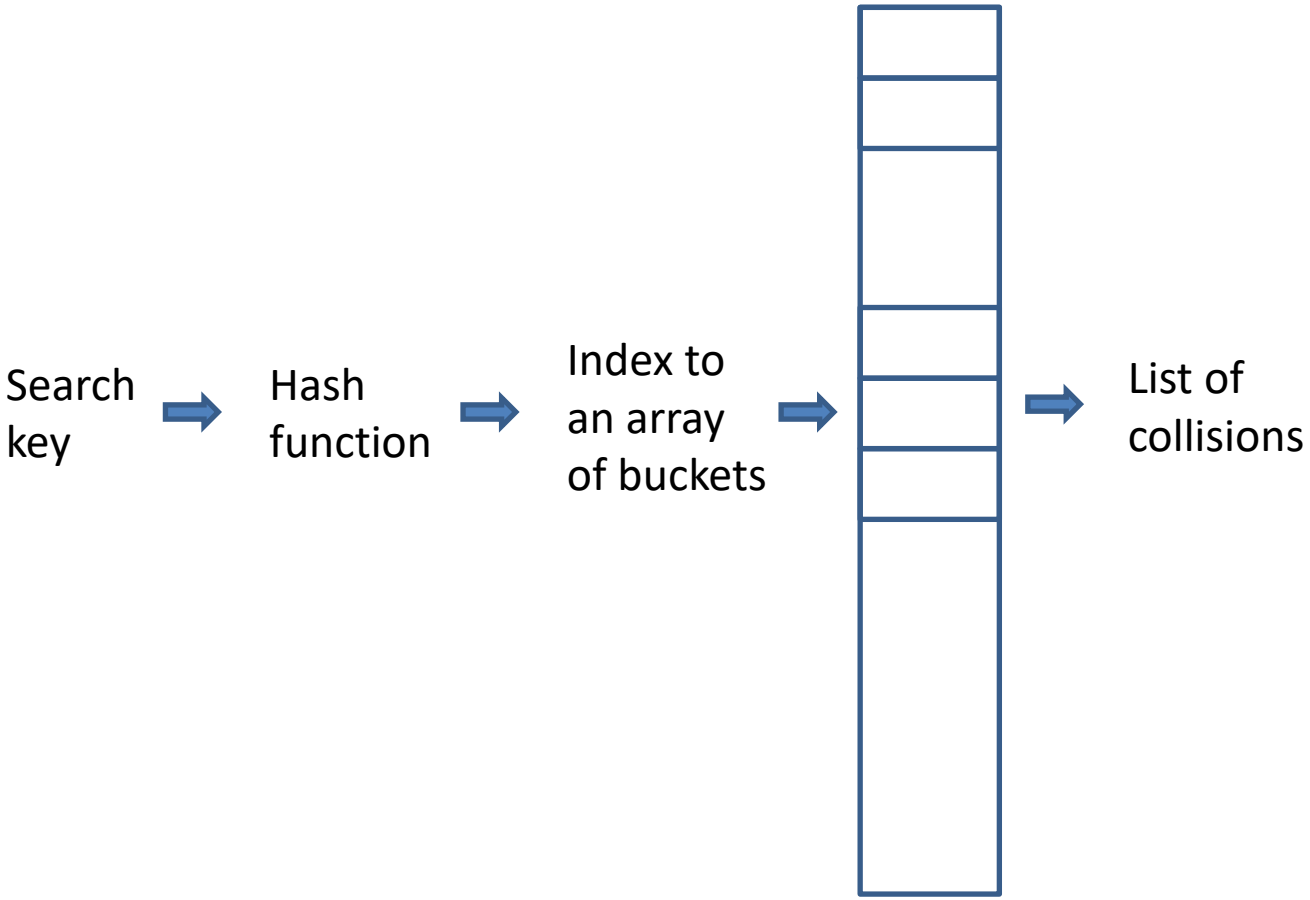


Hashing

Start with 1024 or 4096 or some other big power-of-two number of buckets that you expect matches the problem.

Use the low bits of the hash function as an index.

You may/may not support growing the hash table.



Hash function

Even a simple hash can work pretty well.

But what's the problem with this one?

```
unsigned Hash(  const char *p )
   {
   unsigned h = 0;
   for ( ;  *p;  p++ )
      h = ( h << 1 ) ^ *p;
   return h;
   }
```

Fowler-Noll-Vo hash function

A more complex hash using big prime numbers.

```
size_t fnvHash ( const char *data, size_t length )
   {
   static const size_t FnvOffsetBasis =
         14695981039346656O
   static const size_t FnvPrime =
         1099511628211ul;

   size_t hash = FnvOffsetBasis;
   for ( size_t i = 0; i < length; ++i )
      {
      hash *= FnvPrime;
      hash ^= data[ i ];
      }
   return hash;
   }
```

Polynomial function

CRCs (cyclic redundancy codes) use shift registers.

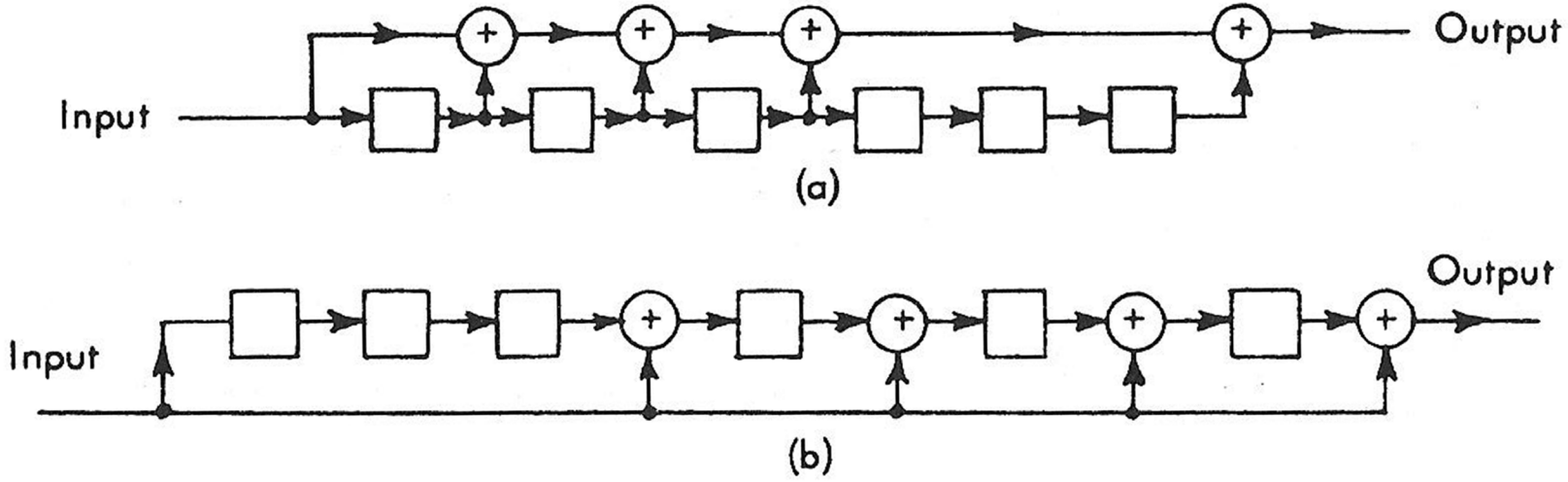


Figure 7.5. Circuits for multiplying by $X^6 + X^5 + X^4 + X^3 + 1$.

Source: W. Wesley Peterson and E.J. Weldon, Jr, *Error-Correcting Codes*, Second Edition, MIT Press, 1972, pp 178-179.

Polynomial function

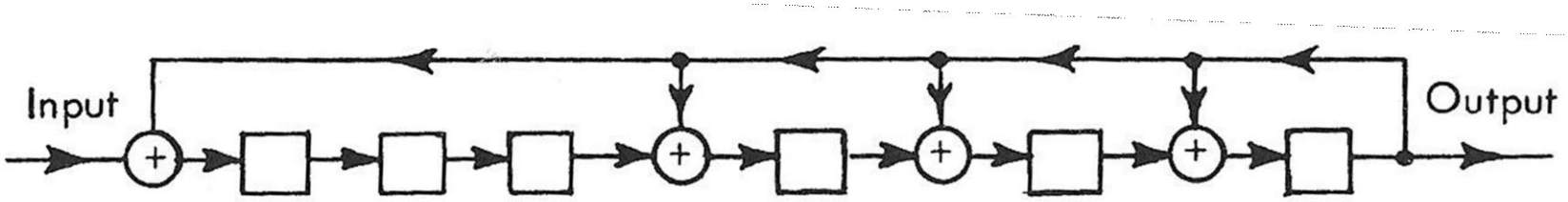


Figure 7.7. A circuit for dividing by $X^6 + X^5 + X^4 + X^3 + 1$.

```
/*
   Ethernet CRC
   Polynomial   x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 +
                      x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1

   Represent the polynomial as an unsigned number with each bit corresponding
   to a term of the polynomial set to one.  E.g., the x^5 term means bit 5
   (counting from the LSB as bit 0) is on.  The x^32 term is actually the
   feedback term and is "off the end" of the polynomial.
*/

#define  Ethernet_Polynomial     0x04c11db7
#define  Poly                    Ethernet_Polynomial
```

```
/* Calculate the CRC table entries.  crc(x) returns table entry x.  Use this
   routine to fill a table crctab[256] with the appropriate values.
*/

ulong crc( register ulong i )
      {
      i <<= 24;

      i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
      i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
      i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
      i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
      i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
      i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
      i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
      i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);

      return i;
      }
```

```
static const unsigned long CRCTable[ ] =
      {  0x00000000, 0x04c11db7, 0x09823b6e, 0x0d4326d9,
         0x130476dc, 0x17c56b6b, 0x1a864db2, 0x1e475005,
         0x2608edb8, 0x22c9f00f, 0x2f8ad6d6, 0x2b4bcb61,
         0x350c9b64, 0x31cd86d3, 0x3c8ea00a, 0x384fbdbd,
         0x4c11db70, 0x48d0c6c7, 0x4593e01e, 0x4152fda9,
         0x5f15adac, 0x5bd4b01b, 0x569796c2, 0x52568b75,
         0x6a1936c8, 0x6ed82b7f, 0x639b0da6, 0x675a1011,
         . . .
         0xbcb4666d, 0xb8757bda, 0xb5365d03, 0xb1f740b4   };
```

```
// To calculate a CRC, use the UpdateCRC( crc, c ) procedure to
// accumulate the data into the CRC.  The crc argument is the
// accumulated CRC, c is the next character of data to be added.
// Initially, crc = 0.

unsigned long UpdateCRC( unsigned long crc, unsigned char c )
   {
   return ( ( ( crc ) << 8 ) ^ CRCTable[ ( ( crc ) >> 24 ) ^ ( c ) ] );
   }
```

Buckets

Can either contain the object being indexed or a pointer to the object.

If it’s the object itself, you need at least as many buckets as objects and each bucket will be larger.

If it’s a pointer, it can be to a list.

Collisions

You have to assume that you will have collisions.

When searching a hash bucket, you must compare the search keys to make sure that what you found is what you wanted.

If the key fields are long, you might decide to add a field to a bucket entry with the complete hash of the key and compare that before comparing the entire field.

Collisions

Several ways to deal with collisions.

1. Increment the bucket number modulo the number of buckets and try there.
2. Use a second hash (or just different bits of the same hash) and try there.
3. Create a list off each bucket.
4. Try (1) or (2) and if not successful, create list.

Collisions

When using a list for collisions, the insertion order matters.

The STL map template pushes collisions onto the front of the list.

Why did they do that and is that a good choice?

Collisions

If query performance is more important than build time, collisions should be placed onto the end.

Why?

Collisions

If query performance is more important than build time, collisions should be placed onto the end.

Why?

Because you expect to encounter the most frequent terms earlier in the build and you want those at the front.

Collisions

If you use a list for collisions, most people normally think of something like a linked list.

Since you don’t know what will have to go into the list, one way to deal with collisions by pushing new node onto the list (or onto the end).

Collisions

But suppose you already know what the data looks like.

Could you use that information to build a better performant, smaller memory footprint, more efficient hash table to hold exactly that known data?

If you know the data

You might try several hash algorithms to see which gives you the lowest rate of collisions.

You could tune the number of buckets.

You could compress the collision lists. Instead of a linked list, it could be an array or a serialized stream.

Compressing the list

Buckets contain pointers or, even better, offsets into one large contiguous buffer containing the items stored in the hash table.

Each list is a contiguous series of bytes representing a concatenation of the entries on a collision list.

If you only need to read them serially, no problem if the entries vary in size, as long as you can tell them apart.

You do need to know when you've reached the end, typically with a sentinel.

Agenda

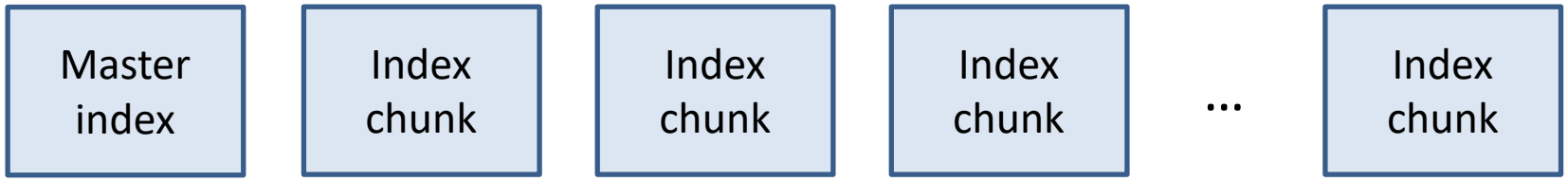
1. Course details.
2. RAII.
3. The frontier.
4. Hashing.
5. The index.

Index

Basic problem: Create a merged inverted word index of all the documents that have been crawled, allowing you to report all the documents and individual locations (postings) where any given word was found.

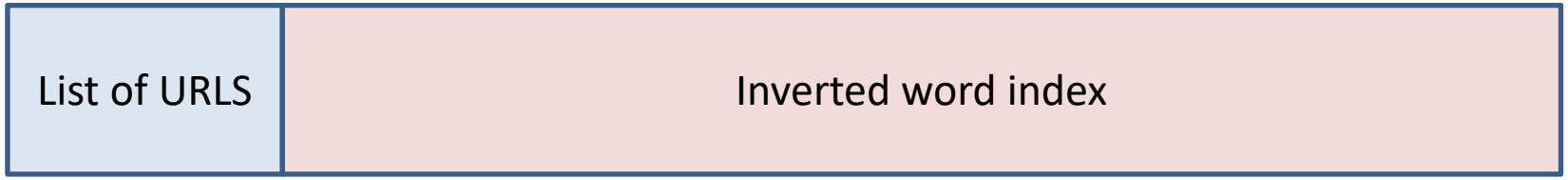
Due to the size, the posting lists will have to be on disk. For performance, you'll need to keep some directory and indexing information in memory.

A search engine index is typically a set of files

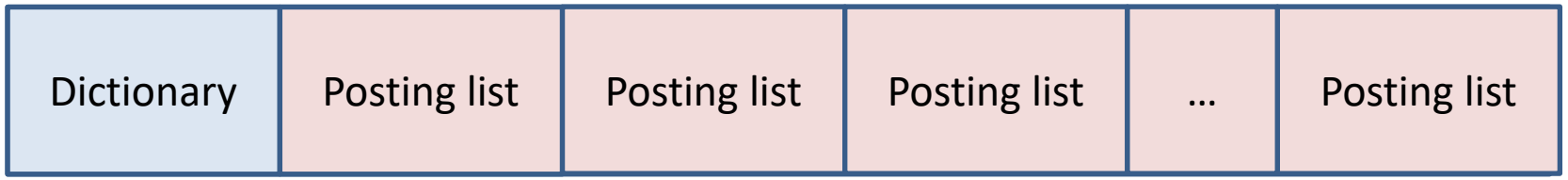


A master index

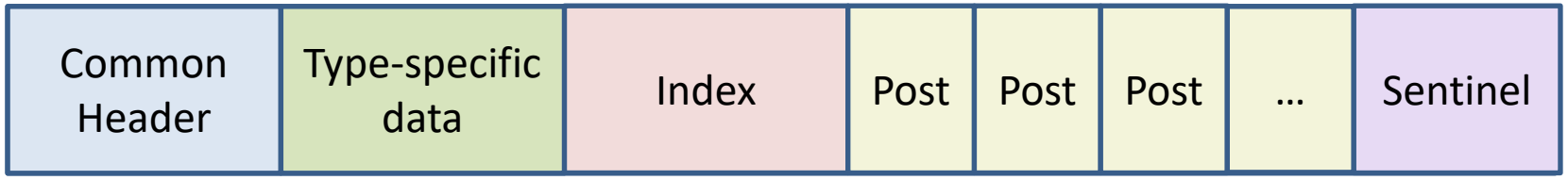
Each index chunk



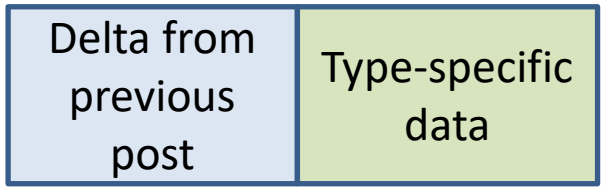
The inverted word index



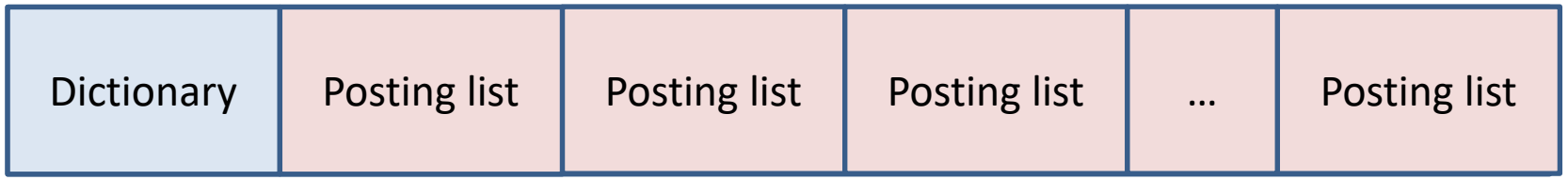
A posting list



An individual post



The inverted word index file format



Dictionary contains:

1. Number of tokens in the index.
2. Number of unique tokens in the index.
3. Number of documents in the index.
4. Hash table to translate from token to offset to the posting list.

Tokens can be decorated to distinguish words in the title vs. the body or URL, etc., and to create special tokens, e.g., end-of-document.

Index functions

Index stream readers (ISRs)

first(t)	returns the first position at which t occurs.
last(t)	returns the last position at which t occurs.
next(t, current)	returns the next position where t occurs after the current position.
prev(t, current)	returns the last position where t occurs before the current position.

Numbering locations

You have a choice whether to number locations relative to:

Start of the document

Start of the index

If you go with *start of document*, individual word locations will be *(docid, offset)* and you probably use a *separate index of documents*.

If you go with *start of index*, a word location is simply an *offset* and you probably enter *document boundaries as postings*.

Dictionary

Words are usually case-folded.

Special characters and numbers often discarded.

May do stemming, lemmatization and stop word elimination.

May special case certain terms, e.g., C++.

May have to word-break in some languages.

Stemming and Lemmatization

Either of these is a strategy for attempting facilitate finding close matches where the words have similar meaning.

Stemming is an algorithmic process of replacing words with simpler forms, e.g., with production rules to discard prefixes or suffixes. Swim, swimmer, swimmers, swimming all reduced to swim. Most famous is the Porter stemmer.

Lemmatization is a dictionary-based process for replacing words with their root or lemma. Better becomes good, walking becomes walk.

Dictionary

May have multiple kinds of things in the dictionary, e.g., document boundaries vs. words.

Each type of post may have *attributes*, for example:

word	Bold, heading, large font.
document	URL, number of word or unique words in the URL, title, body, anchor.

May also distinguish variations on word, e.g., only in the URL vs. only in the title, by *decorating* the word when entering it into the dictionary.

Posting list

1. Huge.
2. Important to reduce space.
3. Usual strategy is to encode each new location as a delta from the previous.
4. Further encode with varying numbers of bits depending on the delta, e.g., as utf-8 .
5. Some number of bits may encode attributes, e.g., bold, italic.
6. Synchronization points allow seeking to a location just prior to desired location, then scanning forward.

Things to decide

In addition to the posting list, what information will you have for each entry?

1. Number of occurrences in the corpus
2. Number of documents containing this word

What information will you keep for each index?

1. Number of documents in the corpus
2. Total number of words
3. Total number of unique words

What kinds of posts will you have and what information will each contain?

What attributes or decorations will you use?

How will you encode the location numbers?

Will you have synchronization points?

Decorating

Add characters that get stripped out during HTML parsing to indicate special characteristics or types of posts, e.g.,

amazon	amazon in the body text
#amazon	amazon only in the URL
@amazon	amazon only in the title
$amazon	amazon only in the anchor text
%	End-of-document token.

Might also be used for *stemming*:

swim*	swim, swims, swimming, etc.

A posting list

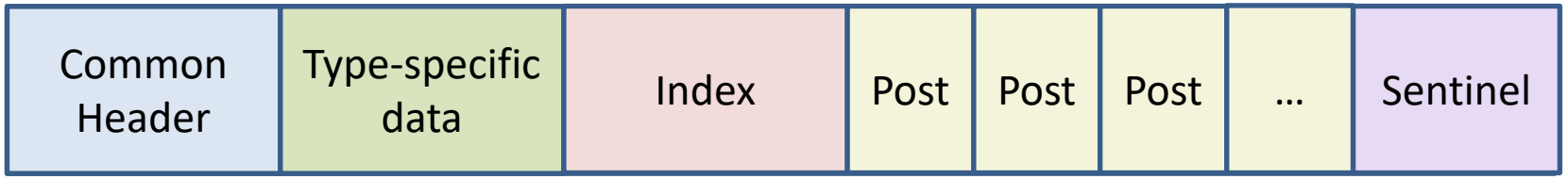
Common header contains:

1. Number of occurrences of this token in the index.
2. Number of documents in which this token occurs.
3. Type of token: end-of-document, word in anchor, URL, title or body.

For an end-of-document list, type-specific data might include:

1. Lengths of the document, URL and title + URL.
2. Amount of anchor text, number of unique words.
3. Any additional static rank information, e.g., date, number of links pointing to the page, etc.

A posting list



Posting list index

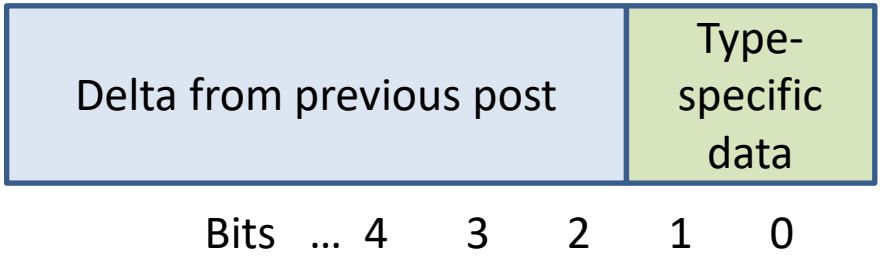
Seek location

	Synchronization points	
High bits of seek location	Seek offset in the postings	Actual location of that post.
0000 0000	0	32
0000 0001	531	20142
0000 0010	2012	912348
:	:	:

Delta from previous post

The offset will typically be encoded with a variable length scheme like UTF-8.

If only a few bits of type-specific information are needed, they can be encoded into the low bits of the UTF-8 character.



00 Normal
01 Italic
10 Bold
11 Heading

Anchor text tends to duplicate, with many links to the same page with the same anchor text.

For a word in anchor text, it can be useful to sort the phrases, retaining only the unique phrases but with counts on the words.

Because the counts can be so large, it can be helpful to shrink the number of bits required with the log function.