Object pool pattern

• Problem:
  – Expensive to create objects (allocation, initialization)
  – Expensive to destroy objects (cleanup, GC)
  – Few objects are in use at any one time

• Examples: connections, threads, memory, fonts

• Solution: re-use objects
  – Obtain objects from the pool
    • Re-initialize some fields
    • What if the pool is empty?
      – Err
      – Create and add to the pool
      – Wait for resources to become available
  – Return them to the pool when done (empty some fields)
Null object pattern

• Problem: null pointer errors
• myMap.get(key).doSomething()
• Solutions:
  – Suffer a crash at run time
  – Test return value before use
  – Statically prove correctness
  – Make doSomething work on null values
  – Return a special value for which doSomething is a no-op (null object pattern)
Memento pattern

• Representation of previous state
• Permits undo or redo
• Examples:
  – seed in a pseudo-random number generator
  – state in a FSM
• Issues:
  – efficient representation
  – undoability of undo
  – how does your DVCS handle this?
The World Wide Web: Stateful connections or not?
Word processor data structures

• Represent the text and formatting of the document

• Goals:
  – fast lookup (char at a location)
  – fast insertion/deletion
  – supports multi-level undo
  – scales to large documents
Linked list of pieces

• Linked list of text fragments

  ![Linked list of text fragments diagram]

• User operations:
  – insert
  – delete
  – move to new location
  – search

• Additional data structure operations:
  – split, merge

• How to support undo?
Piece tree

Size of left subtree

Size of right subtree
Piece table

• List of pieces
  – Each piece is part of original file, or an addition
• Pieces are added at end of a buffer (fast)
• No mutation or copying of text data structures
• Originally used in the Bravo editor (Lampson & Simonyi)
MapReduce

• Goal: process large amounts of data
  – parallelism
  – fault recovery
  – simple programming model

• Previous approaches:
  – Databases (including parallel databases)
  – Ad hoc programs
MapReduce architecture

- map(k1, v1) -> list of <k2, v2>
- reduce(k2, list of v2) -> list of v3
Count each word in a corpus (a set of documents)

map(Document key, String text):
   for each word w in text:
      EmitIntermediate(w, "1")

reduce(String word, Iterator values):
   int result = 0
   for each v in values:
      result += toInt(v)
   Emit(new Pair(word, toString(result)))
Distributed grep

- Input: corpus (set of documents)
- Output: lines matching a given pattern
- map: emits a line if it matches the pattern
- reduce: identity function that just copies the supplied intermediate data to the output.
URL Access Frequency

• Input: logs of web page requests
• Output: for each webpage, number of accesses
• map: outputs $\langle$URL, 1$\rangle$
• reduce: adds together all values for the same URL and emits a $\langle$URL, total count$\rangle$ pair
Reverse Web-Link Graph

• Input: Set of webpages
• Output: For each URL, webpages that link to it
• map: outputs <target, source> pairs for each link to a target URL found in a page named "source"
• reduce: concatenates the list of all source URLs associated with a given target URL and emits the pair: <target, list(source)>
Inverted index

• Input: corpus (set of documents)
• Output: For each word, a list of documents in which it appears
• map: parses each document, and emits a sequence of <word, document ID> pairs
• reduce: accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair.
  – Optionally, keep track of word positions.
Anagram generator

• Input: list of words
• Output: list of all possible anagrams
• map: outputs <word with letters sorted, original word>
• reduce(sortedWord, Iterator realWords):
  for each realWord:
    output <realWord, realWords>