



ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Crawler Overview

Basic Crawler operation

- Initialize queue with URLs of known seed pages
- Repeat
 - Take URL from queue
 - Fetch and parse page
 - Extract URLs from page
 - Add URLs to queue
- Fundamental assumption: The web is well linked.
- If this is implemented as Queue, what sort of traversal of the web graph does it represent ?
 - Roughly (since it is initiated from several seeds) it is equivalent to Breadth First Search since it is FIFO based (deleting from front and inserting at the back)

What's wrong with this crawler

urlqueue := (some carefully selected set of seed urls) while

urlqueue is not empty:

 myurl := urlqueue.getlastanddelete() mypage :=

 myurl.fetch() fetchedurls.add(myurl)

 newurls := mypage.extracturls() for myurl in

 newurls:

 if myurl not in fetchedurls and not in urlqueue:

 urlqueue.add(myurl) addtoinvertedindex(mypage)

What is wrong with the simple crawler

- **Scale:** we need to **distribute**.
- **We can't index everything:** we need to **subselect**. How?
- **Duplicates:** need to integrate **duplicate detection**
- **Spam and spider traps:** need to integrate **spam detection**
- **Politeness:** we need to be “nice” and space out all requests for a site over a longer period (hours, days)
- **Freshness:** we need to recrawl periodically.
 - Because of the size of the web, we can do frequent recrawls only for a small subset.
 - Again, subselection problem or **prioritization**

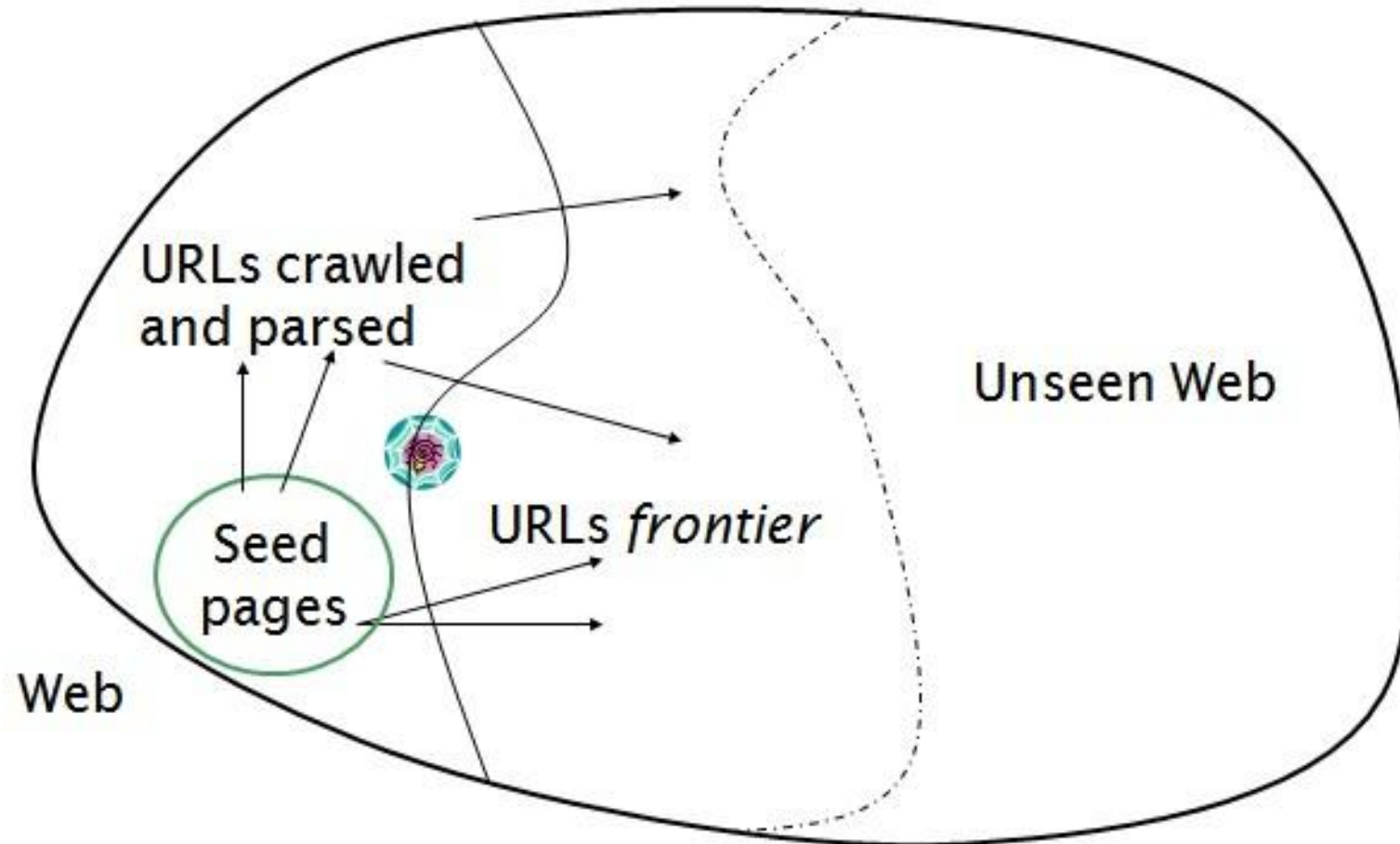
Magnitude of the crawling problem

- To fetch 20,000,000,000 pages in one month . . .
- . . . we need to fetch almost 8000 pages per second!
- Actually: many more since many of the pages we attempt to crawl will be duplicates, unfetchable, spam etc.

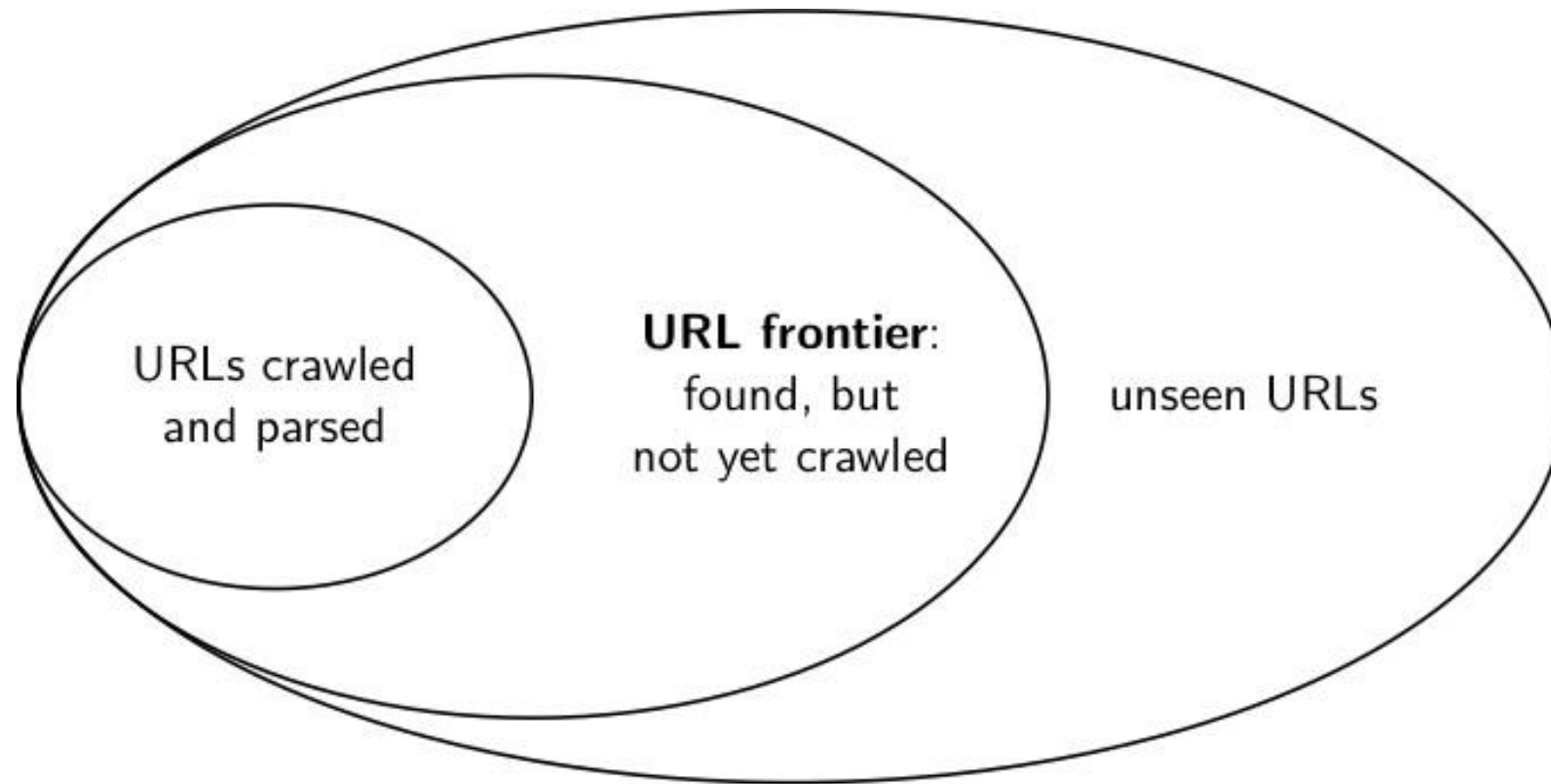
Basic Crawler operation

- Crawler also called Robot and Spider
- Job of the Crawler is to get the web documents that the indexer indexes.
- Crawler Workflow :
 - Begin with known “seed” URLs
 - Fetch and parse them
 - Extract URLs they point to
 - Place the extracted URLs on a queue
 - Fetch each URL on the queue and repeat

Crawling Picture



URL frontier consists of the set of URLs
identified for crawling but not yet crawled



Simple picture- complications

- Web crawling **isn't feasible with one machine**
 - All of the above steps distributed
 - Malicious pages
 - Spam pages
 - Spider traps – incl dynamically generated
- Even **non-malicious pages pose challenges**
 - Latency/bandwidth to remote servers vary
 - Webmasters' stipulations
 - How “deep” should you crawl a site's URL hierarchy?
 - Site mirrors and duplicate pages
- Politeness – don't hit a server too often

What any crawler must do - 1

- **Be Polite:** Respect implicit and explicit politeness considerations
 - Only crawl allowed pages
 - Respect *robots.txt* (more on this shortly)
- **Be Robust:** Be immune to spider traps and other malicious behavior from web servers

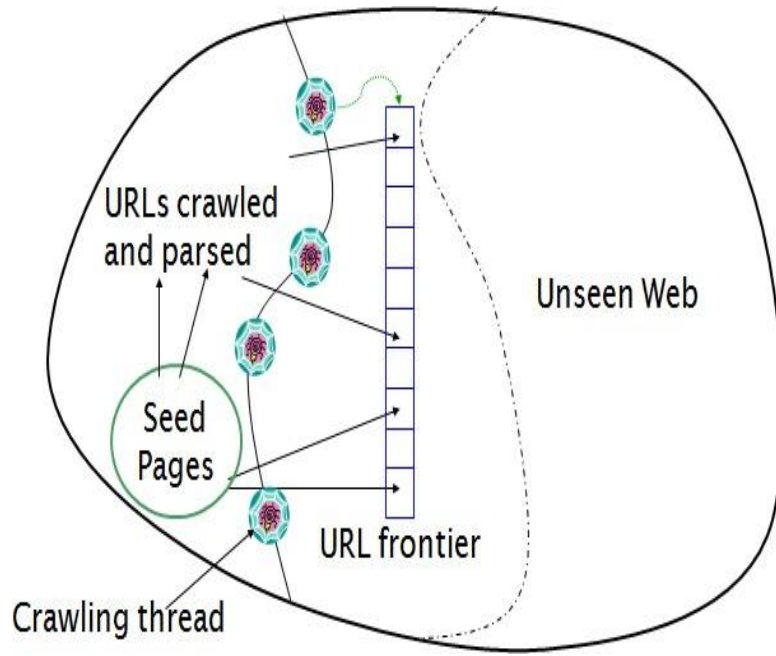
What any crawler should do - 2

- Be capable of distributed operation: designed to run on multiple distributed machines (may not be able to fetch all pages into a single machine)
- Be scalable: designed to increase the crawl rate by adding more machines to your distributed systems
- Performance/efficiency: permit full use of available processing and network resources

What any crawler should do - 3

- Fetch pages of “higher quality” first
- Continuous operation: Continue fetching fresh copies of a previously fetched page (crawling is never over)
- Extensible: Adapt to new data formats, protocols

Updated Crawler picture



- We have multiple spiders (distributed or multiple threads)
- URL frontier depicted as queue (structure later)
- Each crawler thread requests the next URL from frontier
- Parses the text at that URL
- Text sent to Indexing pipeline
- New found URLs are pushed back to URL frontier
- Region shown by Seed pages will grow into “crawled and parsed pages”

URL frontier Specifications - 1

1. Can include multiple pages from the same host
2. Must avoid trying to fetch them all at the same time
3. Must try to keep all crawling threads busy
4. There has to be a trade-off between 2 and 3

- Explicit politeness: specifications from webmasters on what portions of site can be crawled
 - robots.txt
- Implicit politeness: even with no specification, avoid hitting any site too often

Robots.txt example

- Protocol for giving spiders (“robots”) limited access to a website, originally from 1994
 - www.robotstxt.org/wc/norobots.html
 - Website announces its request on what can(not) be crawled
 - For a server, create a file /robots.txt
 - This file specifies access restrictions
- No robot should visit any URL starting with
"/yoursite/temp/", except the robot called
“searchengine”:
User-agent: *
Disallow:
/yoursite/temp/

- DNS : lookup service on the internet
 - Given a URL, retrieve its IP address
 - Service provided by a distributed set of servers – thus, lookup latencies can be high (even seconds)
- Common OS implementations of DNS lookup are *blocking*: only one outstanding request at a time
- Solutions
 - DNS caching
 - Batch DNS resolver – collects requests and sends them out together

Parsing: URL normalization

- When a fetched document is parsed, some of the extracted links are *relative* URLs
- During parsing, must normalize (expand) such relative URLs

Content Seen?

- Duplication is widespread on the web
- If the page just fetched is already in the index, do not further process it
- This is verified using document fingerprints or shingles

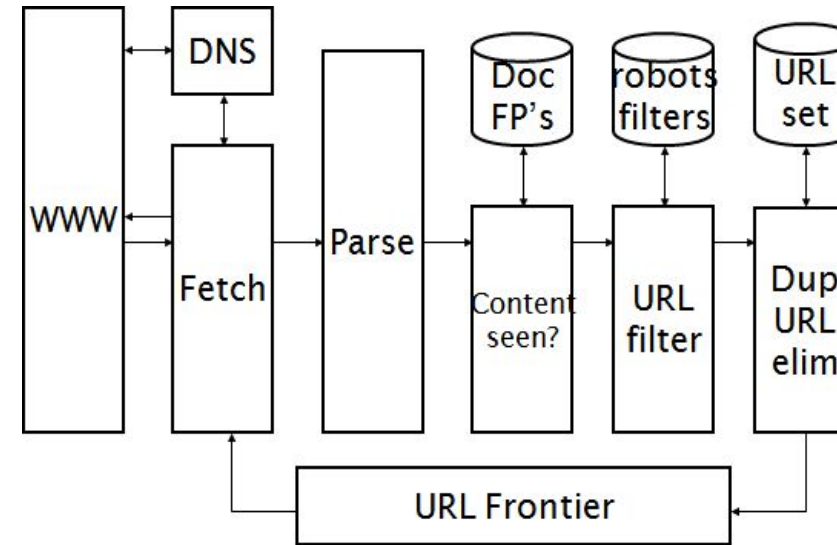
- Filters – regular expressions for URL's to be crawled/not
- Once a robots.txt file is fetched from a site, need not fetch it repeatedly
 - Doing so burns bandwidth, hits web server
- Cache robots.txt files

Duplicate URL elimination

- For a non-continuous (one-shot) crawl, test to see if an extracted+filtered URL has already been passed to the frontier
- For a continuous crawl – see details of frontier implementation

Processing steps in crawling

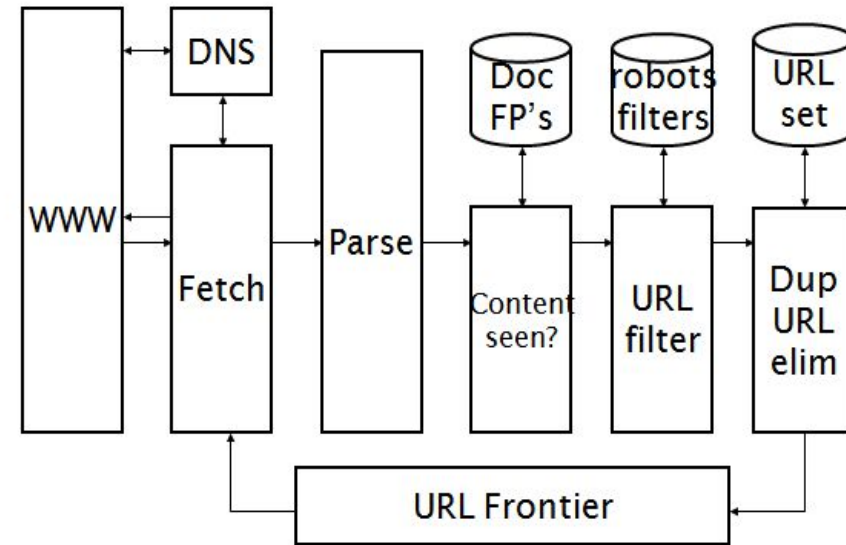
- Pick a URL from the frontier
- Fetch the document at the URL
- Parse the URL
 - Extract links from it to other docs (URLs)
- Check if URL has content already seen
 - If not, add to indexes
- For each extracted URL
 - Ensure it passes certain URL filter tests
 - Check if it is already in the frontier (duplicate URL elimination)



E.g., only crawl .edu,
obey robots.txt, etc.

Recap : Processing steps in crawling

- Pick a URL from the frontier
- Fetch the document at the URL
- Parse the URL
 - Extract links from it to other docs (U
- Check if URL has content already seen
 - If not, add to indexes
- For each extracted URL
 - Ensure it passes certain URL filter tests
 - Check if it is already in the frontier (duplicate URL elimination)



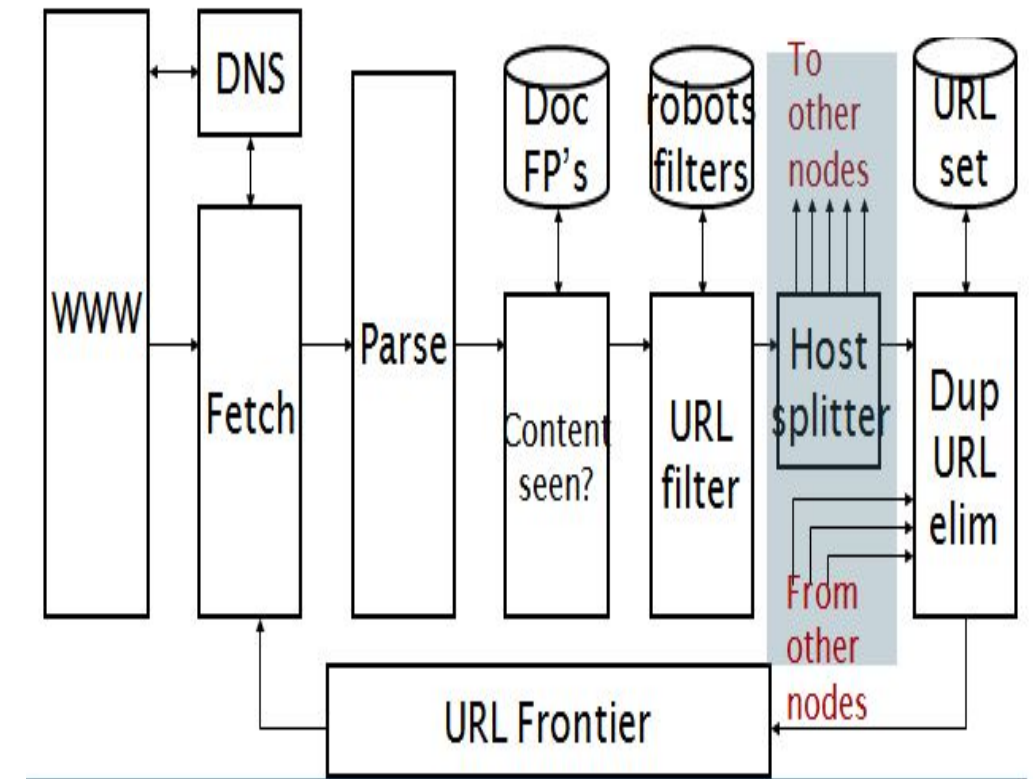
E.g., only crawl .edu,
obey robots.txt, etc.

Distributing the Crawler

- Run multiple crawl threads, under processes – different potentially at different nodes
 - Geographically distributed nodes
- Partition hosts being crawled into nodes
 - Hash used for partition
- How do these nodes communicate and share URLs?

Communication between Nodes in a Setup with Multiple Hosts

- Had it not been distributed, processed URL would have been pushed to URL frontier
- In the distributed setup, output of the URL filter at each node is sent to the Dup URL Eliminator of the appropriate node (hash into a host id)
- Like this particular node is sending a URL, it is also receiving processed URL
- Since already processed, previous processing steps can be skipped



URL Frontier: two main considerations

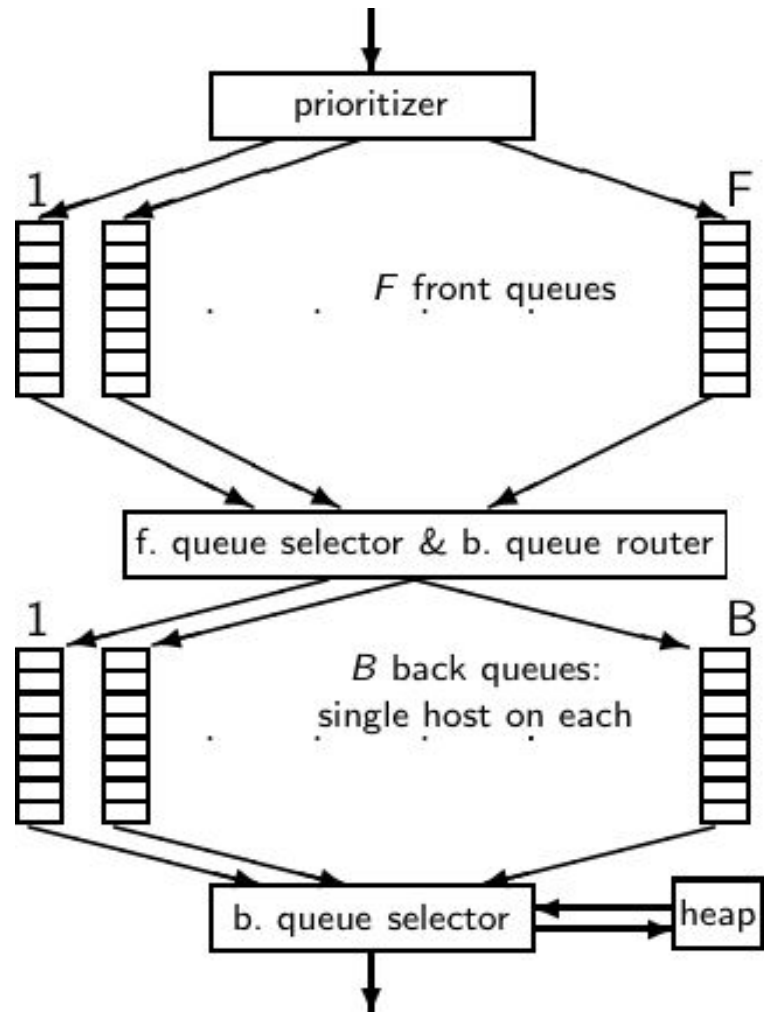
- Politeness: do not hit a web server too frequently
- Freshness: crawl some pages **more often than others** (=priority)
 - E.g., pages (such as News sites) whose content changes often

Had the URL frontier been implemented as a simple FIFO queue, **these goals may conflict each other.**

- A popular site may have many links out of a page go to its own site and these pages need to be refreshed for indexing
- They would have been inserted together into the FIFO queue
- They would be taken out together from the FIFO queue for the crawler threads creating a burst of accesses to that site
- This would have violated “politeness”

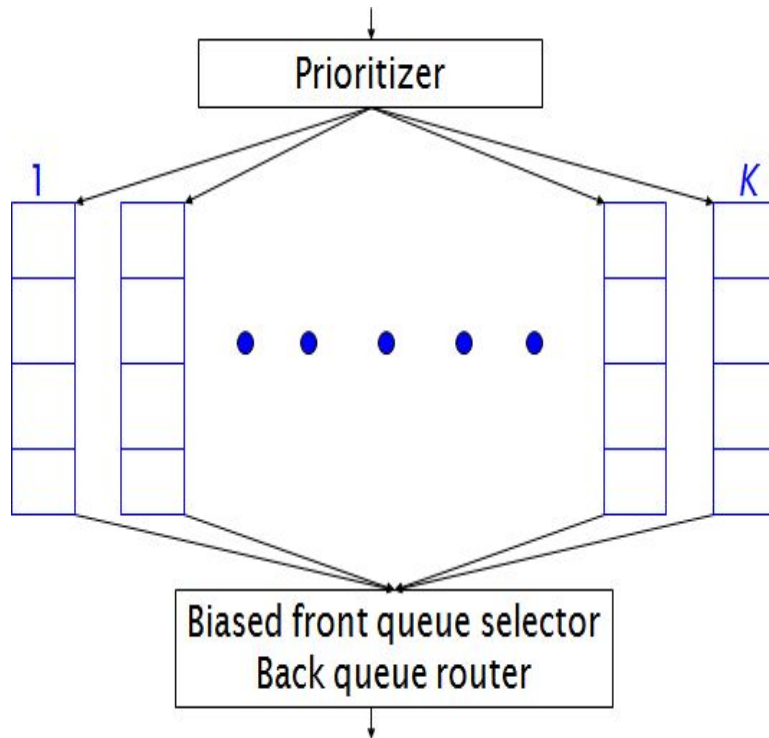
Freshness

- Prioritizer assigns to URL an integer priority between 1 and K
 - Appends URL to corresponding queue
- Heuristics for assigning priority
 - Refresh rate sampled from previous crawls
 - Application-specific (e.g., “crawl news sites more often”)

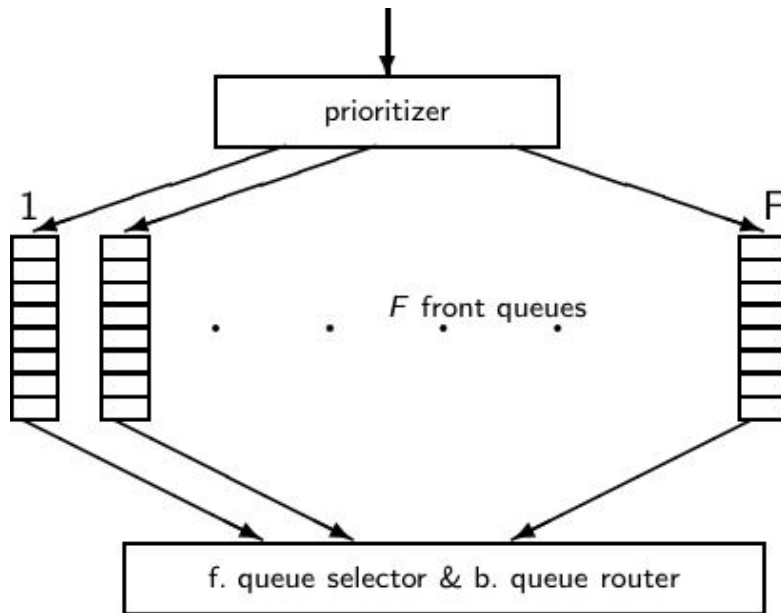


- URLs flow in from the top into the frontier.
- Front queues manage prioritization.
- Back queues enforce politeness. Heap helps in doing that
 - Each queue is FIFO.

Front Queues : Freshness

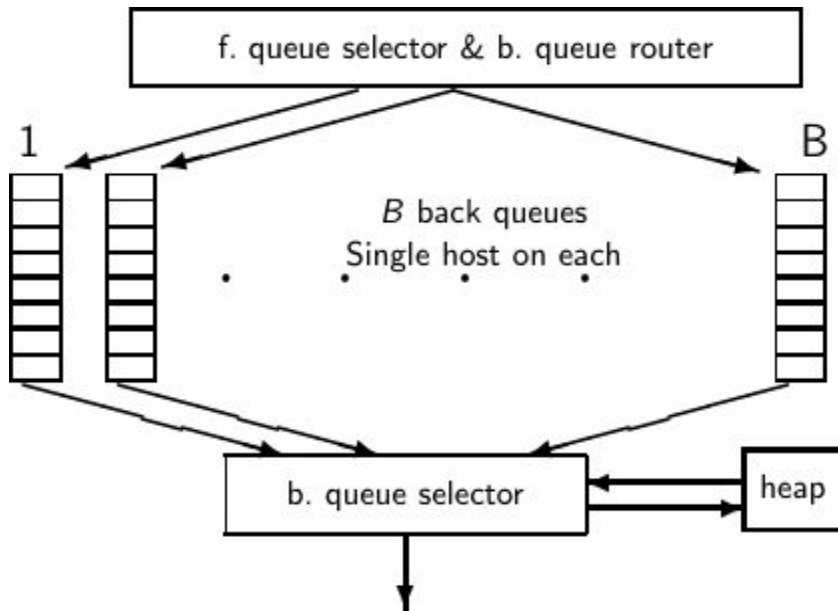


- Prioritizer assigns to URL an integer priority between 1 and F .
- Priority 1 Queue for webserver that needs to be crawled very frequently (by tracking how frequently web pages are changing)
- Heuristics for assigning priority: refresh rate, PageRank etc
- Then appends URL to corresponding queue



- Selection from front queues is initiated by back queues
- Pick a front queue from which to select next URL: Round robin (Example, 1, 1- 2, 1-2-3, etc), randomly (random number generated but a bigger range meant for high priority queue), or more sophisticated variant
- With a bias in favor of high-priority front queues.

Back Queue : Politeness



- **Invariant 1.** Each back queue only contains URLs from a single host server.
- **Invariant 2.** Each back queue is kept non-empty while the crawl is in progress
- Maintain a table from hosts to back queues.

| Host name | Back queue |
|-----------|------------|
| ... | 3 |
| | 1 |
| | B |

Back queue heap

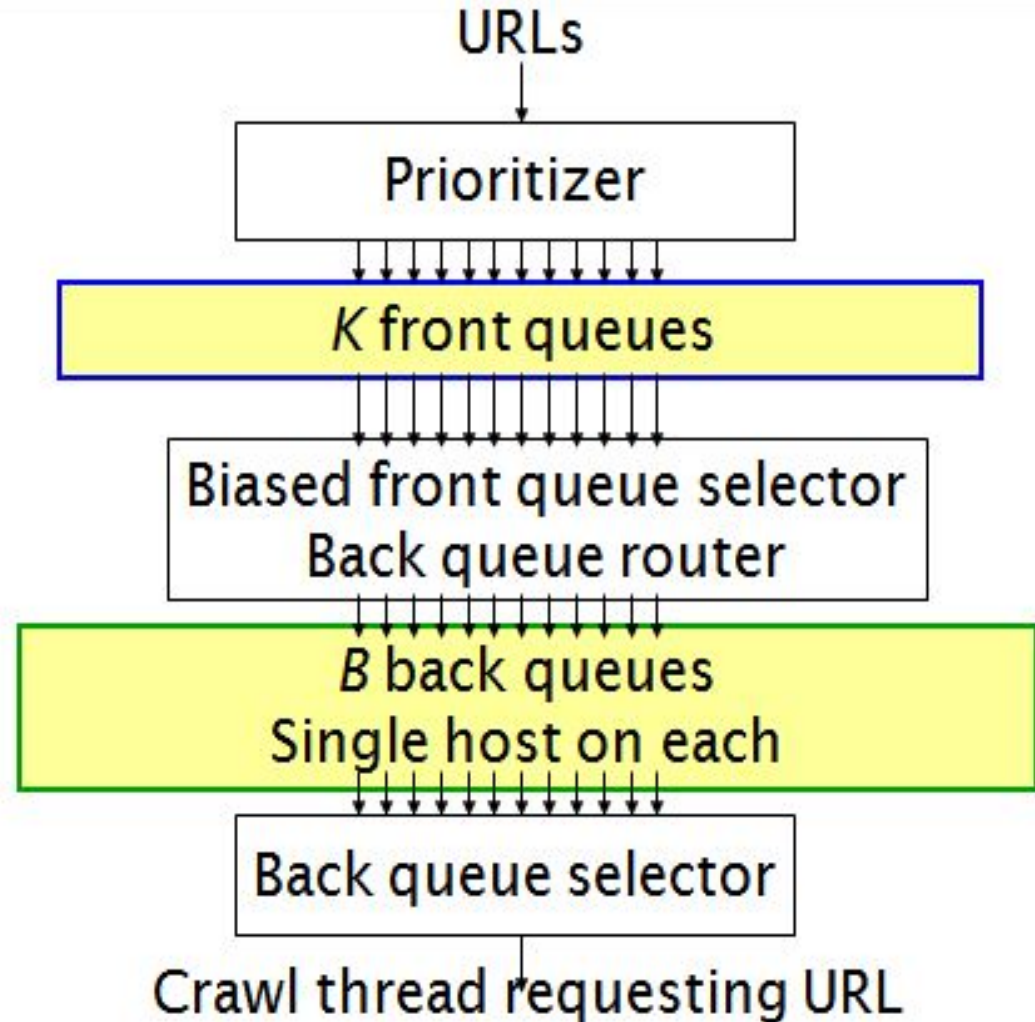
- One entry for each back queue
- The entry is the earliest time t_e at which the host corresponding to the back queue can be hit again
- This earliest time is determined from
 - Last access to that host
 - Any time buffer heuristic we choose
- The min-heap keeps the “earliest hitting time” for all the servers and root of the min-heap is selected

Back Queue Processing : How Fetcher Interacts with back Queue

- A crawler thread seeking a URL to crawl:
 - Extracts the root of the heap
- Fetches URL at head of corresponding back queue q (look up from table)
- Checks if queue q is now empty – if so, pulls a URL v from front queues
 - If there's already a back queue q for v 's host, append v to q and pull another URL from front queues, repeat till the back queue is filled
 - Else add v to q if v is a new host
- When q is non-empty, create heap entry for it

Number of back queue B

- Should we have as many back queue as there are web servers that we are crawling ?
 - No ! Number of back queues are limited i.e. a queue allocated for host A can get reallocated to host B
 - These back queues are filled in a priority driven manner by biased front queue selector
 - Mercator recommendation: three times as many back queues as crawler threads
- Keep all threads busy while respecting politeness



- **Apache Lucene** is a full-text search engine which can be used from various programming languages.
- Lucene is an open source Java based search library.
- It is used in Java based applications to add document search capability to any kind of application in a very simple and efficient way.
- Scalable and high-performance library

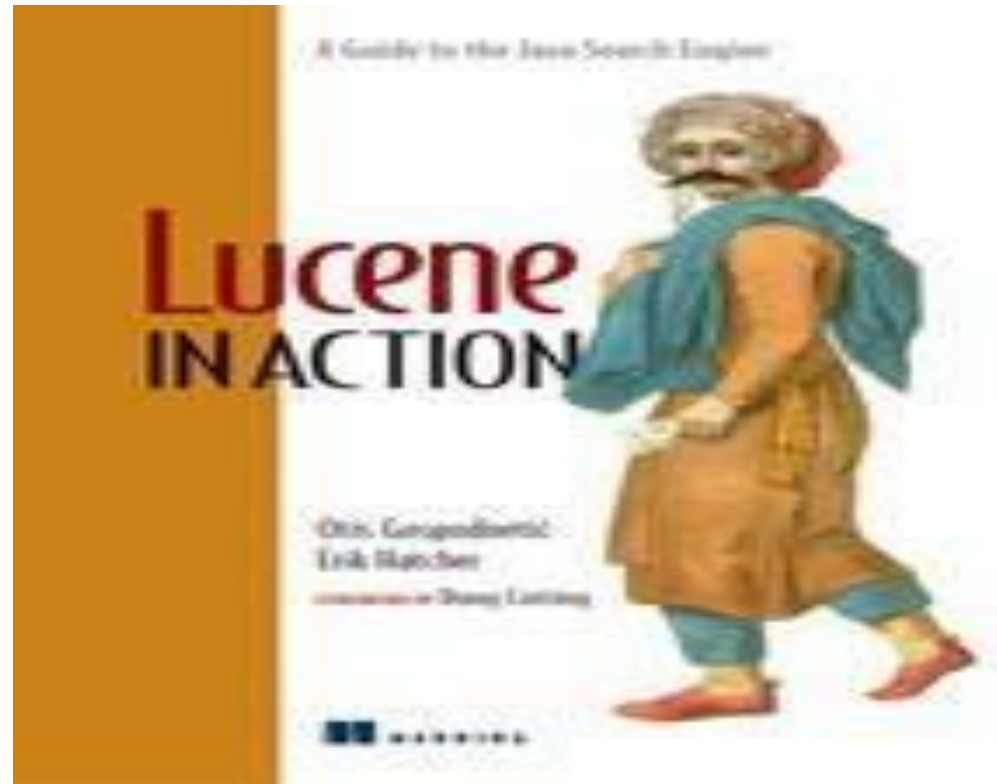
- Widely used academic systems
 - Terrier (Java, U. Glasgow) <http://terrier.org>
 - Indri/Galago/Lemur (C++ (& Java), U. Mass & CMU)
 - Tail of others (Zettair, ...)
- Widely used non-academic open source systems
 - **Lucene**
 - Things built on it: Solr, ElasticSearch
 - A few others (Xapian, ...)

- Open source Java library for indexing and searching
 - Lets you add search to your application
 - Not a complete search system by itself
 - Written by Doug Cutting
- Used by: Twitter, LinkedIn, Zappos, CiteSeer, Eclipse, ...
 - ... and many more (see <http://wiki.apache.org/lucene-java/PoweredBy>)
- Ports/integrations to other languages
 - C/C++, C#, Ruby, Perl, Python, PHP, ...

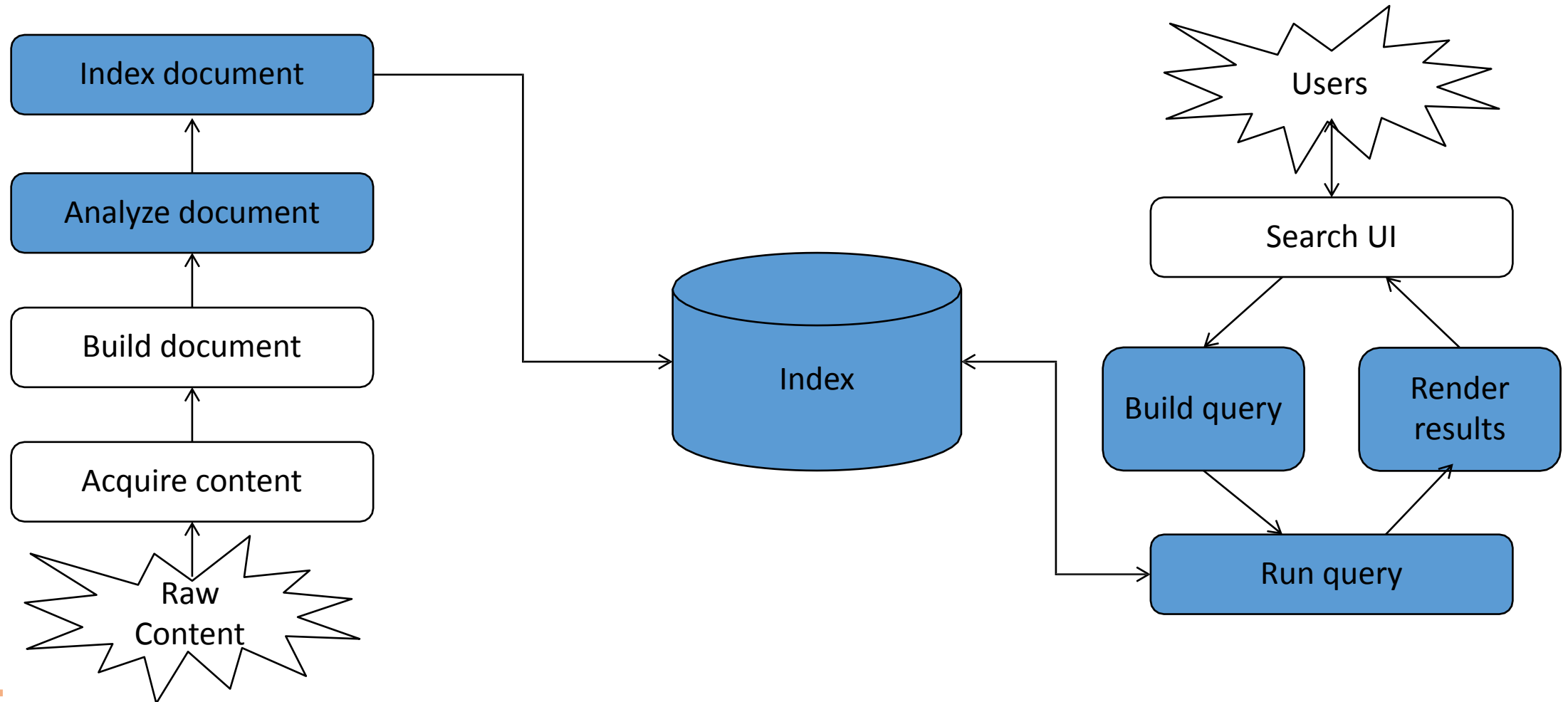


Reference - Based on “Lucene in Action”

By Michael McCandless, Erik Hatcher, Otis Gospodnetic



Covers Lucene 3.0.1. It's now up to 5.1.0



- **Acquire Raw Content:** The first step of any search application is to collect the target contents on which search application is to be conducted.
- **Build the document:** build the document(s) from the raw content, which the search application can understand and interpret easily.
- **Analyze the document :** Before the indexing process starts, the document is to be analyzed as to which part of the text is a candidate to be indexed. This process is where the document is analyzed.
- **Indexing the document:** Once documents are built and analyzed, the next step is to index them so that this document can be retrieved based on certain keys instead of the entire content of the document.

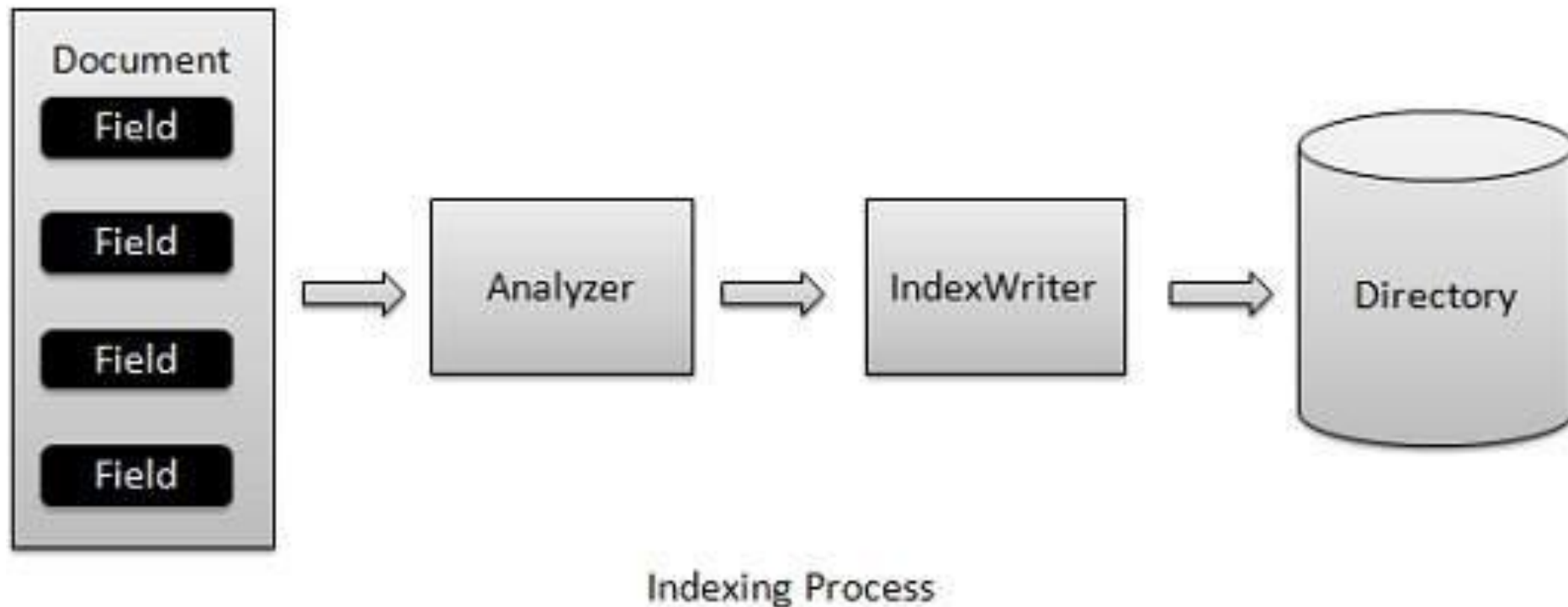
- **User Interface for Search:** Once a database of indexes is ready then the application can make any search. To facilitate a user to make a search, the application must provide a user a **mean** or a **user interface** where a user can enter text and start the search process.
- **Build Query:** Once a user makes a request to search a text, the application should prepare a Query object using that text which can be used to inquire index database to get the relevant details.
- **Search Query :** Using a query object, the index database is then checked to get the relevant details and the content documents.
- **Render Results :** Once the result is received, the application should decide on how to show the results to the user using User Interface. How much information is to be shown at first look and so on.

- Source files in lia2e/src/lia/meetlucene/
 - Actual sources use Lucene 3.6.0
 - Code in these slides upgraded to Lucene 5.1.0
- Command line **Indexer**
 - lia.meetlucene.Indexer
- Command line **Searcher**
 - lia.meetlucene.Searcher



Core Indexing Classes

- Indexing process is one of the core functionalities provided by Lucene. The following diagram illustrates the indexing process and the use of classes. **IndexWriter** is the most important and the core component of the indexing process.



Core Indexing Classes

- We add **Document(s)** containing **Field(s)** to **IndexWriter** which analyzes the **Document(s)** using the **Analyzer** and then creates/open/edit indexes as required and store/update them in a **Directory**. **IndexWriter** is used to update or create indexes. It is not used to read indexes.
- **IndexWriter** : This class acts as a core component which creates/updates indexes during the indexing process.
- **Directory** : This class represents the storage location of the indexes.
- **Analyzer** : This class is responsible to analyze a document and get the tokens/words from the text which is to be indexed. Without analysis done, **IndexWriter** cannot create index.

- **Document** : This class represents a virtual document with Fields where the Field is an object which can contain the physical document's contents, its meta data and so on. The Analyzer can understand a Document only.
- **Field** : This is the lowest unit or the starting point of the indexing process. It represents the key value pair relationship where a key is used to identify the value to be indexed. Let us assume a field used to represent contents of a document will have key as "contents" and the value may contain the part or all of the text or numeric content of the document. Lucene can index only text or numeric content only.



Core Indexing Classes

- **IndexWriter**

- Central component that allows you to create a new index, open an existing one, and add, remove, or update documents in an index
- Built on an IndexWriterConfig and a Directory

- **Directory**

- Abstract class that represents the location of an index

- **Analyzer**

- Extracts tokens from a text stream

Creating an IndexWriter

```
import org.apache.lucene.analysis.Analyzer;  
import org.apache.lucene.index.IndexWriter;  
import org.apache.lucene.index.IndexWriterConfig;  
import org.apache.lucene.store.Directory;  
...  
  
private IndexWriter writer;  
  
public Indexer(String dir) throws IOException {  
    Directory indexDir = FSDirectory.open(new File(dir));  
    Analyzer analyzer = new StandardAnalyzer();  
    IndexWriterConfig cfg = new IndexWriterConfig(analyzer);  
    cfg.setOpenMode(OpenMode.CREATE);  
    writer = new IndexWriter(indexDir, cfg)  
}
```


Core indexing classes (contd.)

- Document
 - Represents a collection of named Fields. Text in these Fields are indexed.
- Field
 - Note: Lucene Fields can represent both “fields” and “zones” as described in the textbook
 - Or even other things like numbers.
 - StringField are indexed but not tokenized
 - TextField are indexed and tokenized



A Document contain Fields

```
import org.apache.lucene.document.Document;  
import org.apache.lucene.document.Field;  
  
...  
protected Document getDocument(File f) throws Exception {  
    Document doc = new Document();  
    doc.add(new TextField("contents", new FileReader(f)))  
    doc.add(new StringField("filename",  
        f.getName(),  
        Field.Store.YES));  
  
    doc.add(new StringField("fullpath", f.getCanonicalPath(),Field.Store.YES));  
    return doc;  
}
```

Index a Document with IndexWriter

```
private IndexWriter writer;  
...  
private void indexFile(File f) throws  
    Exception {  
    Document doc = getDocument(f);  
    writer.addDocument(doc);  
}
```

Indexing a directory

```
private IndexWriter writer;  
...  
public int index(String dataDir, FileFilter filter)  
    throws Exception {  
    File[] files = new File(dataDir).listFiles();  
    for (File f: files) {  
        if (... &&  
            (filter == null || filter.accept(f))) {  
            indexFile(f);  
        }  
    }  
    return writer.numDocs();  
}
```

Closing the IndexWriter

```
private IndexWriter writer;  
  
...  
public void close() throws IOException {  
    writer.close();  
}
```



- The Index is the kind of inverted index we know and love
- The default Lucene50 codec is:
 - variable-byte and fixed-width encoding of delta values
 - multi-level skip lists
 - natural ordering of docIDs
 - encodes both term frequencies and positional information
- APIs to customize the codec



Core searching classes

- The process of Searching is again one of the core functionalities provided by Lucene. Its flow is similar to that of the indexing process. Basic search of Lucene can be made using the following classes which can also be termed as foundation classes for all search related operations.
- **IndexSearcher** : This class act as a core component which reads/searches indexes created after the indexing process. It takes directory instance pointing to the location containing the indexes.
- **Term** : This class is the lowest unit of searching. It is similar to Field in indexing process.
- **Query** : Query is an abstract class and contains various utility methods and is the parent of all types of queries that Lucene uses during search process.

Core searching classes

- **TermQuery** : TermQuery is the most commonly-used query object and is the foundation of many complex queries that Lucene can make use of.
- **TopDocs** : TopDocs points to the top N search results which matches the search criteria. It is a simple container of pointers to point to documents which are the output of a search result.



Core searching classes

- **IndexSearcher**

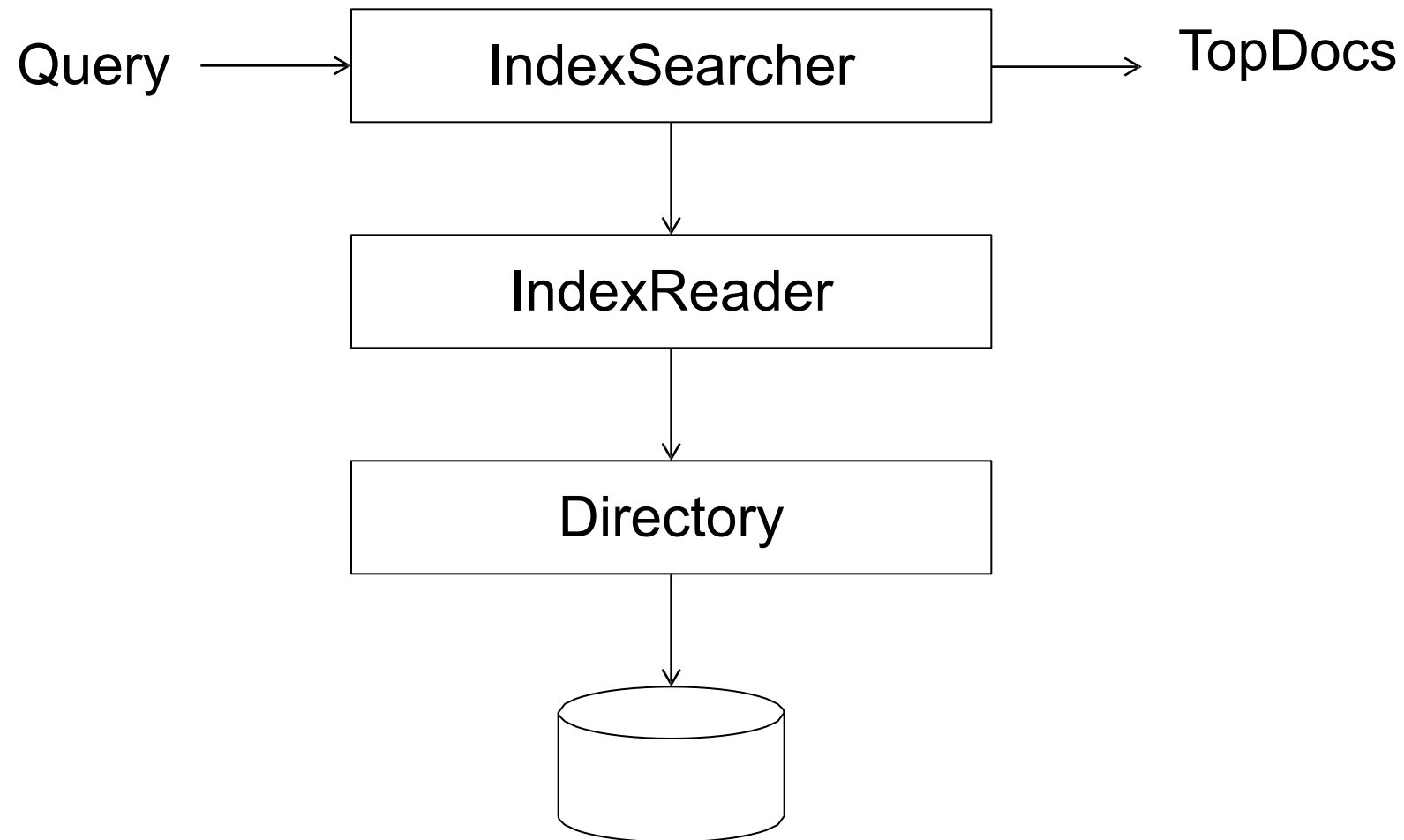
- Central class that exposes several search methods on an index
- Accessed via an `IndexReader`

- **Query**

- Abstract query class. Concrete subclasses represent specific types of queries, e.g.,
matching terms in fields, boolean queries, phrase queries, ...

- **QueryParser**

- Parses a textual representation of a query into a `Query` instance



Creating an IndexSearcher

```
import org.apache.lucene.search.IndexSearcher;  
  
...  
public static void search(String indexDir, String q) throws  
    IOException, ParseException {  
    IndexReader rdr = DirectoryReader.open(FSDirectory.open(new File(indexDir)));  
    IndexSearcher is = new IndexSearcher(rdr);  
    ...  
}
```

Query and QueryParser

```
import
org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.Query;

...

public static void search(String indexDir, String q)
    throws IOException, ParseException
    ...
    QueryParser parser =
        new QueryParser("contents", new StandardAnalyzer());
    Query query = parser.parse(q);
    ...
}
```

- TopDocs
 - Contains references to the top documents returned by a search
- ScoreDoc
 - Represents a single search result



search() returns TopDocs

```
import org.apache.lucene.search.TopDocs;  
...  
public static void search(String indexDir,String q)  
    throws IOException, ParseException  
    ...  
    IndexSearcher is = ...;  
    ...  
    Query query = ...;  
    ...  
    TopDocs hits = is.search(query, 10);  
}
```

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

TopDocs contain ScoreDocs

```
import org.apache.lucene.search.ScoreDoc;  
  
...  
public static void search(String indexDir, String q)  
    throws IOException, ParseException  
  
    ...  
    IndexSearcher is = ...;  
  
    ...  
    TopDocs hits = ...;  
  
    ...  
    for(ScoreDoc scoreDoc : hits.scoreDocs) {  
        Document doc = is.doc(scoreDoc.doc);  
        System.out.println(doc.get("fullpath"));  
    }  
}
```

```
public static void search(String indexDir, String q)  
    throws IOException, ParseException
```

```
...
```

```
IndexSearcher is = ...;
```

```
...
```

```
is.close();
```

```
}
```



How Lucene models content

- A Document is the atomic unit of indexing and searching
 - A Document contains Fields
- Fields have a name and a value
 - You have to translate raw content into Fields
 - Examples: Title, author, date, abstract, body, URL, keywords, ...
 - Different documents can have different fields
 - Search a field using name:term, e.g., title:lucene



Fields

- Fields may
 - Be indexed or not
 - Indexed fields may or may not be analyzed (i.e., tokenized with an Analyzer)
 - Non-analyzed fields view the entire value as a single token (useful for URLs, paths, dates, social security numbers, ...)
 - Be stored or not
 - Useful for fields that you'd like to display to users
 - Optionally store term vectors
 - Like a positional index on the Field's terms
 - Useful for highlighting, finding similar documents, categorization



Field construction - Lots of different constructors

```
import org.apache.lucene.document.Field
import
org.apache.lucene.document.FieldType
```

```
Field(String name,
        String value,
        FieldType type);
```

value can also be specified with a Reader, a TokenStream, or a

byte[]. FieldType specifies field properties.

Can also directly use sub-classes like TextField, StringField, ...

| Index | Store | TermVector | Example usage |
|--------------|-------|------------------------|--|
| NOT_ANALYZED | YES | NO | Identifiers, telephone/SSNs, URLs, dates, ... |
| ANALYZED | YES | WITH_POSITIONS_OFFSETS | Title, abstract |
| ANALYZED | NO | WITH_POSITIONS_OFFSETS | Body |
| NO | YES | NO | Document type, DB keys (if not used for searching) |
| NOT_ANALYZED | NO | NO | Hidden keywords |

Multi-valued fields

- You can add multiple Fields with the same name
 - Lucene simply concatenates the different values for that named Field

```
Document doc = new Document();  
doc.add(new TextField("author", "chris manning"));  
doc.add(new TextField("author", "prabhakar raghavan"));  
...
```



- Tokenizes the input text
- Common Analyzers
 - WhitespaceAnalyzer
Splits tokens on whitespace
 - SimpleAnalyzer
Splits tokens on non-letters, and then lowercases
 - StopAnalyzer
Same as SimpleAnalyzer, but also removes stop words
 - StandardAnalyzer
Most sophisticated analyzer that knows about certain token types, lowercases, removes stop words, ...

Analysis example

- “The quick brown fox jumped over the lazy dog”
- WhitespaceAnalyzer
 - [The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- SimpleAnalyzer
 - [the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dog]
- StopAnalyzer
 - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]
- StandardAnalyzer
 - [quick] [brown] [fox] [jumped] [over] [lazy] [dog]

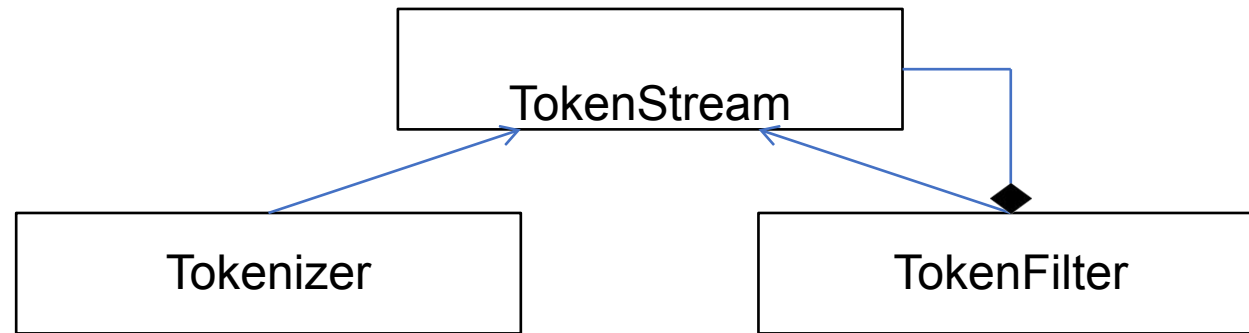
Another analysis example

- “XY&Z Corporation – xyz@example.com”
- WhitespaceAnalyzer
 - [XY&Z] [Corporation] [-] [xyz@example.com]
- SimpleAnalyzer
 - [xy] [z] [corporation] [xyz] [example] [com]
- StopAnalyzer
 - [xy] [z] [corporation] [xyz] [example] [com]
- StandardAnalyzer
 - [xy&z] [corporation] [xyz@example.com]

What's inside an Analyzer?

- Analyzers need to return a `TokenStream`

```
public TokenStream tokenStream(String fieldName, Reader reader)
```



Tokenizers and TokenFilters

- Tokenizer

- WhitespaceTokenizer
- KeywordTokenizer
- LetterTokenizer
- StandardTokenizer
- ...

- TokenFilter

- LowerCaseFilter
- StopFilter
- PorterStemFilter
- ASCIIFoldingFilter
- StandardFilter
- ...

Adding/deleting Documents to/from an IndexWriter

```
void addDocument(Iterable<IndexableField> d);
```

IndexWriter's Analyzer is used to analyze document.

Important: Need to ensure that Analyzers used at indexing time are consistent with Analyzers used at searching time

```
// deletes docs containing terms or matching
```

```
// queries. The term version is useful for
```

```
// deleting one document.
```

```
void deleteDocuments(Term... terms);
```

```
void deleteDocuments(Query... queries);
```

Index format

- Each Lucene index consists of one or more segments
 - A segment is a standalone index for a subset of documents
 - All segments are searched
 - A segment is created whenever IndexWriter flushes adds/deletes
- Periodically, IndexWriter will merge a set of segments into a single segment
 - Policy specified by a MergePolicy
- You can explicitly invoke `forceMerge()` to merge segments



Basic merge policy

- Segments are grouped into levels
- Segments within a group are roughly equal size (in log space)
- Once a level has enough segments, they are merged into a segment at the next level up



Searching a changing index

```
Directory dir = FSDirectory.open(...); DirectoryReader reader =  
DirectoryReader.open(dir);  
IndexSearcher searcher = new IndexSearcher(reader);
```

Above reader does not reflect changes to the index unless you reopen it. Reopening is more resource efficient than opening a brand new reader.

```
DirectoryReader newReader =  
    DirectoryReader.openIfChanged(reader);  
If (newReader != null) {  
    reader.close(); reader = newReader;  
    searcher = new IndexSearcher(reader);  
}
```

Near-real-time search

```
IndexWriter writer = ...;
DirectoryReader reader =
    DirectoryReader.open(writer, true);
IndexSearcher searcher = new IndexSearcher(reader);

// Now let us say there's a change to the index using writer
writer.addDocument(newDoc);

DirectoryReader newReader =
    DirectoryReader.openIfChanged(reader, writer, true);
if (newReader != null) {
    reader.close();
    reader = newReader;
    searcher = new IndexSearcher(reader);
}
```

- Constructor
 - `QueryParser(String defaultField, Analyzer analyzer);`
- Parsing methods
 - `Query parse(String query)` throws `ParseException`;
 - ... and many more



QueryParser syntax examples

| Query expression | Document matches if... |
|-----------------------------------|--|
| java | Contains the term <i>java</i> in the default field |
| java junit java OR junit | Contains the term <i>java</i> or <i>junit</i> or both in the default field (<i>the default operator can be changed to AND</i>) |
| +java +junit java AND junit | Contains both <i>java</i> and <i>junit</i> in the default field |
| title:ant | Contains the term <i>ant</i> in the title field |
| title:extreme –subject:sports | Contains <i>extreme</i> in the title and not <i>sports</i> in subject |
| (agile OR extreme) AND java | Boolean expression matches |
| title:"junit in action" | Phrase matches in title |
| title:"junit action"~5 | Proximity matches (within 5) in title |
| java* | Wildcard matches |
| java~ | Fuzzy matches |
| lastmodified:[1/1/09 TO 12/31/09] | Range matches |

- TermQuery
 - Constructed from a Term
- TermRangeQuery
- NumericRangeQuery
- PrefixQuery
- BooleanQuery
- PhraseQuery
- WildcardQuery
- FuzzyQuery
- MatchAllDocsQuery

- Methods
 - TopDocs search(Query q, int n);
 - Document doc(int docID);



TopDocs and ScoreDoc

- TopDocs methods
 - Number of documents that matched the search
totalHits
 - Array of ScoreDoc instances containing results
scoreDocs
 - Returns best score of all matches
getMaxScore()
- ScoreDoc methods
 - Document id
doc
 - Document score
score

- Original scoring function uses basic tf-idf scoring with
 - Programmable boost values for certain fields in documents
 - Length normalization
 - Boosts for documents containing more of the query terms
- IndexSearcher provides an `explain()` method that explains the scoring of a document



Lucene 5.0 Scoring

- As well as traditional tf.idf vector space model, Lucene 5.0 has:
 - BM25
 - drf (divergence from randomness)
 - ib (information (theory)-based similarity)

```
indexSearcher.setSimilarity( new  
    BM25Similarity());
```

```
BM25Similarity custom = new BM25Similarity(1.2, 0.75); // k1, b  
indexSearcher.setSimilarity(custom);
```



- Lucene: <http://lucene.apache.org>
- Lucene in Action:
<http://www.manning.com/hatcher3/>
 - Code samples available for download
- Ant: <http://ant.apache.org/>
 - Java build system used by “Lucene in Action” code



Implementation example - Importing packages and dependencies

```
package com.luceneapp;

import java.io.IOException;
import java.io.InputStream;
import java.nio.file.FileVisitResult;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;

import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.Field.Store;
import org.apache.lucene.document.LongPoint;
import org.apache.lucene.document.StringField;
import org.apache.lucene.document.TextField;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.index.IndexWriterConfig.OpenMode;
import org.apache.lucene.index.Term;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
```


Implementation example - Creating Inverted Index

```
public static void main(String[] args)
{
    //Input folder
    String docsPath = "inputFiles";

    //Output folder
    String indexPath = "indexedFiles";

    //Input Path Variable
    final Path docDir = Paths.get(docsPath);

    try
    {
        //org.apache.lucene.store.Directory instance
        Directory dir = FSDirectory.open( Paths.get(indexPath) );

        //analyzer with the default stop words
        Analyzer analyzer = new StandardAnalyzer();

        //IndexWriter Configuration
        IndexWriterConfig iwc = new IndexWriterConfig(analyzer);
        iwc.setOpenMode(OpenMode.CREATE_OR_APPEND);

        //IndexWriter writes new index files to the directory
        IndexWriter writer = new IndexWriter(dir, iwc);

        //Its recursive method to iterate all files and directories
        indexDocs(writer, docDir);

        writer.close();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

Implementation example - Creating Inverted Index

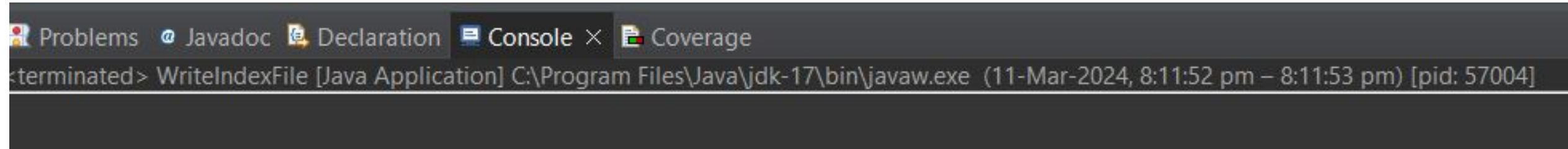
```
static void indexDocs(final IndexWriter writer, Path path) throws IOException
{
    //Directory?
    if (Files.isDirectory(path))
    {
        //Iterate directory
        Files.walkFileTree(path, new SimpleFileVisitor<Path>()
        {
            @Override
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException
            {
                try
                {
                    //Index this file
                    indexDoc(writer, file, attrs.lastModifiedTime().toMillis());
                }
                catch (IOException ioe)
                {
                    ioe.printStackTrace();
                }
                return FileVisitResult.CONTINUE;
            }
        });
    }
    else
    {
        //Index this file
        indexDoc(writer, path, Files.getLastModifiedTime(path).toMillis());
    }
}
```

```
static void indexDoc(IndexWriter writer, Path file, long lastModified) throws IOException
{
    try (InputStream stream = Files.newInputStream(file))
    {
        //Create lucene Document
        Document doc = new Document();

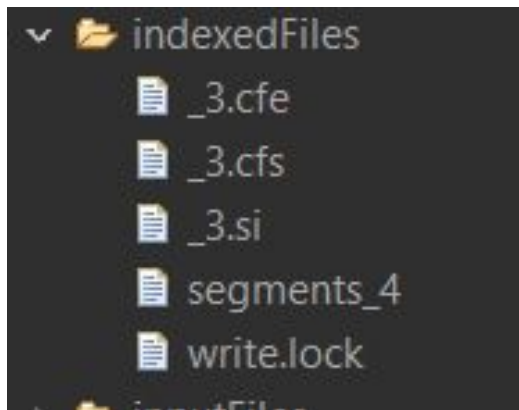
        doc.add(new StringField("path", file.toString(), Field.Store.YES));
        doc.add(new LongPoint("modified", lastModified));
        doc.add(new TextField("contents", new String(Files.readAllBytes(file)), Store.YES));

        //Updates a document by first deleting the document(s)
        //containing <code>term</code> and then adding the new
        //document. The delete and then add are atomic as seen
        //by a reader on the same index
        writer.updateDocument(new Term("path", file.toString()), doc);
    }
}
```

Implementation example - Creating Inverted Index



```
Problems  Javadoc  Declaration  Console  Coverage  
terminated> WriteIndexFile [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (11-Mar-2024, 8:11:52 pm – 8:11:53 pm) [pid: 57004]
```



.cfe = Compound file entry table
.cfs = Compound file
.si = Segment info

Implementation example - Searching

```
package com.luceneapp;

import java.io.IOException;
import java.nio.file.Paths;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
```

```
public class ReadIndexFile
{
    //directory contains the lucene indexes
    private static final String INDEX_DIR = "indexedFiles";

    public static void main(String[] args) throws Exception
    {
        //Create lucene searcher. It search over a single IndexReader.
        IndexSearcher searcher = createSearcher();

        //Search indexed contents using search term
        TopDocs foundDocs = searchInContent("India", searcher);

        //Total found documents
        System.out.println("Total Results :: " + foundDocs.totalHits);

        //Let's print out the path of files which have searched term
        for (ScoreDoc sd : foundDocs.scoreDocs)
        {
            Document d = searcher.doc(sd.doc);
            System.out.println("Path : " + d.get("path") + ", Score : " + sd.score);
        }
    }
}
```


Implementation example - Searching

```
private static TopDocs searchInContent(String textToFind, IndexSearcher searcher) throws Exception
{
    //Create search query
    QueryParser qp = new QueryParser("contents", new StandardAnalyzer());
    Query query = qp.parse(textToFind);

    //search the index
    TopDocs hits = searcher.search(query, 10);
    return hits;
}
```

```
private static IndexSearcher createSearcher() throws IOException
{
    Directory dir = FSDirectory.open(Paths.get(INDEX_DIR));

    //It is an interface for accessing a point-in-time view of a lucene index
    IndexReader reader = DirectoryReader.open(dir);

    //Index searcher
    IndexSearcher searcher = new IndexSearcher(reader);
    return searcher;
}
```

Implementation example - Results

```
Problems Javadoc Declaration Console × Coverage
<terminated> ReadIndexFile [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (11-Mar-2024, 8:22:55 pm – 8:22:55 pm) [pid: 53060]
Total Results :: 1 hits
Path : inputFiles\file4.txt, Score : 1.2647467
```



Content-based Visual Information Retrieval (CBVIR)

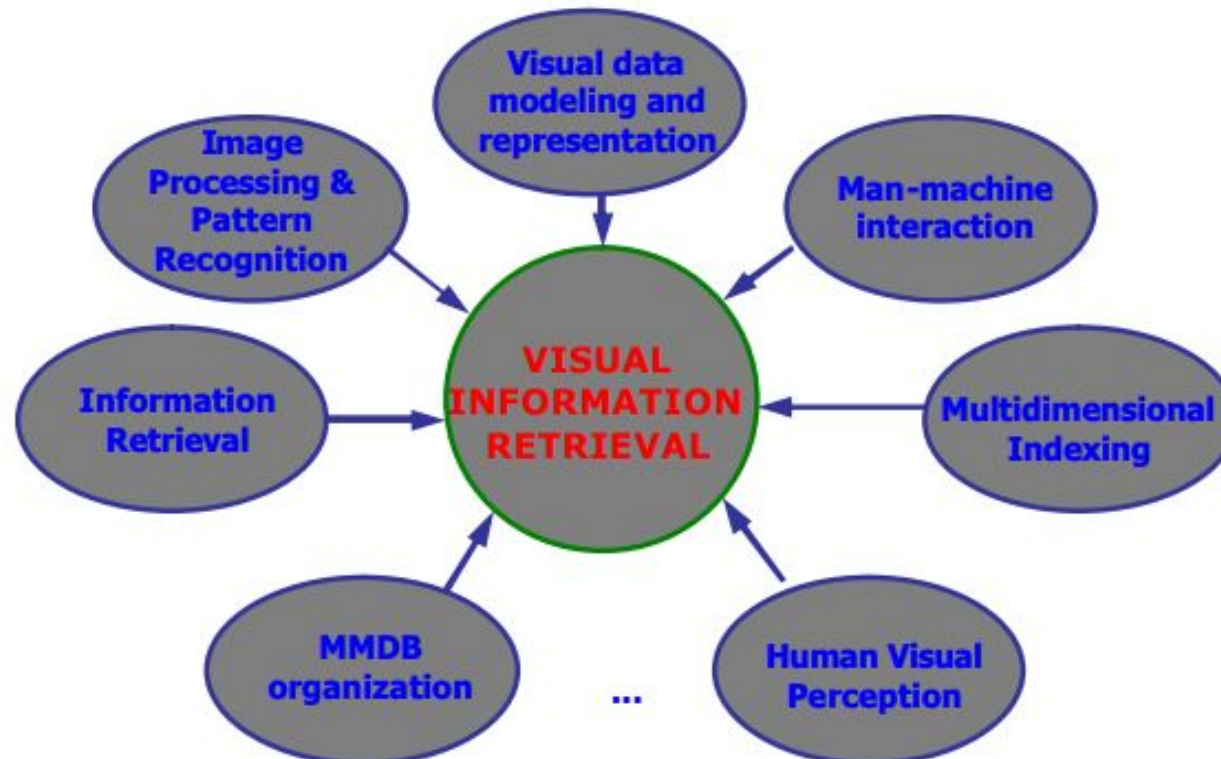
- Content-based image retrieval, also known as Query By Image Content (QBIC), is an automated technique that takes an image as a query and returns a set of images similar to the query.
- Content-based image retrieval techniques use visual contents of the images described in the form of low-level features like color, texture, shape, and spatial locations to represent and search the images in the databases.

Evolution of CBVIR

- Research in the field of Content-Based Visual Information Retrieval (CBVIR) started in the early 1990's and is likely to continue during the first decade of the 21st century.
- Many research groups in leading universities and companies are actively working in the area and a fairly large number of prototypes and commercial products are already available.

Evolution of CBVIR

- Progress in visual information retrieval has been fostered by many research fields particularly: (text -based) information retrieval, image processing and computer vision, pattern recognition, multimedia database organization, multidimensional indexing, psychological modeling of user behavior, man-machine interaction, among many others.



VIR systems can be classified in two main generations, according to the attributes used to search and retrieve a desired image or video file:

- **First-generation VIR systems:** use query by text, allowing queries such as “all pictures of red Ferraris” or “all images of Van Gogh’s paintings”. They rely strongly on metadata, which can be represented either by alphanumeric strings, keywords, or full scripts.
- **Second-generation (CB)VIR systems:** support query by content, where the notion of content, for still images, includes, in increasing level of complexity: perceptual properties (e.g., color, shape, texture), semantic primitives (abstractions such as objects, roles, and scenes), and subjective attributes such as impressions, emotions and meaning associated to the perceptual properties. Many second-generation systems use content-based techniques as a complementary component, rather than a replacement, of text-based tools.

Overview of how CBVIR works



- The system retrieves similar images when an example image or sketch is presented as input to the system.
- The query image is converted into the internal representation of the feature vector using the same feature extraction routine that was used for building the feature database.
- The similarity measure is employed to calculate the distance between the feature vectors of the query image and those of the target images in the feature database.

ALGORITHMS FOR INFORMATION RETRIEVAL AND INTELLIGENT WEB

Overview of how CBVIR works

- Finally, the retrieval is performed using an indexing scheme which facilitates the efficient searching of the image database.
- Therefore, when similarity measurement is performed based on image features, the output set achieves a high retrieval performance.

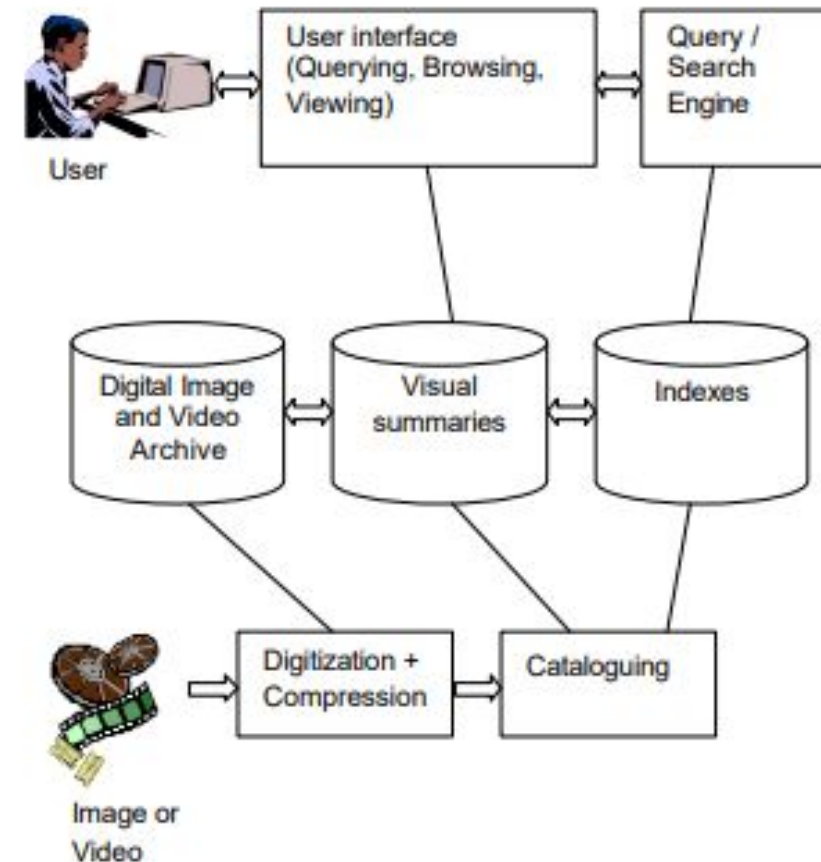


Figure 2. Block diagram of a CBVIR system.

CBVIR has several advantages over traditional text-based image retrieval.

- Using the visual contents of the query image in CBVIR, it is a more efficient and effective way of finding relevant images than searching based on text annotations.
- It is more close to human perception of visual data. Also, CBIR does not consume the time in the manual annotation process as in the text-based approach.
- User-relevant feedback can also be incorporated to further improve the retrieval process to produce perceptually and semantically more meaningful retrieval results

Typical Architecture of a CBVIR system

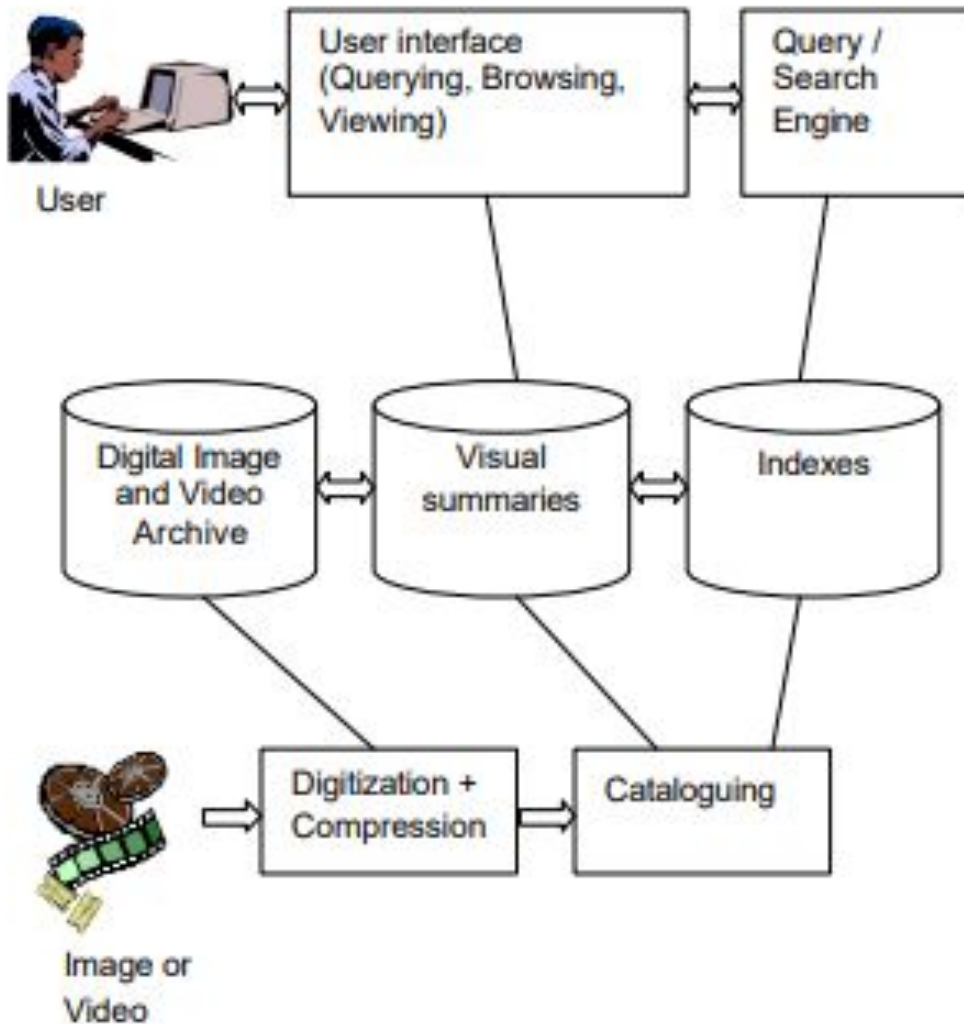


Figure 2. Block diagram of a CBVIR system.

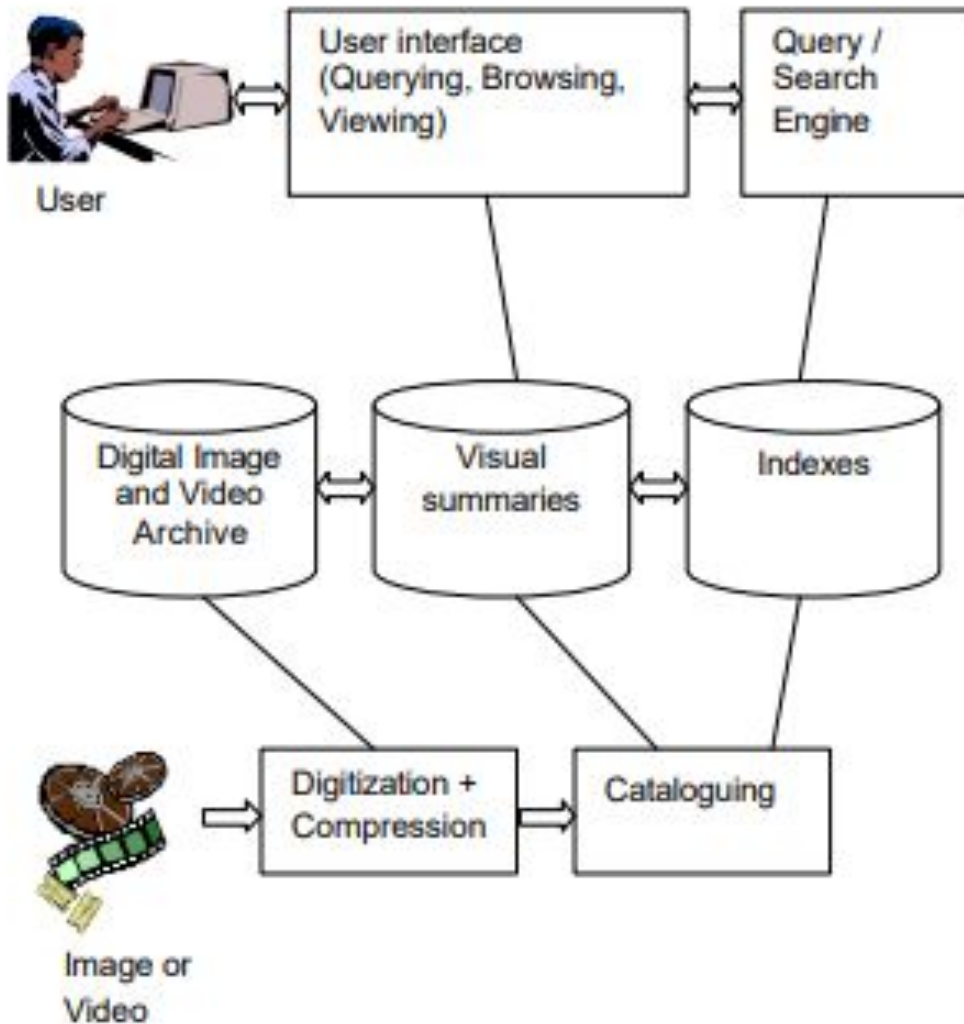
User interface: friendly GUI that allows the user to interactively query the database, browse the results, and view the selected images / video clips.

Query / search engine: responsible for searching the database according to the parameters provided by the user

Digital image and video archive : repository of digitized, compressed images and video clips.

Visual summaries: representation of image and video contents in a concise way, such as thumbnails for images or keyframes for video sequences.

Typical Architecture of a CBVIR system



Indexes: pointers to images or video segments.

Digitization and compression: hardware and software necessary to convert images and videos into digital compressed format.

Cataloguing: process of extracting features from the raw images and videos and building the corresponding indexes.

Figure 2. Block diagram of a CBVIR system.

References

- “Content-Based Visual Information Retrieval” - Oge Marques and Borko Furht



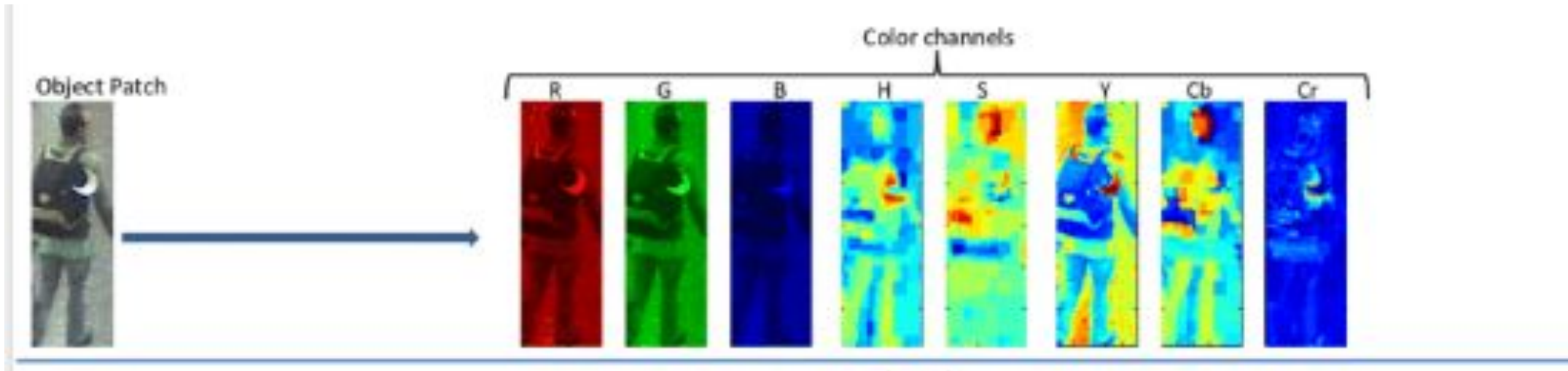


CBVIR systems should be able to automatically extract visual features that are used to describe the contents of an image or video clip. Examples of such features include color, texture, size, shape, and motion information.

We will keep our discussion limited to features such as Color, Texture, Shape and Spatial Information.

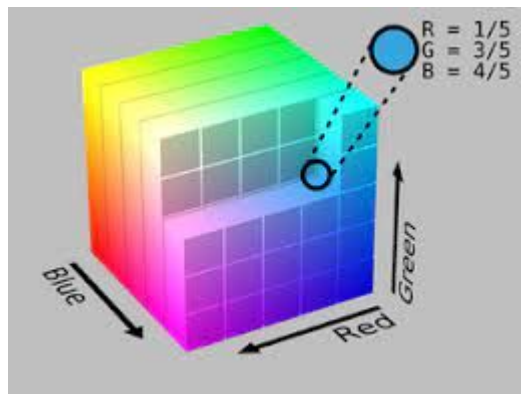
Color Feature

- Color is the most extensively used visual content for image retrieval.
- Its three-dimensional values make its discrimination potentiality superior to the single-dimensional gray values of images.
- Before selecting an appropriate color description, color space must be determined first

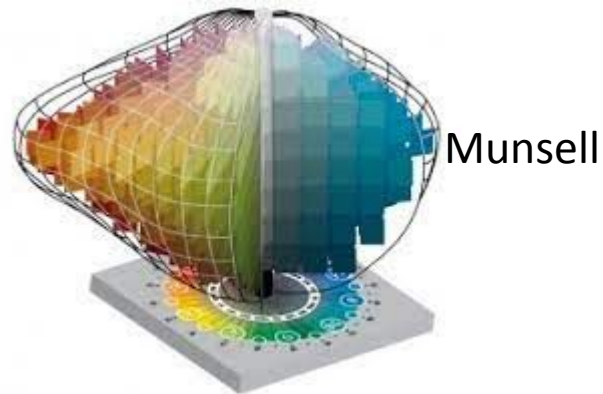


- **Color space:**

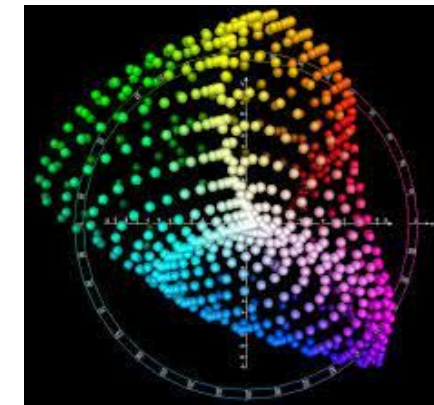
- Each pixel of the image can be represented as a point in a 3D color space.
- Commonly used color space for image retrieval includes RGB, Munsell, CIE $L^*a^*b^*$, CIE $L^*u^*v^*$, HSV (or HSL, HSB), and opponent color space.



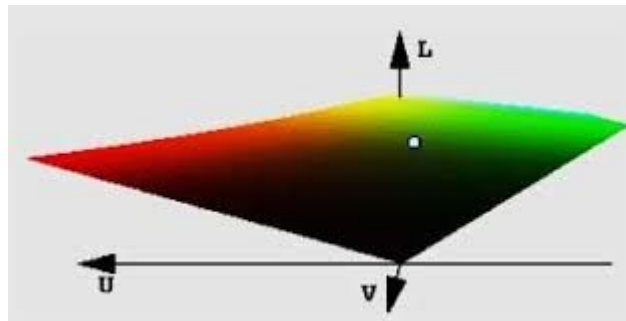
RGB



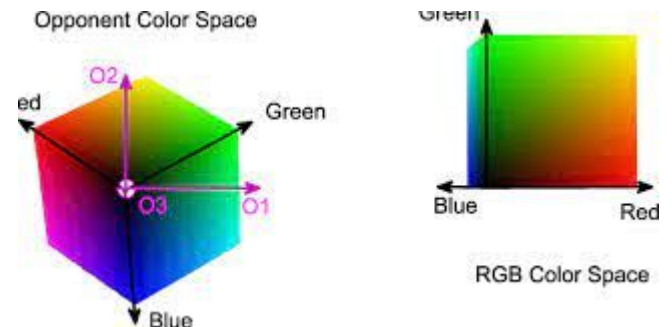
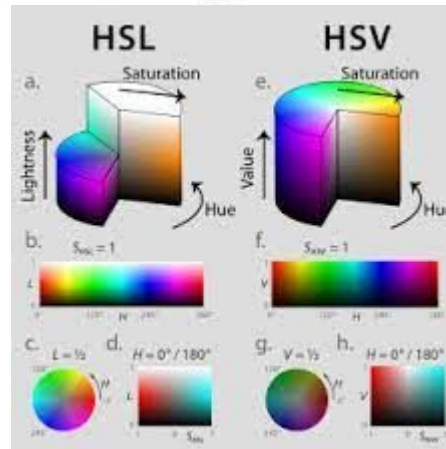
Munsell



CIE $L^*a^*b^*$



CIE $L^*u^*v^*$

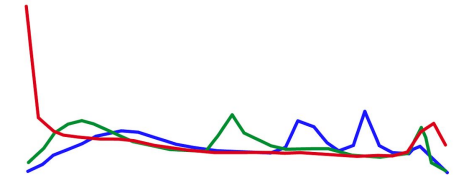


opponent color space.

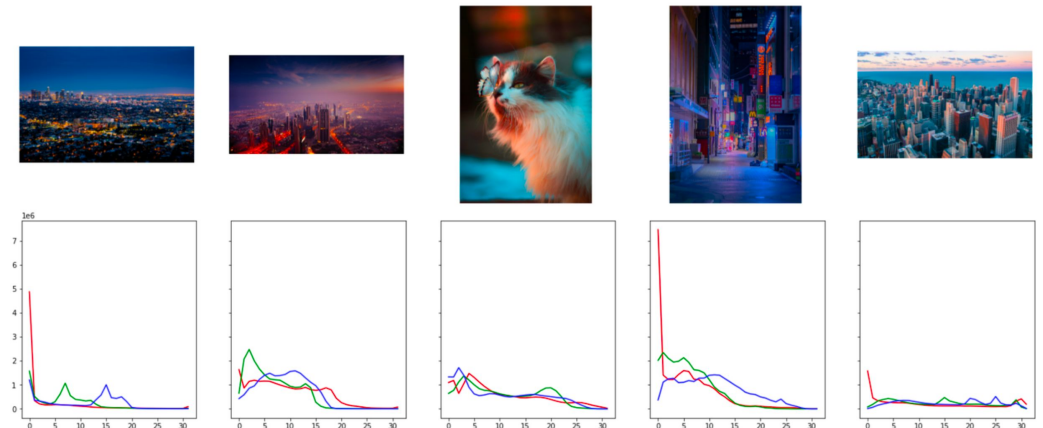
- **Color Descriptors:**
 - A color descriptor is a generalization of a color histogram that captures some spatial characteristics of the color distribution in an image. Color descriptors are essential for accessing reliable visual information as illumination variation is inevitable in many practical cases.
 - Some commonly used color descriptors are as follows: the color moments, color histogram, color coherence vector, and color correlogram.

- **Color Descriptors:**
 - **Color Moments:** Color moments have been successfully used in many retrieval systems like QBIC, especially when the image contains only objects.
 - **Color Histogram:** The color histogram serves as an effective representation of the color content of an image if the color pattern is unique compared with the rest of the dataset. The color histogram is easy to compute and effective in characterizing both the global and local distributions of colors in an image

Query image:

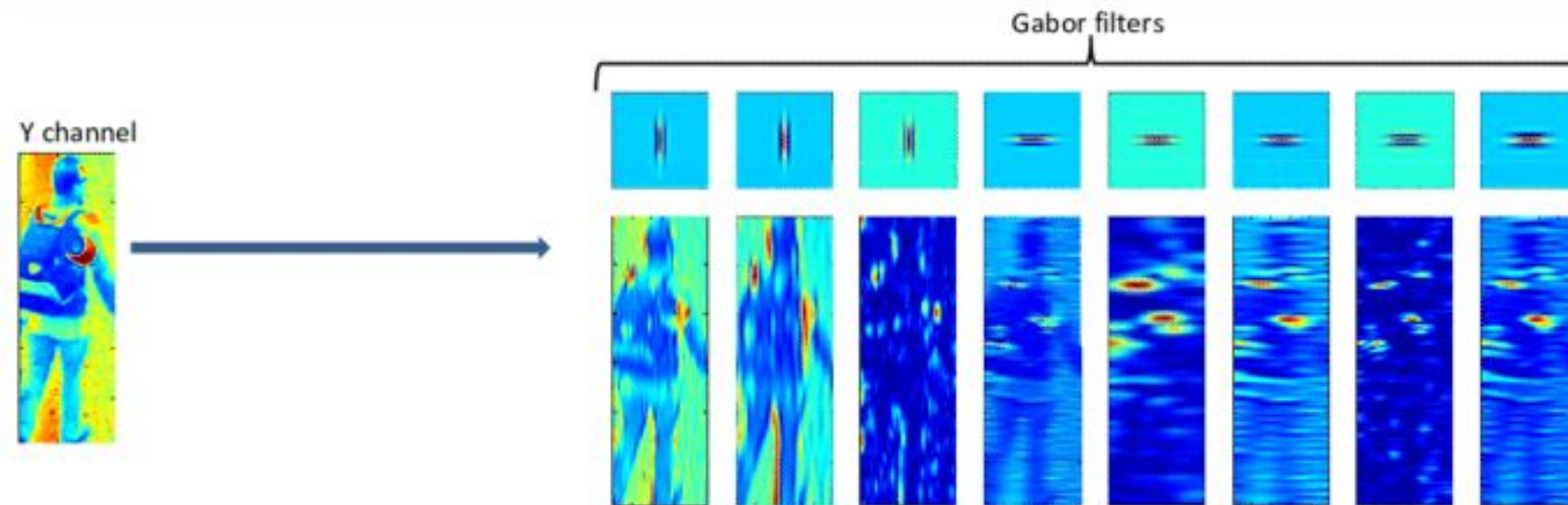


Results:



Texture Feature

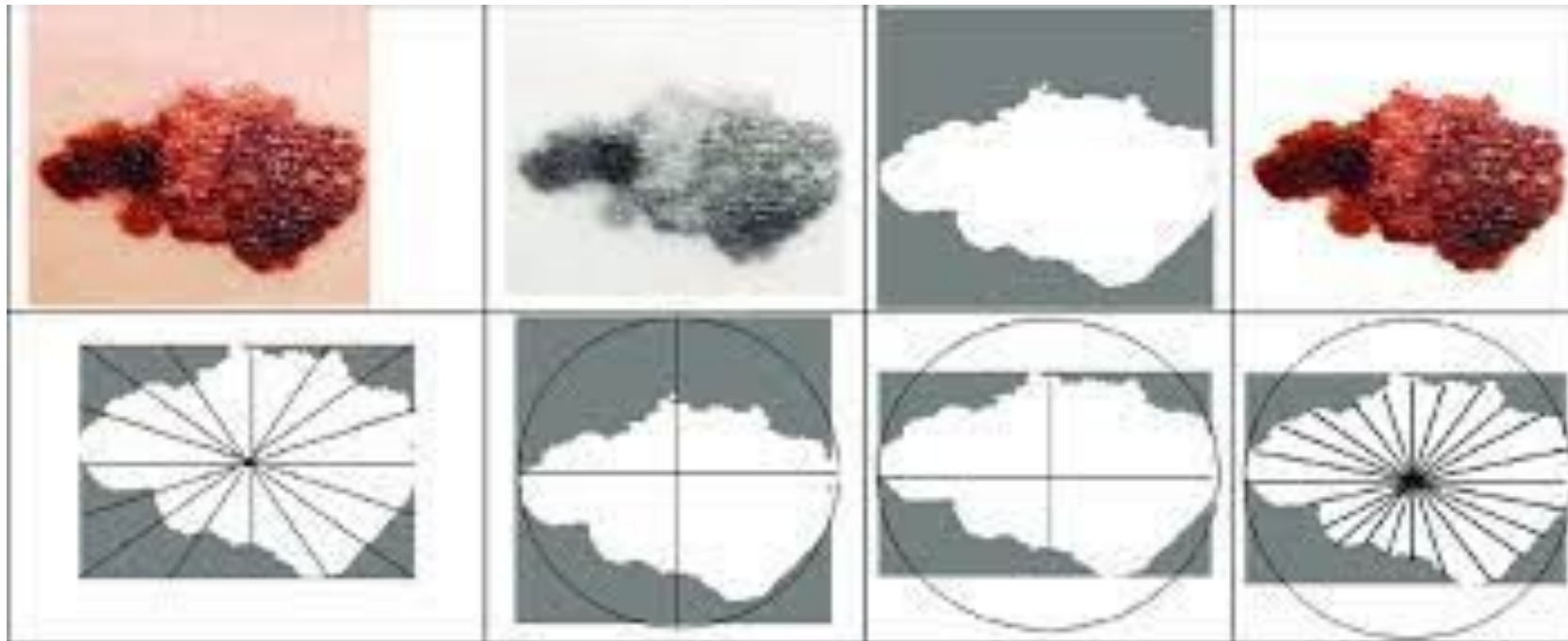
- Texture is another important property of images.
- Various texture representations have been investigated in pattern recognition and computer vision.



- Basically, texture representation methods can be classified into two categories: structural and statistical.
 - Structural methods, including morphological operator and adjacency graph, describe texture by identifying structural primitives and their placement rules. They tend to be most effective when applied to textures that are very regular.
 - Statistical methods, including Fourier power spectra, cooccurrence matrices, shift-invariant principal component analysis (SPCA), Tamura feature, Wold decomposition, Markov random field, fractal model, and multiresolution filtering techniques such as Gabor and wavelet transform, characterize texture by the statistical distribution of the image intensity

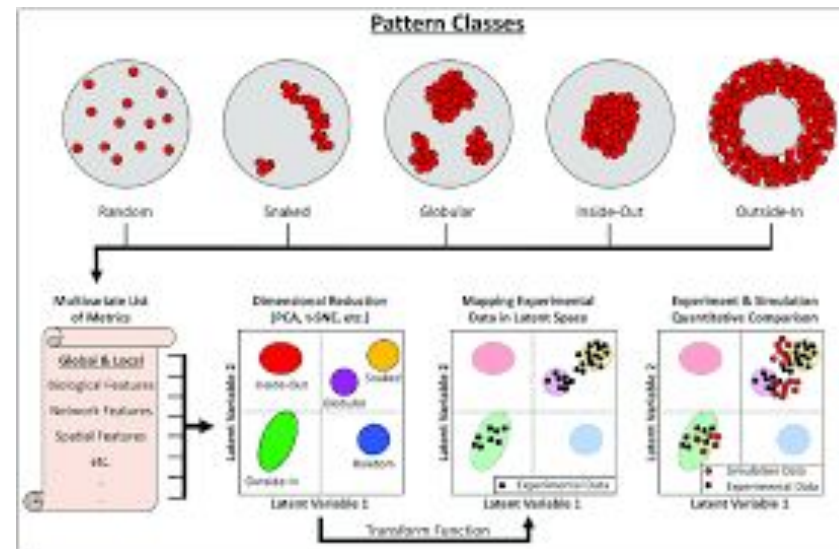
Shape Feature

- Shape features of objects or regions have been used in many content-based image retrieval systems.
- Compared with color and texture features, shape features are usually described after images have been segmented into regions or objects.



- Since robust and accurate image segmentation is difficult to achieve, the use of shape features for image retrieval has been limited to special applications where objects or regions are readily available.
- The state-of-the-art methods for shape description can be categorized into either boundary-based (e.g., rectilinear shapes, polygonal approximation, finite element models, and Fourier-based shape descriptors) or region-based methods (e.g., statistical moments).

- Regions or objects with similar color and texture properties can be easily distinguished by imposing spatial constraints.
- For instance, regions of blue sky and ocean may have similar color histograms, but their spatial locations in images are different. Therefore, the spatial location of regions (or objects) or the spatial relationship between multiple regions (or objects) in an image is very useful for searching images.



References

- “Content-Based Image Retrieval : Ideas, Influences, and Current Trends” - Vipin Tyagi, Springer Nature Singapore Pte Ltd., 2017.