EECS 440 System Design of a Search Engine

Winter 2021

Lecture 11: The index and constraint solver

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

Agenda

1. Course details.
2. The index.
3. The constraint solver.

Agenda

1. Course details.
2. CRCs.
3. The index.
4. The constraint solver.

details

1. Thank you all for meeting with me over the weekend. I enjoyed our meetings and I hope you found them helpful.
2. Every team appears to be pretty much exactly where you should be. Most of you appeared to like and enjoy working with your teammates, which is a huge factor in success.
3. Most discussions were about how to split up your engine across multiple AWS machines multiple identical instances of an entire search engine that only crawled and indexed a slice of the web. You should be able to run and debug one of these on your laptop, crawling and indexing perhaps a few hundred pages.

details

Suggested plan for most teams:

1. Get your crawlers finished, including robots.txt and a crash-resistant way of managing the frontier.
2. Start work on the hashing homeworks to learn how to build the index.
3. Begin working on defining some C++ interfaces to the remaining parts, e.g., you'll pass an object with a set of methods even if you're not yet sure how they'll work.
4. Read my paper on paper on ranking.

midterm

Sorry, we are only beginning work on it so I nothing to report except that it will all short answer, mostly asking about concepts. If I think something's important enough to put on an exam, I probably thought it was important enough to talk about in lecture. I don't release previous and I ask that you be on your honor not to seek them out.

Agenda

1. Course details.
2. The index.
3. The constraint solver.

Index

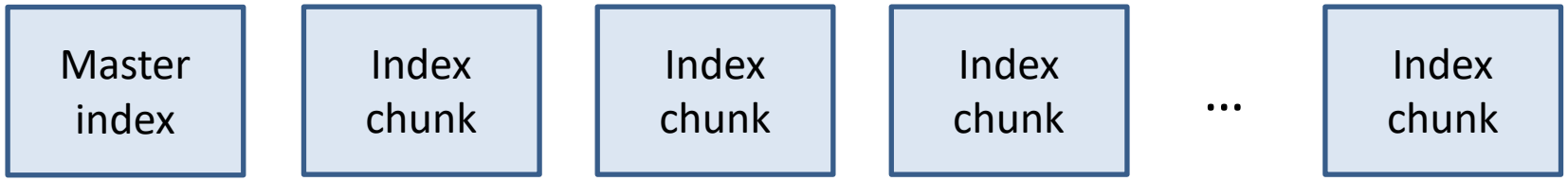
Basic problem: Create an inverted word index of all the documents that have been crawled, allowing you to report all the documents and individual locations (postings) where any given word was found.

Due to the size, it will all have to be on disk but memory-mapped into your address space. You'll depend on the virtual memory manager to keep heavily-referenced sections, e.g., the dictionary at the front in memory. But the bottleneck will be disk access.

Search time will grow linearly with the size of the index because every matching page will have to found and scored unless you short-circuit, e.g., by arbitrarily cutting it off after the first 20,000 matches to avoid getting trapped by searches for "a" or "the".

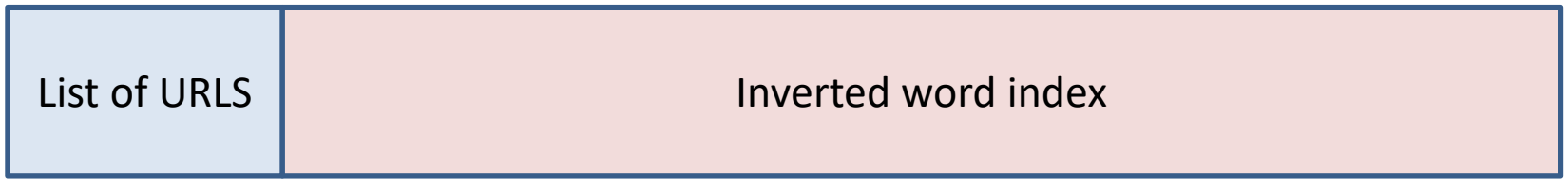
Most of size will be in the posting lists, so if the posting lists can be compressed, your engine will be faster.

A search engine index is typically a set of files

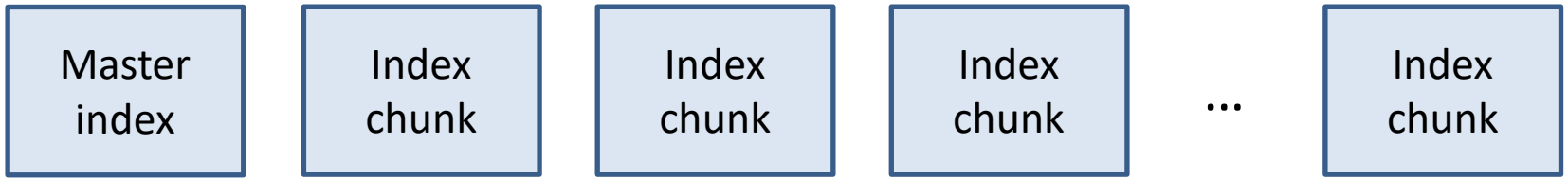


The master index might itself be a set of files capturing the overall state of the search engine.

Each index chunk



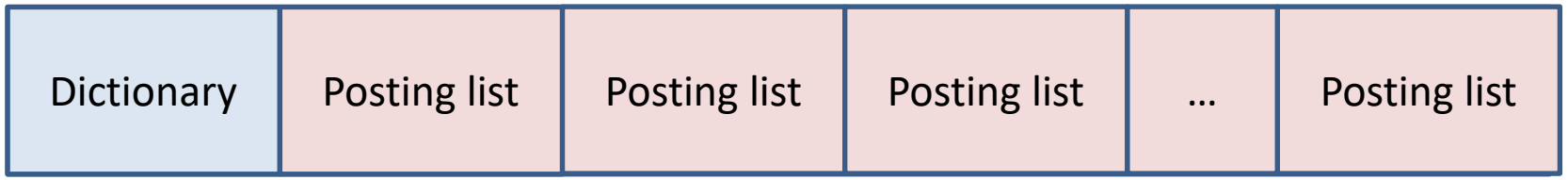
A search engine index is typically a set of files



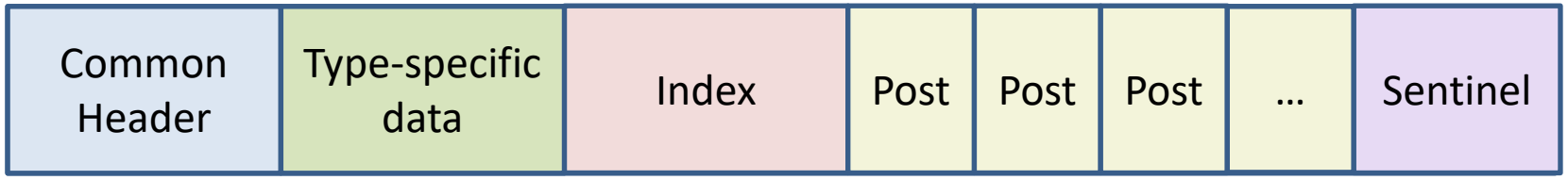
Typically, as pages are crawled and parsed, they're added to the next index chunk under construction. Once a chunk is full, you start filling a new chunk. Don't make them too small nor too big, but there's a huge in-between. Common choices might be a few hundred megabytes.

When a query comes in, each chunk must be searched, either sequentially or in parallel with separate threads.

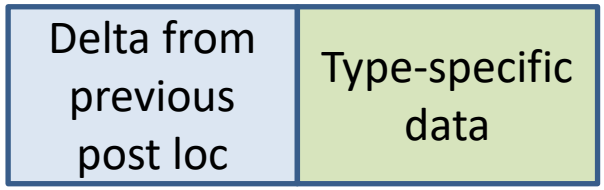
The inverted word index within a chunk.



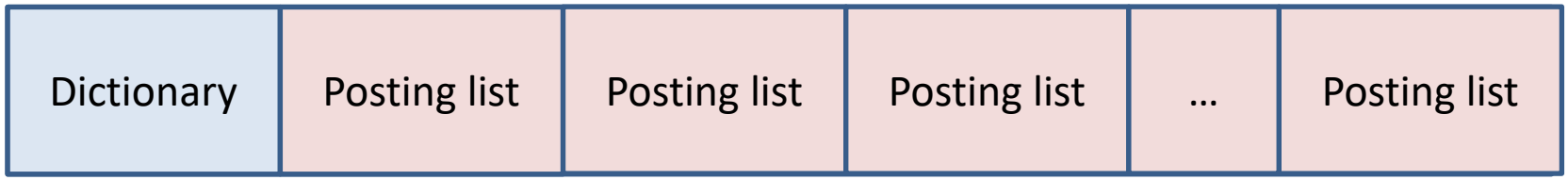
A posting list



An individual post



The inverted word index file format

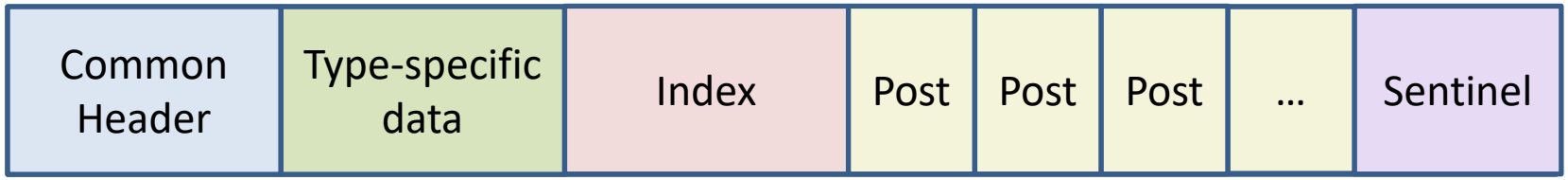


Dictionary contains:

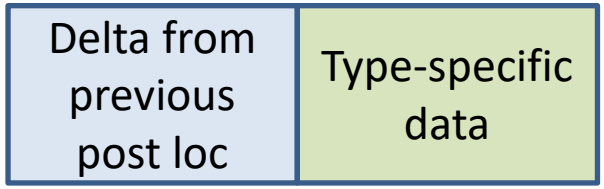
1. Number of tokens in the index.
2. Number of unique tokens in the index.
3. Number of documents in the index.
4. Hash table to translate from token to offset to the posting list.
5. Since collisions are possible, it's possible you might have to skip over the collisions, likely pasted one after another.

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.

A posting list



An individual post



It's important to keep the posts as small as possible on average by giving the location as a delta from the previous post and then encoding that delta in a variable number of bytes, e.g., as UTF-8.

Most posting lists will be of actual words found on a page. But there will be at least one other important posting list of document ends, marking the boundaries between documents pasted one after another in the index.

All the posts on a given list likely have the same type-specific data but the amount of data and number of bytes might vary and be whatever you like. For a word, the type-specific information might be title or bold. For a document end, it might be the URL (or an index into a table of URLs), the document title, document statistics, whatever you want.

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. *But slow and usually omitted.*

Numbering locations

You have a choice whether to number locations relative to:

1. Start of the document or
2. Start of the index

If you go with *start of document*, individual word locations will be *(docid, offset).*

If you go with *start of index*, a word location is simply an *offset,* and the deltas between locations will likely to be a smaller binary number. You will probably enter *document boundaries as postings*.

Dictionary

Words are usually case-folded.

Special characters and numbers are often discarded but perhaps one should only discard outer punctuation.

May do stemming, lemmatization and/or stop word elimination but it's unclear how helpful this is. Tends to increase the number of matches but recall isn't as important for a search engine as precision.

At Microsoft, we discarded punctuation but then needed to special case certain terms, e.g., C++. We used dictionary-based word-breaking on URLs and with documents and queries in Japanese, which happened to be an important market for Microsoft.

Stemming and Lemmatization

1. The common goal is to reduce a word a simpler form.
2. Stemming reduces a token to *pseudostems* (not necessarily real words) using a heuristic process that chops off or replaces prefixes or suffixes.
3. Lemmatization uses a vocabulary to find the dictionary form of the word, known as a lemma.
4. This is a special case of normalization, like lower-casing all the letters.

Porter Stemmer

M.R. Porter, *"An algorithm for suffix stripping",* 1980.
https://tartarus.org/martin/PorterStemmer/def.txt

Probably the most popular.

Uses 5 phases of word reductions.

Each phase has rules for replacing the longest suffix.

Downloadable as a library but off-limits for the project.

Rules can have conditions

*S - the stem ends with S (and similarly for the other letters).
v - the stem contains a vowel.
*d - the stem ends with a double consonant (e.g. -TT, -SS).
*o - the stem ends cvc, where the second c is not W, X or Y (e.g. -WIL, -HOP).

Step 1a

SSES -> SS	caresses -> caress
IES -> I	ponies -> poni
	ties -> ti
SS -> SS	caress -> caress
S ->	cats -> cat

Step 1b

(m>0) EED -> EE	feed -> feed
	agreed -> agree
(**v**) ED ->	plastered -> plaster
	bled -> bled
(**v**) ING ->	motoring -> motor
	sing -> sing

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.
5. Some number of bits may encode attributes.
6. Certainly don’t want to have to add all the previous deltas (there could be millions of them!) to know the actual location number.
7. Synchronization points allow seeking to a location just prior to desired location, then scanning forward, so you only have to add up a smaller number of deltas.

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

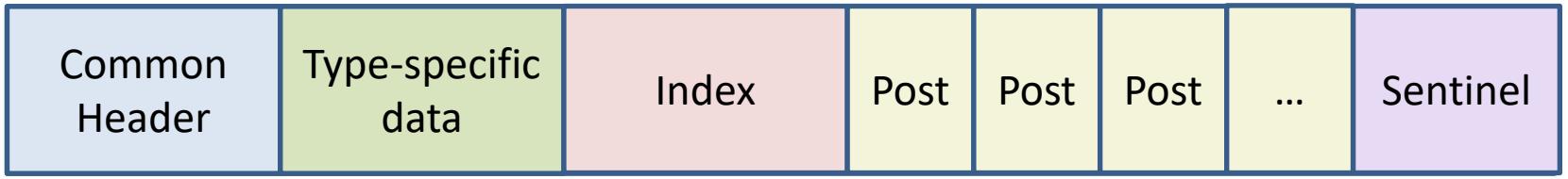
Common header contains:

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.
4. Size of the list for skipping over collisions.

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

1. Lengths of the document, URL and title.
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
:	:	:

Imagine we used 16-bit location numbers and consider a seek to hex 0x01AD.

We might use the high byte, 0x01, as the index into a seek table of 256 entries.

SeekTable[0x01] should contain two members:

1. A pointer to the first posting that occurs at a word location >= 0x0100 or null if there isn't one.

2. The actual word location of that first post, perhaps 20142.

Posting list index

	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
:	:	:

To seek to a specific word location without having to start from the beginning, adding up all the deltas until you get there, you use the seek table to get you close.

It gives you a synchronization point where you can jump in and start reading, looking for the location you want.

From there, you read forward, accumulating location deltas until we either hit a post at the seek address or the first post after that (or end of list).

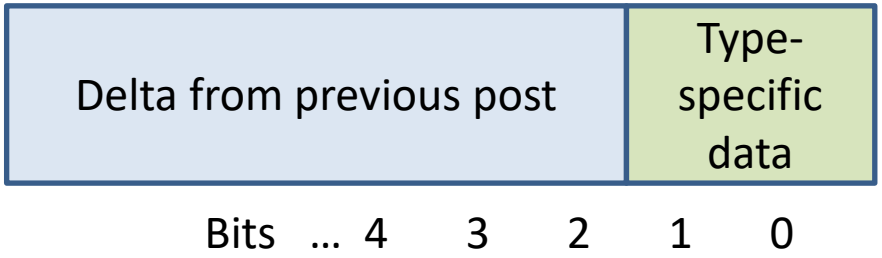
Posting list index

	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.

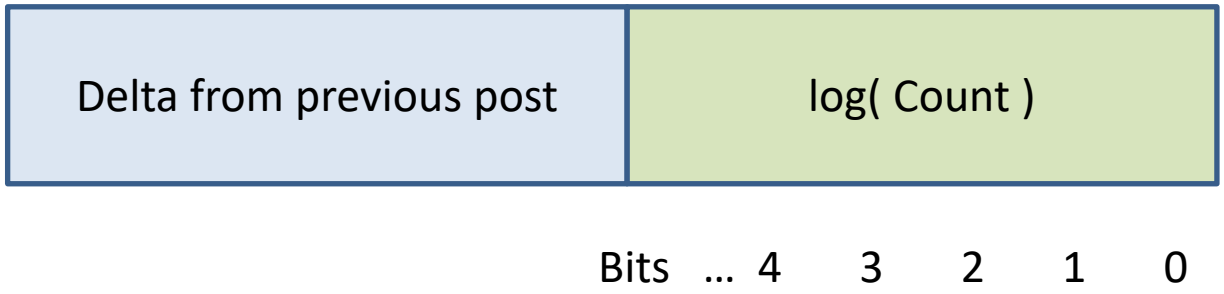


00 Normal
01 Italic
10 Bold
11 Heading

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.



Agenda

1. Course details.
2. CRCs.
3. The index.
4. The constraint solver.

Constraint solver

Given an inverted word index and a constraint, e.g., a list of words that must appear together or as a phrase in a document, find the list of matching documents.

Original inventor

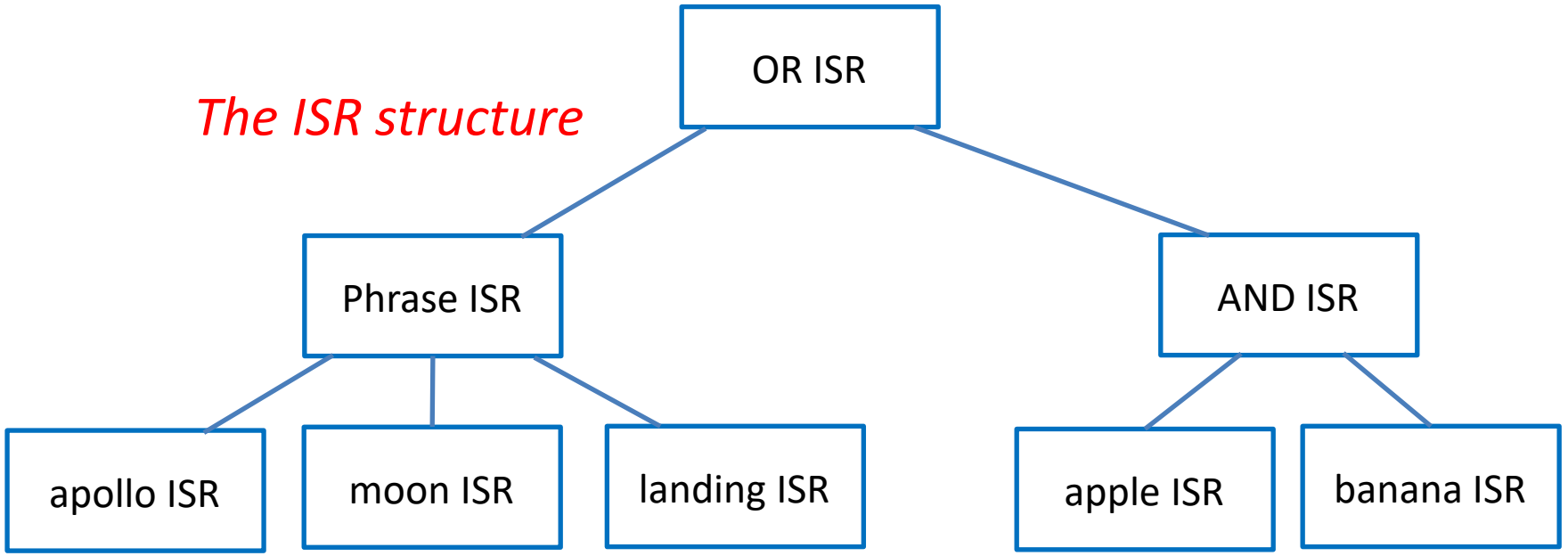
1. Mike Burrows at AltaVista.
2. Briefly worked for Microsoft at the very beginning of what became Bing and contributed this to the initial design.
3. (Left because his girlfriend was in the SF Bay area, shortly before I joined the team.)

Two uses

1. *Find matching documents.* Skip over documents that don't match as fast as possible.
2. *Rank those documents.* Scan through all the occurrences of the search words and phrases on a matching page as fast as possible.

Basic idea to create ISR structures that match the query constraints.
Each ISR can find the next occurrence of whatever it's looking for.

"apollo moon landing" | (apple banana)



Index Stream Reader (ISR)

Finds the next occurrence of the desired token or combination of child ISRs.

`ISRWord`	Find occurrences of individual words.
`ISREndDoc`	Find occurrences of document ends.
`ISROr`	Find occurrences of any child ISR.
`ISRAnd`	Find occurrences of all child ISRs within a single document.
ISRPhrase	Find occurrences of all child ISRs as a phrase.
`ISRContainer`	Find occurrences of contained ISRs in a single document not containing any excluded ISRs.

```
typedef size_t Location;      // Location 0 is the null location.
typedef size_t FileOffset;

typedef union Attributes
   {
   WordAttributes Word;
   DocumentAttributes Document;
   };


class ISR;

class Post
   {
   public:
      virtual Location GetStartLocation( );
      virtual Location GetEndLocation( );
      virtual Attributes GetAttributes( );
   };
```

```
class PostingList
   {
   private:
      struct PostingListIndex
         {
         FileOffset Offset;
         Location PostLocation;
         };

      PostingListIndex *index;
      virtual char *GetPostingList( );
   public:
      virtual Post *Seek( Location );
   };


class Index
   {
   public:
         Location WordsInIndex,
         DocumentsInIndex,
         LocationsInIndex,
         MaximumLocation;

      ISRWord *OpenISRWord( char *word );
      ISRWord *OpenISREndDoc( );
   };
```

```
class Dictionary
  {
   public:
      ISR *OpenIsr( char *token );
      Location GetNumberOfWords( );
      Location GetNumberOfUniqueWords( );
      Location GetNumberOfDocuments( );
   };

class ISR
   {
   public:
      virtual Post *Next( );
      virtual Post *NextDocument( );
      virtual Post *Seek( Location target );
      virtual Location GetStartLocation( );
      virtual Location GetEndLocation( );
   };

class ISRWord : public ISR
   {
   public:
      unsigned GetDocumentCount( );
      unsigned GetNumberOfOccurrences( );
      virtual Post *GetCurrentPost( );
   };
```

```
class ISREndDoc : public ISRWord
   {
   public:
      unsigned GetDocumentLength( );
      unsigned GetTitleLength( );
      unsigned GetUrlLength( );
   };
```

Consider these posting lists

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704

To read and merge these lists, we need to move from one entry to the next.

We'll do that with an ISR (index stream reader).

The ISR for each token has to be able to report its current location and attributes, and it needs Next( ) and Seek( ) functions.

OR'ing streams

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704
quick | fox	10 27 87 105 106 513 515 518 520 550 1200

An OR ISR simply merges the streams.

No need to pay attention to document boundaries. Each post is in whichever posting list and whatever document it happens to be.

```
class ISROr : public ISR
   {
   public:
      ISR **Terms;
      unsigned NumberOfTerms;

      Location GetStartLocation( )
         {
         return nearestStartLocation;
         }

      Location GetEndLocation( )
         {
         return nearestEndLocation;
         }

      Post *Seek( Location target )
         {
         // Seek all the ISRs to the first occurrence beginning at
         // the target location. Return null if there is no match.
         // The document is the document containing the nearest term.
         }

      Post *Next( )
         {
         // Do a next on the nearest term, then return
         // the new nearest match.
         }
```

```
    Post *NextDocument( )
        {
        // Seek all the ISRs to the first occurrence just past
        // the end of this document.
        return Seek( DocumentEnd->GetEndLocation( ) + 1 );
        }


private:
    unsigned nearestTerm;
    // nearStartLocation and nearestEndLocation are
    // the start and end of the nearestTerm.
    Location nearestStartLocation, nearestEndLocation;
};
```

AND'ing streams

quick	10 27 105 513 518 520
brown	28 50 62 70 514 790
fox	87 106 515 550 1200
#DocEnd	112 570 1006 1704
quick fox	*?*

AND'ing of terms should find occurrences of all the terms within a single document.

Should it return every possible combination, every combination only changing the nearest ISR or the first match in each matching document?

AND'ing streams

Easier to consider if we show the document boundaries.

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

quick fox *?*

To determine what document a post falls within, we advance a #DocEnd ISR to the next document end, where we can retrieve information about the document, including its length.

This tells us the start and end points of the document and whether all the word ISRs point within the same document.

AND'ing streams

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

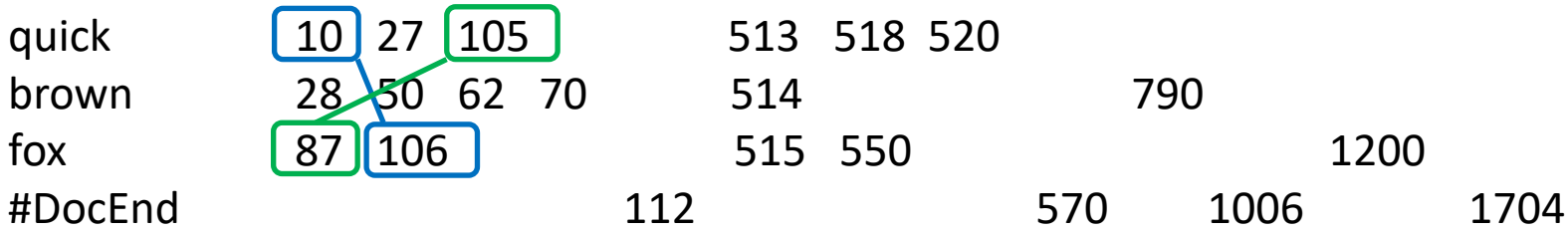
quick fox

How many possible combinations?
Can you reach all of them in a single pass, all ISRs only moving forward?

AND'ing of terms should find occurrences of all the terms within a single document.

Should it return every possible combination, every combination only changing the nearest ISR or the first match in each matching document?

AND'ing streams



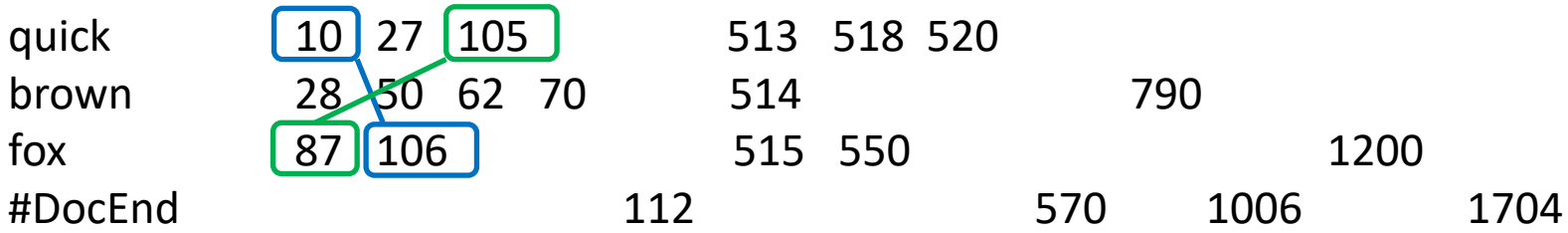
quick fox

How many possible combinations? 6

Can you reach all of them in a single pass, all ISRs only moving forward? No.

Should it return every possible combination, every combination only changing the nearest ISR or the first match in each matching document?

AND'ing streams



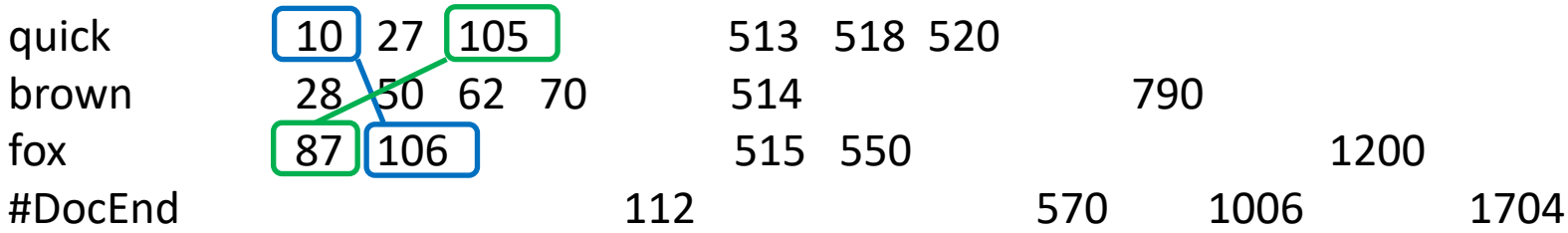
quick fox

How many possible combinations? 6
Can you reach all of them in a single pass, all ISRs only moving forward? No.

Should it return every possible combination, every combination only changing the nearest ISR or the first match in each matching document?

The point of the constraint solver is to find matching pages. Once any match on the page has been found, it's the ranker's job to figure out what to do next

AND'ing streams



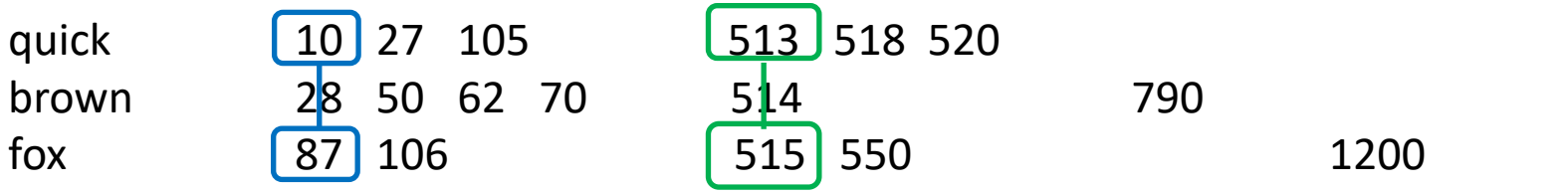
quick fox *How many possible combinations? 6*
Can you reach all of them in a single pass, all ISRs only moving forward? No.

You probably want both:

Next( ) Advance the nearest ISR and look for the first match.

NextDocument( ) Seeks all the ISRs past the end of the document then looks for the first match.

AND'ing streams



quick fox *NextDocument( ) matches*

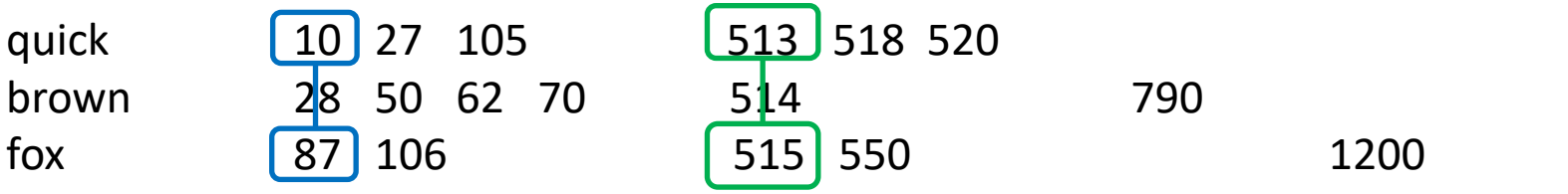
Next( ) Advance the nearest ISR and look for the first match.

returns (10 87) (27 87) (105 87) (105 106) (513 515) (518 515) (518 550) (520 550)

NextDocument( ) Seeks all the ISRs past the end of the document then looks for the first match.

returns (10 87) (513 515)

AND'ing streams



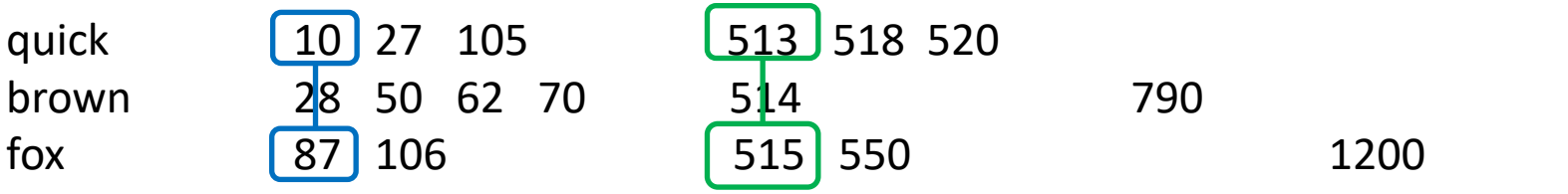
quick fox *NextDocument( ) matches*

To look for a new match, your objective is to skip forward through the index as fast as possible.

If a match is to be made including any of the present set of ISR positions, it must include whatever post is furthest down the index.

So there's no point in considering posts on the other lists that occur before the beginning of the document containing that furthest post.

AND'ing streams



quick fox *NextDocument( ) matches*

To look for a match:

1. Advance the #EndDoc ISR to just past the furthest ISR to get the length of the document.
2. Advance the other ISRs to their first matches starting at the beginning of the document.
3. If any ISR is past the end of document, you pick the new furthest and continue searching.

```
class ISRAnd : public ISR
   {
   public:
      ISR **Terms;
      unsigned NumberOfTerms;
      Post *Seek( Location target )
         {
         // 1. Seek all the ISRs to the first occurrence beginning at
         //    the target location.
         // 2. Move the document end ISR to just past the furthest
         //    word, then calculate the document begin location.
         // 3. Seek all the other terms to past the document begin.
         // 4. If any term is past the document end, return to
         //    step 2.
         // 5. If any ISR reaches the end, there is no match.
         }
      Post *Next( )
         {
         return Seek( nearestStartLocation + 1 );
         }

   private:
      unsigned nearestTerm, farthestTerm;
      Location nearestStartLocation, nearestEndLocation;
   };
```

```
class ISRAnd : public ISR
   {
   public:
      ISR **Terms;
      unsigned NumberOfTerms;
      Post *Seek( Location target )
         {
         // 1. Seek all the ISRs to the first occurrence beginning at
         //    the target location.
         // 2. Move the document end ISR to just past the furthest
         //    word, then calculate the document begin location.
         // 3. Seek all the other terms to past the document begin.
         // 4. If any term is past the document end, return to
         //    step 2.
         // 5. If any ISR reaches the end, there is no match.
         }
      Post *Next( )
         {
         return Seek( nearestStartLocation + 1 );
         }

   private:
      unsigned nearestTerm, farthestTerm;
      Location nearestStartLocation, nearestEndLocation;
   };
```

Phrase match

Terms must be in consecutive locations.

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox" (513 514 515)

Length of the match must equal to sum of the lengths of the terms.

If a match is to be made including any of the present set of ISR positions, it must include whichever post is furthest down the index.

Phrase match

Terms must be in consecutive locations.

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox" (513 514 515)

Can phrase matches be overlapping?

Do you need to pay attention to document boundaries?

If it’s not a match, do all the ISRs have to move?

Phrase match

Terms must be in consecutive locations.

quick	10	27	105			513	518	520						
brown	28	50	62	70		514				790				
fox	87	106				515	550					1200		
#DocEnd					112				570		1006		1704	

"quick brown fox" (513 514 515)

Can phrase matches be overlapping? *Yes, if beginning and ending terms match.*

Do you need to pay attention to document boundaries? *No, not if you skip a location number between documents. All phrase matches will always be within a single document.*

If it's not a match, do all the ISRs have to move? *No, you iterate, trying to move the nearest to correct position relative to the furthest.*

Phrase match

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox" (513 514 515)

So, what are the functions you might want? *Probably want both Next( ) and NextDocument( ).*

Phrase match

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox" (513 514 515)

So, what are the functions you might want? *Probably want both Next( ) and NextDocument( ).*

Phrase match

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

"quick brown fox" (513 514 515)

To look for a match:

1. Pick the furthest ISR.
2. Advance the other ISRs to their first matches starting at exactly where they should appear to be a matching phrase.
3. If any ISR is past the desired location, pick the new furthest and continue searching.

```
class ISRPhrase : public ISR
   {
   public:
      ISR **Terms;
      unsigned NumberOfTerms;
      Post *Seek( Location target )
         {
         // 1. Seek all ISRs to the first occurrence beginning at
         //    the target location.
         // 2. Pick the furthest term and attempt to seek all
         //    the other terms to the first location beginning
         //    where they should appear relative to the furthest
         //    term.
         // 3. If any term is past the desired location, return
         //    to step 2.
         // 4. If any ISR reaches the end, there is no match.
         }

      Post *Next( )
         {
         // Finds overlapping phrase matches.
         return Seek( nearestStartLocation + 1 );
         }
   };
```

NOTs

Terms that cannot appear in a matching document.

quick	10	27	105			513	518	520					
brown	28	50	62	70		514				790			
fox	87	106				515	550					1200	
#DocEnd					112				570		1006		1704

brown -fox 790

-fox *Not allowed*

A NOT matches anywhere the term doesn't appear, which is likely pretty nearly everywhere.

So we don't allow searches for nots alone and we don't check for exclusions until we've found an otherwise matching page.

Container ISRs

ISRs that must match and those that must not within a document.

quick	10	27	105			513	518	520								
brown	28	50	62	70		514					790					
fox	87	106				515	550						1200			
#DocEnd					112				570			1006		1704		

brown -fox 790

-fox *Not allowed*

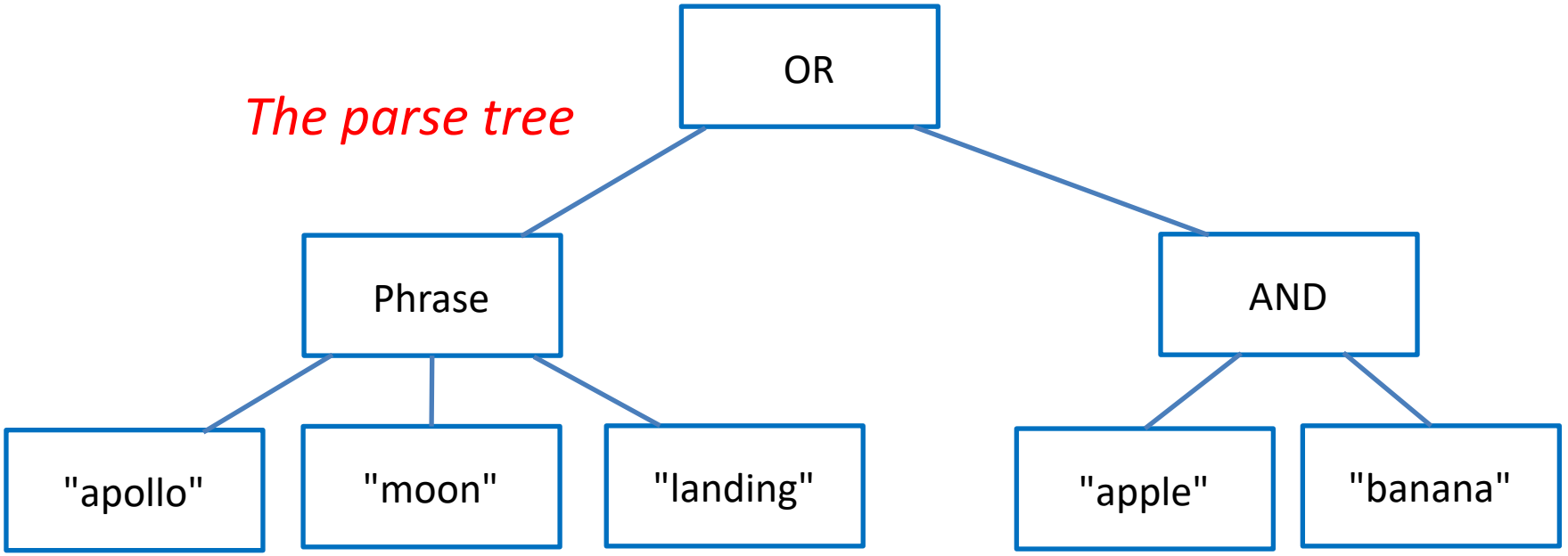
(AND'ing is a special case of a container with no exclusion ISRs.)

```
class ISRContainer : public ISR
   {
   public:
      ISR **Contained,
          *Excluded;
      ISREndDoc *EndDoc;
      unsigned CountContained,
         CountExcluded;
      Location Next( );

      Post *Seek( Location target )
         {
         // 1. Seek all the included ISRs to the first occurrence beginning at
         //    the target location.
         // 2. Move the document end ISR to just past the furthest
         //    contained ISR, then calculate the document begin location.
         // 3. Seek all the other contained terms to past the document begin.
         // 4. If any contained erm is past the document end, return to
         //    step 2.
         // 5. If any ISR reaches the end, there is no match.
         // 6. Seek all the excluded ISRs to the first occurrence beginning at
         //    the document begin location.
         // 7. If any excluded ISR falls within the document, reset the
         //    target to one past the end of the document and return to
         //    step 1.
         };
```

```
    Post *Next( )
        {
        Seek( Contained[ nearestContained ]->GetStartlocation( ) + 1 );
        }

private:
    unsigned nearestTerm, farthestTerm;
    Location nearestStartLocation, nearestEndLocation;
};
```

The query language and the ISRs can be recursive

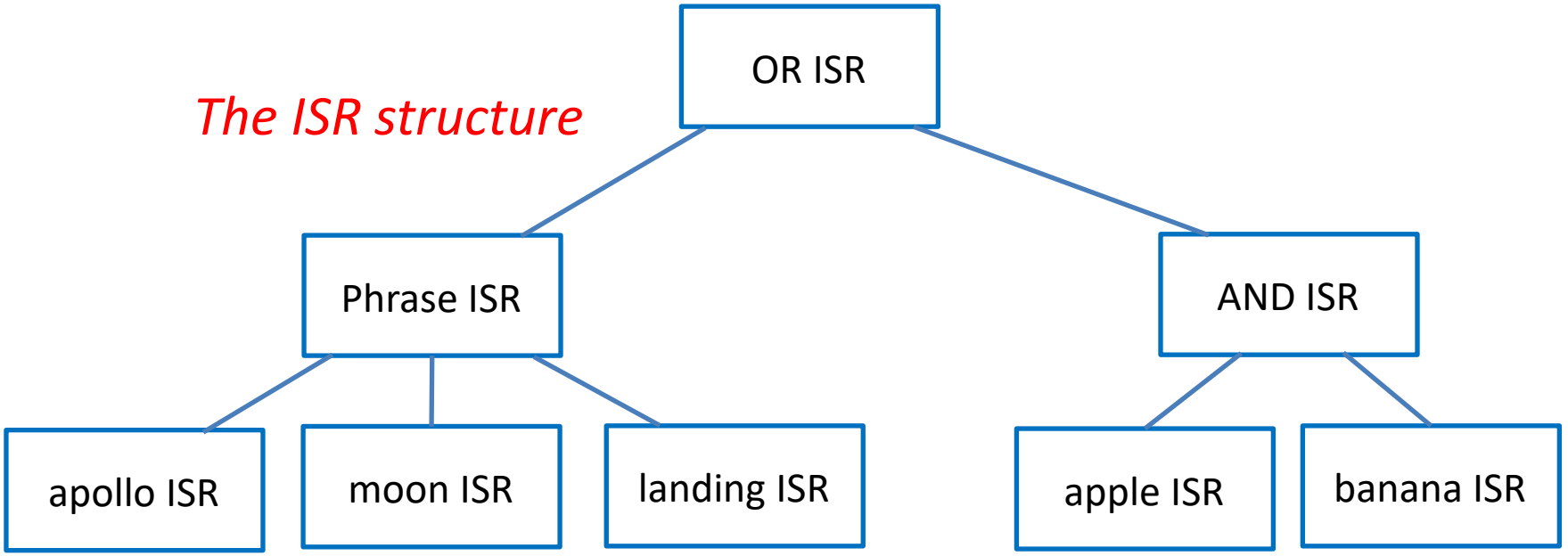
"apollo moon landing" | (apple banana)



The query language and the ISRs can be recursive

"apollo moon landing" | (apple banana)

The ISR structure



"apollo moon landing" | (apple banana)

The trees are the same.