Architecture of Enterprise Applications V Searching

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

Agenda

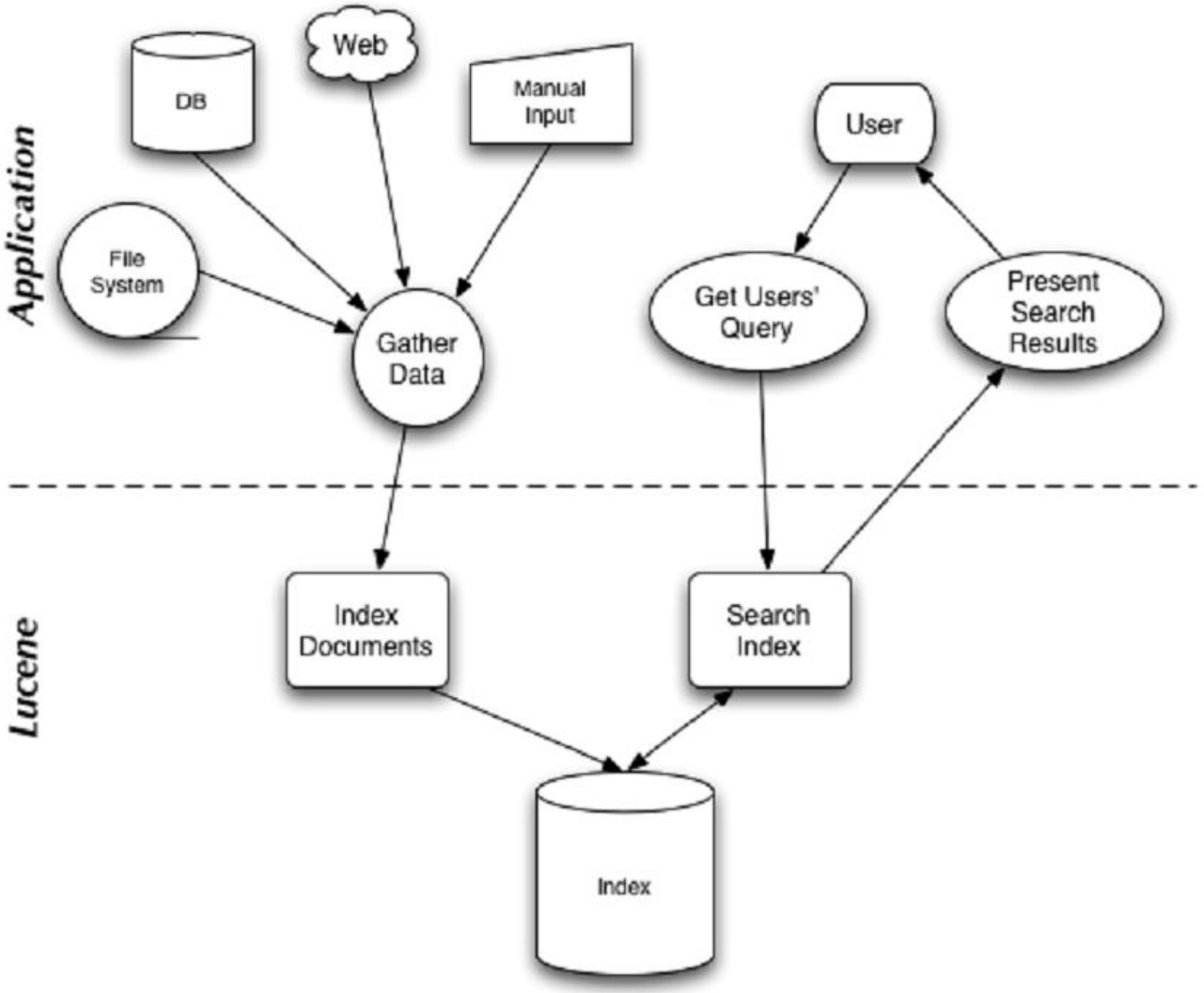
- Searching
  - Lucene
  - Apache Solr
  - Percolator

Lucene

- Lucene is a high performance, scalable Information Retrieval (IR) library.
  - It lets you add indexing and searching capabilities to your applications.
  - Lucene is a mature, free, open-source project implemented in Java.
  - it's a member of the popular Apache Jakarta family of projects, licensed under the liberal Apache Software License.

- Lucene provides a simple yet powerful core API
  - that requires minimal understanding of full-text indexing and searching.

Lucene


Application
DB
Web
Manual Input
File System
Gather Data
User
Get Users' Query
Present Search Results
Lucene
Index Documents
Search Index
Index

Indexing

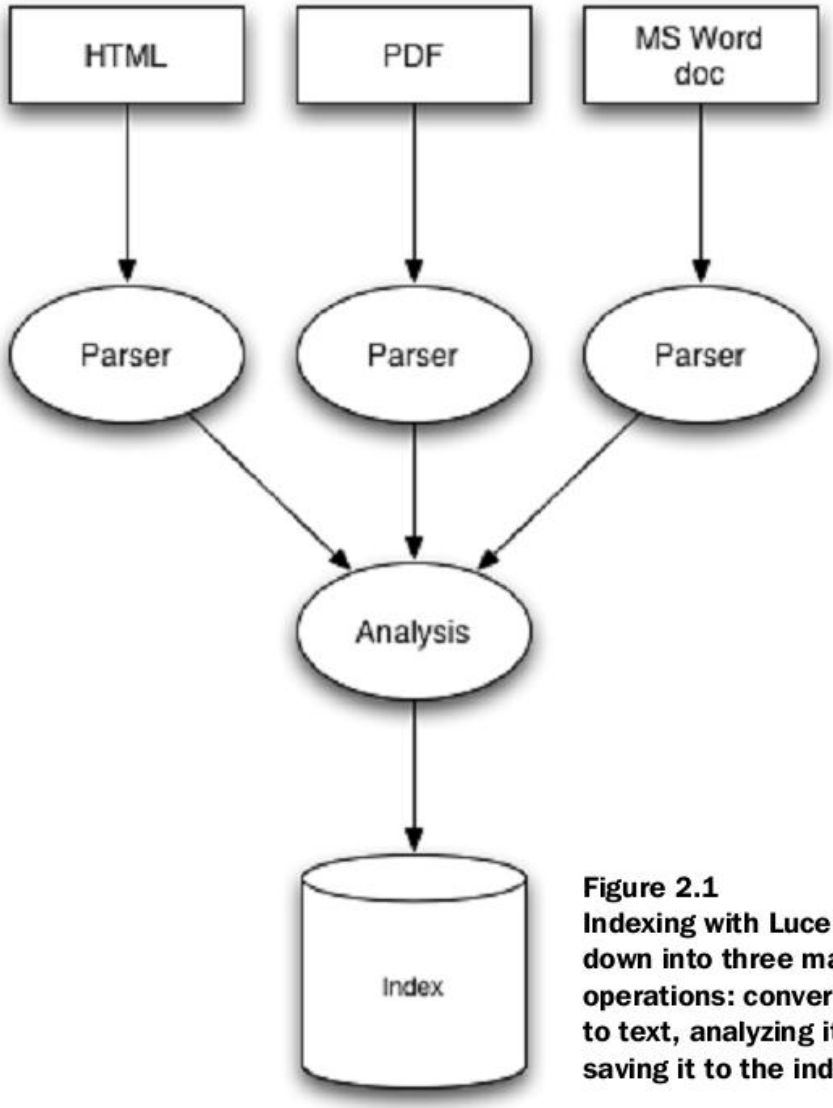


Figure 2.1
Indexing with Lucene breaks down into three main operations: converting data to text, analyzing it, and saving it to the index.

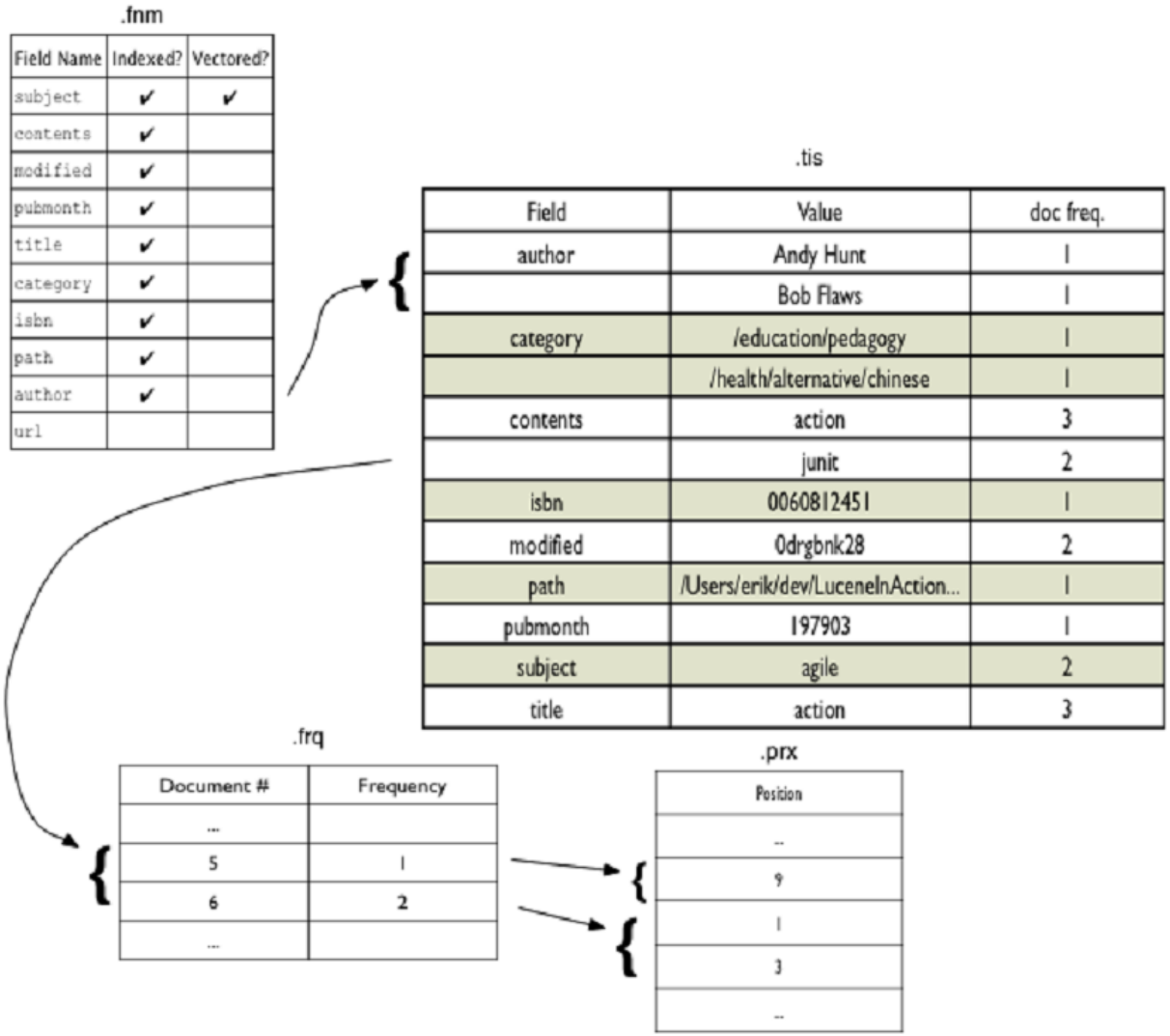
indexing

- At the heart of all search engines is the concept of indexing:
  - processing the original data into a highly efficient cross-reference lookup in order to facilitate rapid searching.

- Suppose you needed to search a large number of files, and you wanted to be able to find files that contained a certain word or a phrase
  - A naïve approach would be to sequentially scan each file for the given word or phrase.
  - This approach has a number of flaws, the most obvious of which is that it doesn't scale to larger file sets or cases where files are very large.

- This is where indexing comes in:
  - To search large amounts of text quickly, you must first index that text and convert it into a format that will let you search it rapidly, eliminating the slow sequential scanning process.
  - This conversion process is called indexing, and its output is called an index.
  - You can think of an index as a data structure that allows fast random access to words stored inside it.

Inverting index



searching

- Searching is the process of looking up words in an index to find documents where they appear.

- The quality of a search is typically described using precision and recall metrics.
  - Recall measures how well the search system finds relevant documents, whereas precision measures how well the system filters out the irrelevant documents.

- A number of other factors
  - speed and the ability to quickly search large quantities of text.
  - Support for single and multi term queries, phrase queries, wildcards, result ranking, and sorting are also important, as is a friendly syntax for entering those queries.

A sample application

- Suppose you need to index and search files stored in a directory tree, not just in a single directory
- These example applications will familiarize you with Lucene's API, its ease of use, and its power.
- The code listings are complete, ready-to-use command-line programs.

Creating an Index


```
/**
 * This code was originally written for
 * Erik's Lucene intro java.net article
 */
public class Indexer {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      throw new Exception("Usage: java " + Indexer.class.getName()
        + " <index dir> <data dir>");
    }
    File indexDir = new File(args[0]);
    File dataDir = new File(args[1]);

    long start = new Date().getTime();
    int numIndexed = index(indexDir, dataDir);
    long end = new Date().getTime();

    System.out.println("Indexing " + numIndexed + " files took " + (end - start) + " milliseconds");
  }
```

Create Lucene index in this directory

Index files in this directory

Creating an Index


```
// open an index and start file directory traversal
 public static int index(File indexDir, File dataDir) throws IOException {
  if (!dataDir.exists() || !dataDir.isDirectory()) {
    throw new IOException(dataDir
      + " does not exist or is not a directory");
  }

  IndexWriter writer = new IndexWriter(indexDir,
      new StandardAnalyzer(), true);
  writer.setUseCompoundFile(false);

  indexDirectory(writer, dataDir);

  int numIndexed = writer.docCount();
  writer.optimize();
  writer.close();
  return numIndexed;
}
```

Create Lucene index

Close index

Creating an Index


```
// recursive method that calls itself when it finds a directory
private static void indexDirectory(IndexWriter writer, File dir)
  throws IOException {

  File[] files = dir.listFiles();

  for (int i = 0; i < files.length; i++) {
    File f = files[i];
    if (f.isDirectory()) {
      indexDirectory(writer, f);
    } else if (f.getName().endsWith(".txt")) {
        indexFile(writer, f);
    }
  }
}
```

recurse

Index .txt files only

Creating an Index


```
// method to actually index a file using Lucene
  private static void indexFile(IndexWriter writer, File f)
    throws IOException {

    if (f.isHidden() || !f.exists() || !f.canRead()) {
      return;
    }

    System.out.println("Indexing " + f.getCanonicalPath());

    Document doc = new Document();
    doc.add(Field.Text("contents", new FileReader(f)));

    doc.add(Field.Keyword("filename", f.getCanonicalPath()));
    writer.addDocument(doc);
  }
}
```

← Index file content

← Index file name

← Add document to Lucene index

Running Indexer


```
% java lia.meetlucene.Indexer build/index/lucene
Indexing /lucene/build/test/TestDoc/test.txt
Indexing /lucene/build/test/TestDoc/test2.txt
Indexing /lucene/BUILD.txt
Indexing /lucene/CHANGES.txt
Indexing /lucene/LICENSE.txt
Indexing /lucene/README.txt
Indexing /lucene/src/jsp/README.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/stemsUnicode.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/test1251.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/testKOI8.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/testUnicode.txt
Indexing /lucene/src/test/org/apache/lucene/analysis/ru/wordsUnicode.txt
Indexing /lucene/todo.txt
Indexing 13 files took 2205 milliseconds
```

Searching an index


```
/**
 * This code was originally written for
 * Erik's Lucene intro java.net article
 */
public class Searcher {

 public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    throw new Exception("Usage: java " + Searcher.class.getName()
      + " <index dir> <query>");
  }
  File indexDir = new File(args[0]);
  String q = args[1];

  if (!indexDir.exists() || !indexDir.isDirectory()) {
   throw new Exception(indexDir +
     " does not exist or is not a directory.");
  }

  search(indexDir, q);
 }
```

Index directory created by Indexer

Query string

Searching an index


```
public static void search(File indexDir, String q)
    throws Exception {
    Directory fsDir = FSDirectory.getDirectory(indexDir, false);
    IndexSearcher is = new IndexSearcher(fsDir);

    Query query = QueryParser.parse(q, "contents",   new StandardAnalyzer());
    long start = new Date().getTime();
    Hits hits = is.search(query);
    long end = new Date().getTime();

    System.err.println("Found " + hits.length() +  " document(s) (in " + (end - start) +
                       " milliseconds) that matched query '" + q + "':");

    for (int i = 0; i < hits.length(); i++) {
      Document doc = hits.doc(i);
      System.out.println(doc.get("filename"));
    }
  }
}
```

Open Index

Parse query

Search Index

Write search stats

Retrieve matching document

Display filename

Running Searcher


```
%java lia.meetlucene.Searcher build/index 'lucene'
Found 6 document(s) (in 66 milliseconds) that matched query 'lucene':
/lucene/README.txt
/lucene/src/jsp/README.txt
/lucene/BUILD.txt
/lucene/todo.txt
/lucene/LICENSE.txt
/lucene/CHANGES.txt
```

Core indexing classes

- IndexWriter
  - This class creates a new index and adds documents to an existing index.
- Directory
  - The Directory class represents the location of a Lucene index.
- Analyzer
  - The Analyzer, specified in the IndexWriter constructor, is in charge of extracting tokens out of text to be indexed and eliminating the rest.
- Document
  - A Document represents a collection of fields.
- Field
  - Each field corresponds to a piece of data that is either queried against or retrieved from the index during search.

Core searching classes

- IndexSearcher
  - IndexSearcher is to searching what IndexWriteris to indexing
- Term
  - A Term is the basic unit for searching.
- Query
  - Query is the common, abstract parent class. It contains several utility methods
- TermQuery
  - TermQuery is the most basic type of query supported by Lucene, and it's one of the primitive query types.
- Hits
  - The Hits class is a simple container of pointers to ranked search results

Apache Solr

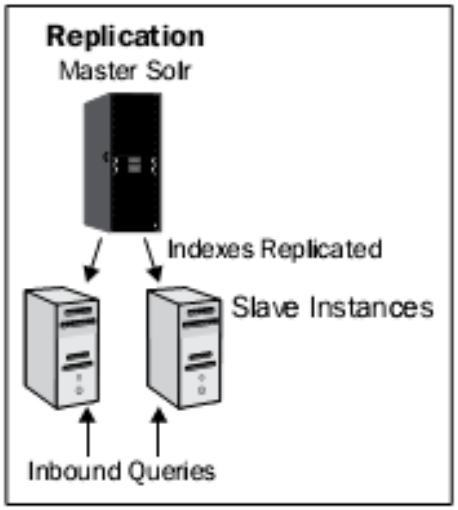
- Solr is the popular, blazing fast open source enterprise search platform from the Apache Lucene project.

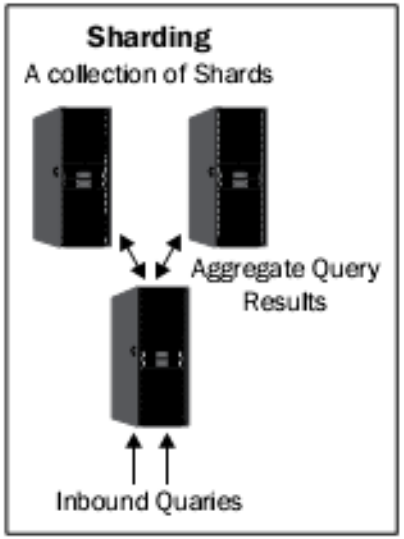
- Its major features include powerful full-text search, hit highlighting, faceted search, dynamic clustering, database integration, rich document (e.g., Word, PDF) handling, and geospatial search.

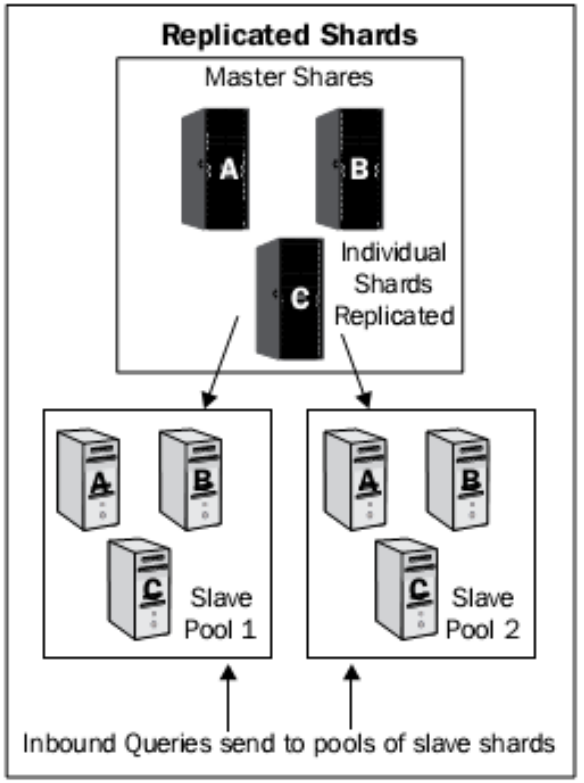
Apache Solr

- Advanced Full-Text Search Capabilities
- Optimized for High Volume Web Traffic
- Standards Based Open Interfaces - XML,JSON and HTTP
- Comprehensive HTML Administration Interfaces
- Server statistics exposed over JMX for monitoring
- Scalability - Efficient Replication to other Solr Search Servers
- Flexible and Adaptable with XML configuration
- Extensible Plugin Architecture

Apache Solr

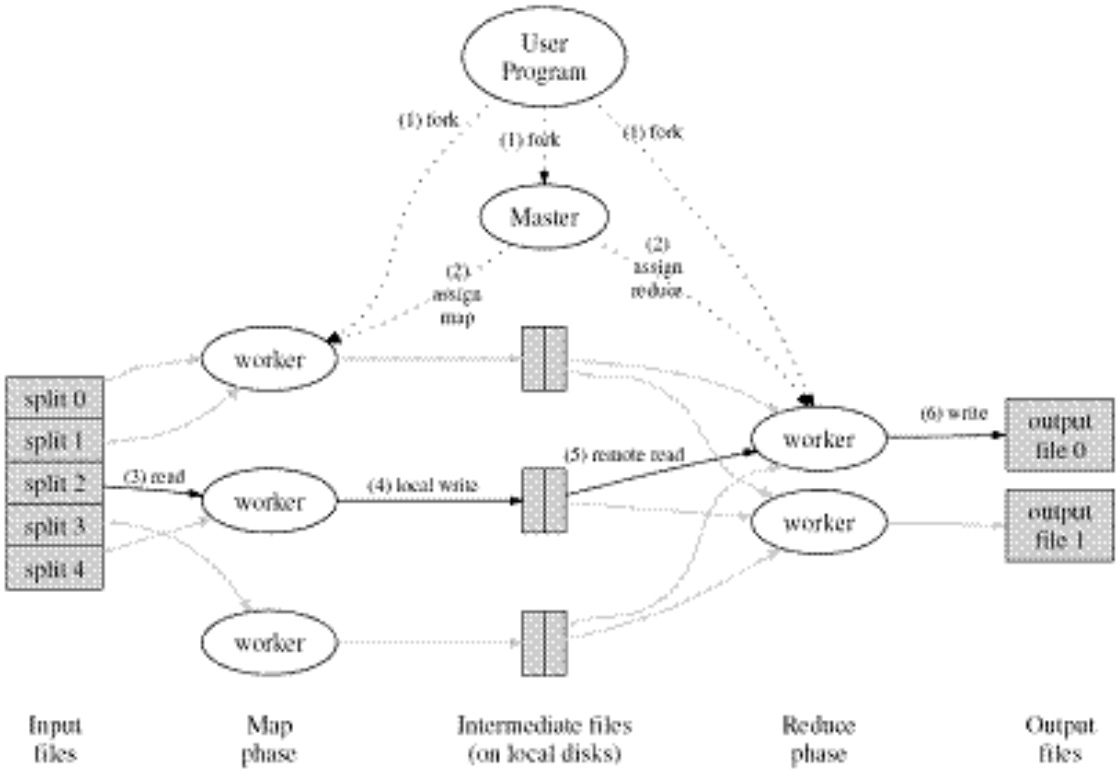

Replication
Master Solr
Indexes Replicated
Slave Instances
Inbound Queries


Sharding
A collection of Shards
Aggregate Query Results
Inbound Quaries


Replicated Shards
Master Shares
A
B
C
Individual Shards Replicated
A
B
C
Slave Pool 1
A
B
C
Slave Pool 2
Inbound Queries send to pools of slave shards

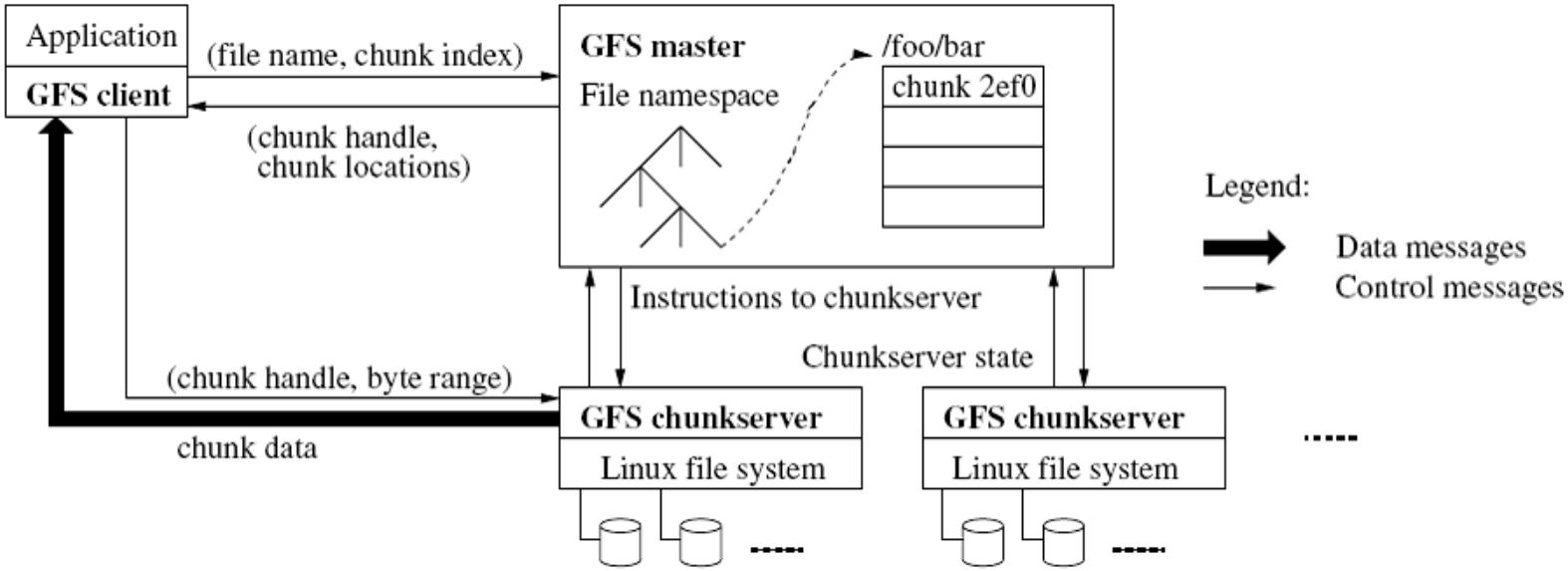
Core techniques of cloud computing

- Google
- MapReduce
  - parallelizes the computation, distributes the data, and handles failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.



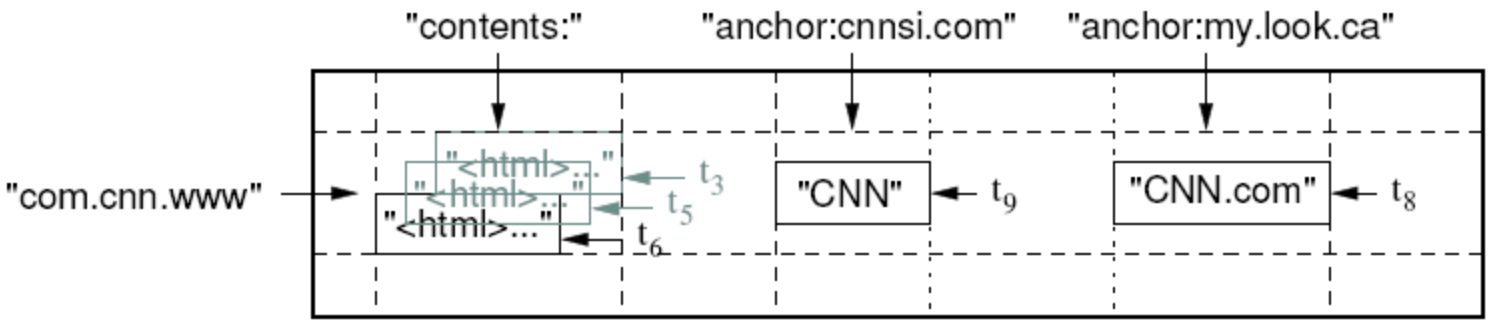
Core techniques of cloud computing

- Google
- Distributed Google File System
  - Google File System(GFS) to meet the rapidly growing demands of Google's data processing needs.



Core techniques of cloud computing

- Google
- Bigtable
  - Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers.



Google Percolator

- Daniel Peng and Frank Dabek
  - OSDI 2010

- Percolator is a system for incrementally processing updates to a large data set, and deployed it to create the Google web search index.

- Problem
  - Existing DBMSs can't handle the sheer volume of data: Google's indexing system stores tens of petabytes across thousands of machines
  - Given that the system will be processing many small updates concurrently, and ideal system would also provide mechanisms for maintaining invariants despite concurrent updates and for keeping track of which updates have been processed.

Google Percolator

- Percolator
  - Provides the user with random access to a multi-PB repository.
  - Random access allows us to process documents individually, avoiding the global scans of the repository that MapReduce requires.
  - To achieve high throughput, many threads on many machines need to transform the repository concurrently, so Percolator provides ACID-compliant transactions to make it easier for programmers to reason about the state of the repository.
  - Percolator provides observers: pieces of code that are invoked by the system whenever a user-specified column changes.

Google Percolator

- Design
  - ACID transaction over a random-access repository
  - Observers, a way to organize an incremental computation

- A Percolator system consists of three binaries that run on every machine in the cluster:
  - A Percolator worker
  - A Bigtable tablet server
  - and a GFS chunk server
  - All observers are linked into the Percolator worker

- The system also depends on two small services:
  - the timestamp oracle
  - and the lightweight lock service.

Google Percolator

- Two phases of commit

Column	*Use*
c:lock	An uncommitted transaction is writing this cell; contains the location of primary lock
c:write	Committed data present; stores the Bigtable timestamp of the data
c:data	Stores the data itself
c:notify	Hint: observers may need to run
c:ack_O	Observer "O" has run ; stores start timestamp of successful last run

Google Percolator

- Two phases of commit

key	*bal:data*	*bal:lock*	*bal:write*
Bob	*6:* *5:* $10	*6:* *5:*	*6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

1. Initial state: Joe's account contains $2 dollars, Bob's $10.

Google Percolator

- Two phases of commit

Bob	*7:* **$3** *6:* *5:* $10	*7:* **I am primary** *6:* *5:*	*7:* *6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

2. The transfer transaction begins by locking Bob's account balance by writing the lock column. This lock is the primary for the transaction. The transaction also writes data at its start timestamp, 7.

Google Percolator

- Two phases of commit

Bob	*7:* $3 *6:* *5:* $10	*7:* I am primary *6:* *5:*	*7:* *6:* data @ 5 *5:*
Joe	***7:* $9** *6:* *5:* $2	***7:* primary @ Bob.bal** *6:* *5:*	*7:* *6:* data @ 5 *5:*

3. The transaction now locks Joe's account and writes Joe's new balance (again, at the start timestamp). The lock is a secondary for the transaction and contains a reference to the primary lock (stored in row "Bob," column "bal"); in case this lock is stranded due to a crash, a transaction that wishes to clean up the lock needs the location of the primary to synchronize the cleanup.

Google Percolator

- Two phases of commit

Bob	*8:* *7:* $3 *6:* *5:* $10	*8:* ***7:*** *6:* *5:*	*8:* **data @ 7** *7:* *6:* data @ 5 *5:*
Joe	*7:* $9 *6:* *5:* $2	*7:* primary @ Bob.bal *6:* *5:*	*7:* *6:*data @ 5 *5:*

4. The transaction has now reached the commit point: it erases the primary lock and replaces it with a write record at a new timestamp (called the commit timestamp): 8. The write record contains a pointer to the timestamp where the data is stored. Future readers of the column "bal" in row "Bob" will now see the value $3.

Google Percolator

- Two phases of commit

Bob	*8:* *7:* $3 *6:* *5:* $10	*8:* *7:* *6:* *5:*	*8:* data @ 7 *7:* *6:* data @ 5 *5:*
Joe	*8:* *7:* $9 *6:* *5:*$2	*8:* ***7:*** *6:* *5:*	***8:* data @ 7** *7:* *6:* data @ 5 *5:*

5. The transaction completes by adding write records and deleting locks at the secondary cells. In this case, there is only one secondary: Joe.

References

- Otis Gospodnetic, Erik Hatcher, *Lucene in Action*, MANNING
- http://lucene.apache.org/solr/
- Rafal Kuc, Apache Solr 3.1 Cookbook, PACKT
- MapReduce: Simplied Data Processing on Large Clusters
  - Jeffrey Dean and Sanjay Ghemawat
  - OSDI 2004
- The Google File System
  - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
  - SOSP 2003
- Bigtable: A Distributed Storage System for Structured Data
  - Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
  - OSDI 2006
- Large-scale Incremental Processing Using Distributed Transactions and Notifications
  - Daniel Peng, Frank Dabek,
  - OSDI 2010

Thank You!