A high-performance multithreaded approach for clustering a stream of documents

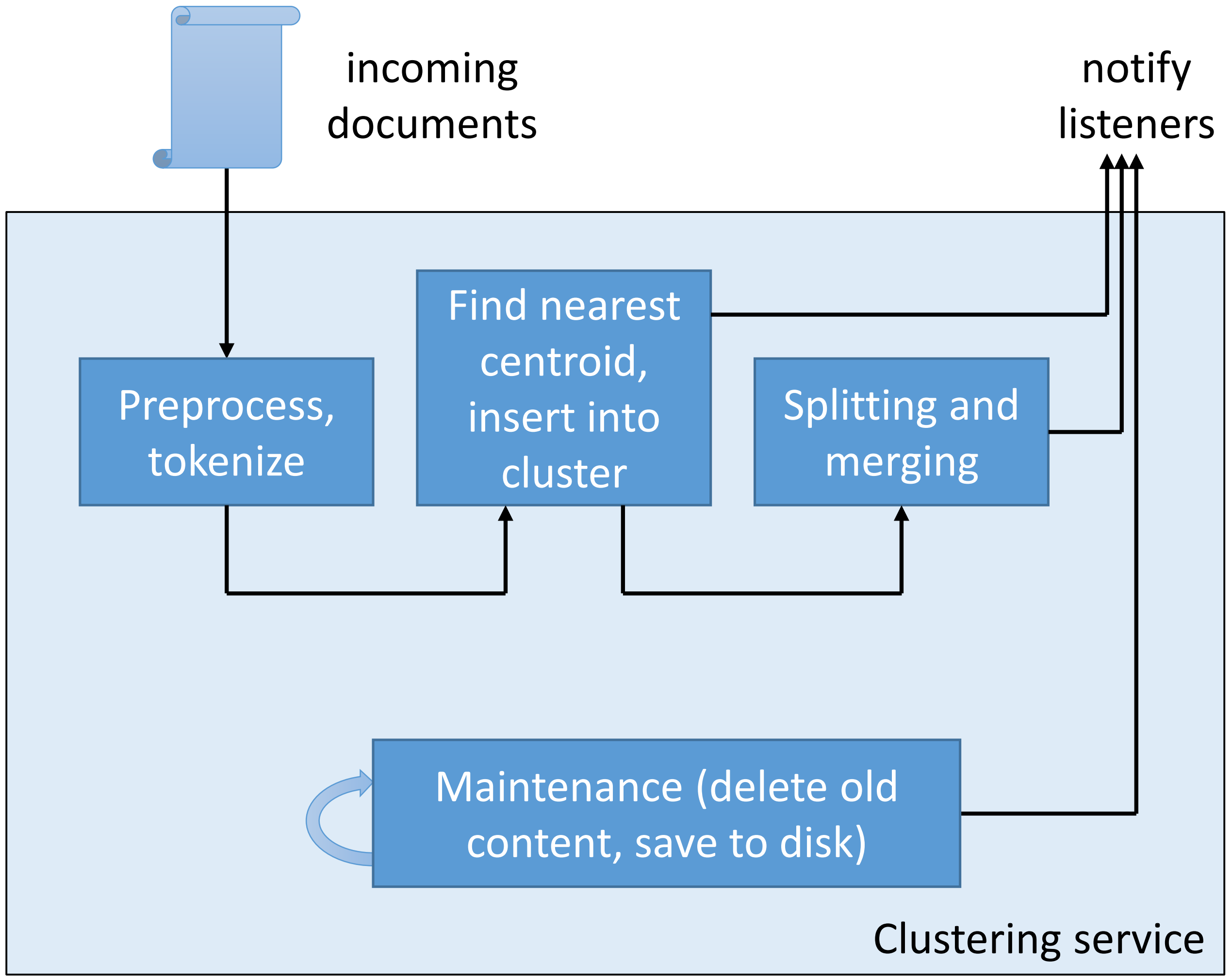
Janez Brank, Gregor Leban, Marko Grobelnik

Motivation

- Clustering
- Stream of documents
  - On-line approach
  - Asynchronous
  - Adaptive to changes in the stream
    - New clusters, splits, merges, discard old documents etc.
- Need for paralellization
  - Current aim: make good use of the parallel processing abilities of an individual computer
  - Multithreading, not distributed computing
- Main application: EventRegistry
  - Events $\approx$ Clusters of news articles

Architecture

- NewsCluster web service
- Receives new documents as HTTP requests
- Notifies listeners (by making HTTP requests) about changes
  - Cluster membership
  - Cluster splits/merges
  - Cluster medians
- Background tasks
  - Delete old clusters and their documents
  - Periodically save state to disk



Underlying clustering approach

- Main idea:
  - Assign each document to the cluster with the nearest centroid
- We can't do too much reassigning of documents between clusters
  - Because it's an on-line setting and we don't have time
  - Because the application wants a certain amount of stability in the clusters
- Cluster maintenance consists of
  - Occasional splits/merges
  - Deleting old clusters

Discard obvious duplicates

Underlying clustering approach

- In our input stream, there are many duplicates or near-duplicates
  - Basically the same article coming through several sources
  - Two documents are duplicates if they
    - Have the same title (modulo whitespace) and
    - Very similar TF-vectors (L1 distance below a threshold)
  - If a new article is found to be a duplicate of an existing article, it is discarded without further processing

Underlying clustering approach

Discard obvious duplicates

Prepare feature vector

- Document representation
  - Bag of words (TF-IDF vector)
  - Bag of concepts [if provided by the user]
  - Relative weight of each part in the resulting feature vector is customizable
  - Cosine similarity is used to compare feature vectors

Underlying clustering approach

- The document will be assigned to the cluster whose centroid is the closest to the document
  - Compute cosine similarity between the document and the centroids of all clusters
  - If even the closest centroid is far enough, start a new cluster containing just this document

Discard obvious duplicates

Prepare feature vector

Find nearest centroid

Discard obvious duplicates

Prepare feature vector

Find nearest centroid

Insert document into cluster

Underlying clustering approach

- Each cluster maintains various statistics that are updated incrementally when the cluster changes
  - Sum of feature vectors
  - Per-feature variances
  - Medoid
- The system supports weighting documents with exponentially time-decaying weights
  - But this is currently not used in our application

Underlying clustering approach

- After every few additions, we consider splitting the cluster into two subclusters
  - The method is based on bisecting k-means
  - Project all points onto a line and split them based on whether they fall left or right of the projection of the centroid
  - In the first pass, the line is simply the principal component, later it's a line through the centroids from the previous pass
- Accept the split if:
  - The Bayesian information criterion is met, and
  - The resulting subclusters would not meet our merge criteria
- Split by timestamp:
  - If the variance of timestamps is above a threshold
  - Split a cluster into an "older" and a "newer" subcluster
  - BIC is used to choose the relative size of the subclusters
  - Hopefully the older subcluster will be discarded soon

Discard obvious duplicates

Prepare feature vector

Find nearest centroid

Insert document into cluster

Consider splitting

Underlying clustering approach

- After every few additions, we consider merging the cluster with some other cluster
  - Candidates for merging are those clusters whose centroid is the closest to our (in terms of cosine similarity)
  - Centroids tend to be dense vectors, so cosine computations are slow
    - Use "pruned" centroids for an intial filtering step
  - Accept the merge if:
    - Cosine similarity is above a threshold; or
    - Lughofer's ellipsoid overlap criterion

Discard obvious duplicates

Prepare feature vector

Find nearest centroid

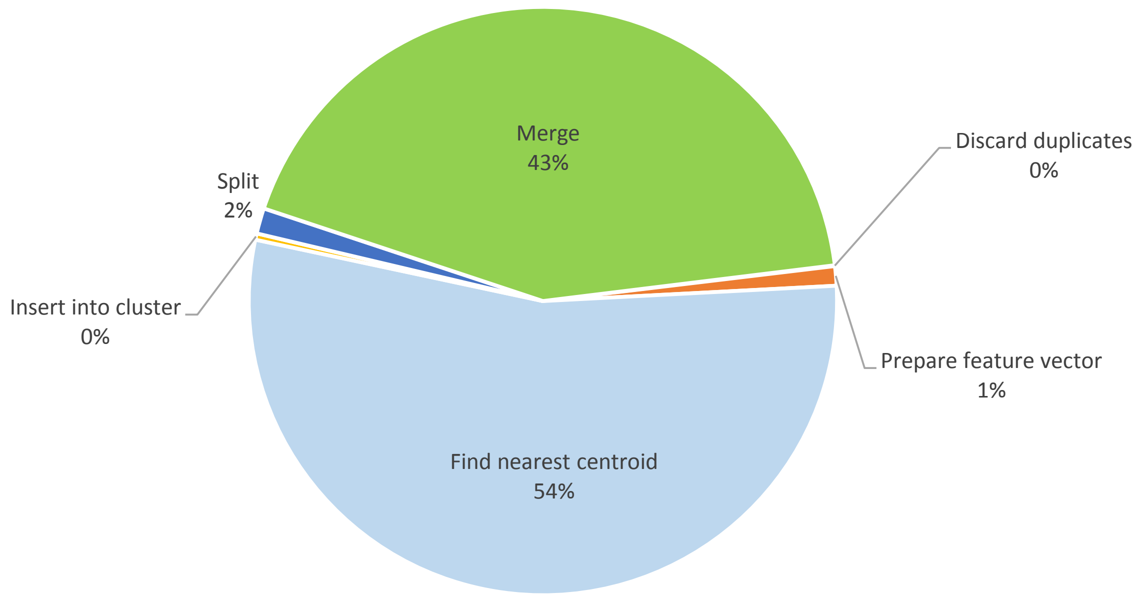
Insert document into cluster

Consider splitting

Consider merging

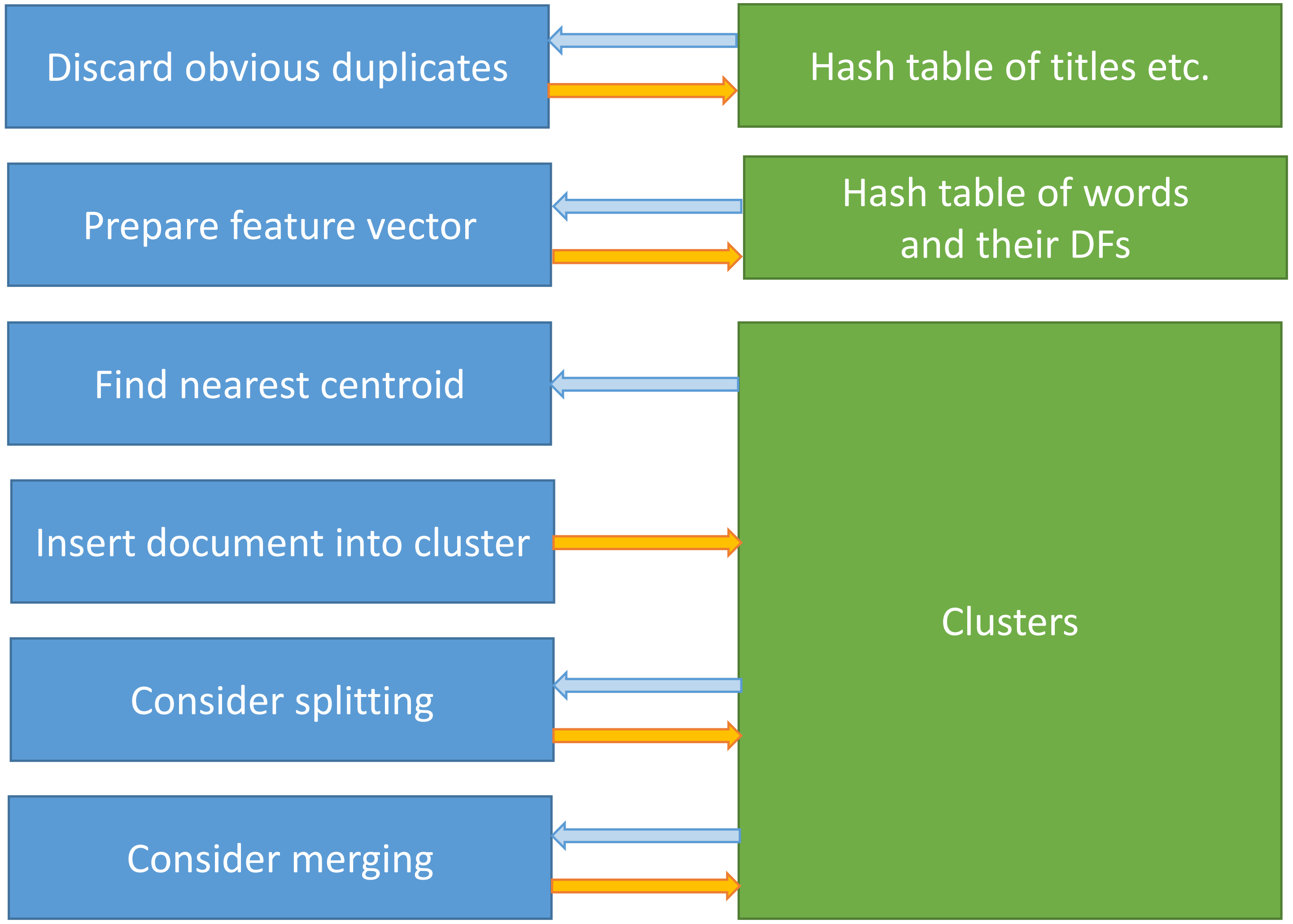
Considerations for paralellization

- Which are the most time-consuming parts of the process?
- 54%: computing cosines between the new document and all centroids
- 43%: computing cosines between one centroid and all other centroids to find merge candidates
- 3%: everything else



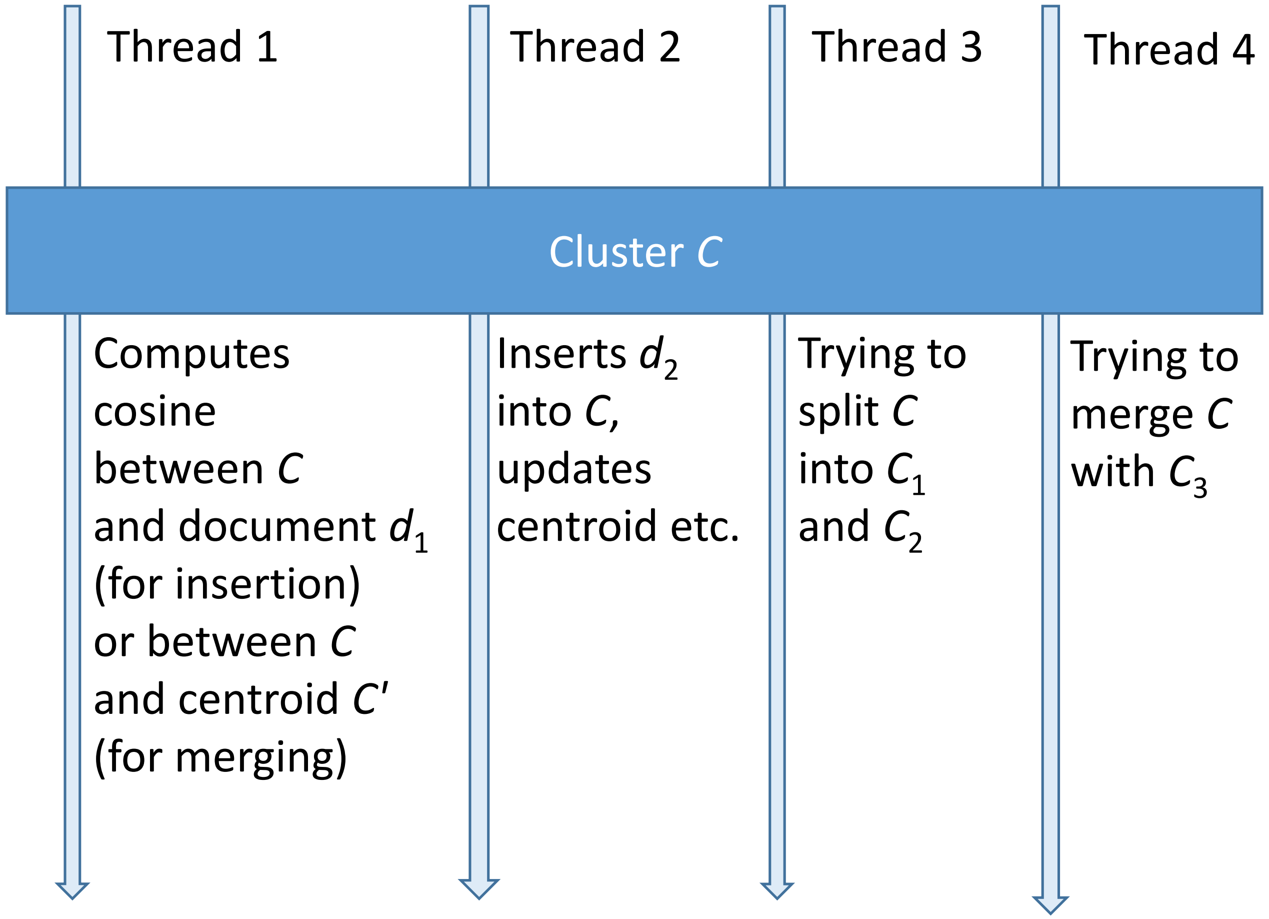
Considerations for paralellization

- Shared data structures:
  - Title hash table for duplicate detection
  - Word and DF table for bag-of-words vectors
  - Cluster membership and statistics


Discard obvious duplicates
Hash table of titles etc.
Prepare feature vector
Hash table of words and their DFs
Find nearest centroid
Insert document into cluster
Clusters
Consider splitting
Consider merging

Downsides of fine-grained paralellization

- Naive idea: assign each document to one worker thread
  - This thread executes the entire clustering algorithm for this document
- Downside #1: too much locking
- Downside #2: information relevant to the same cluster is scattered across multiple worker threads

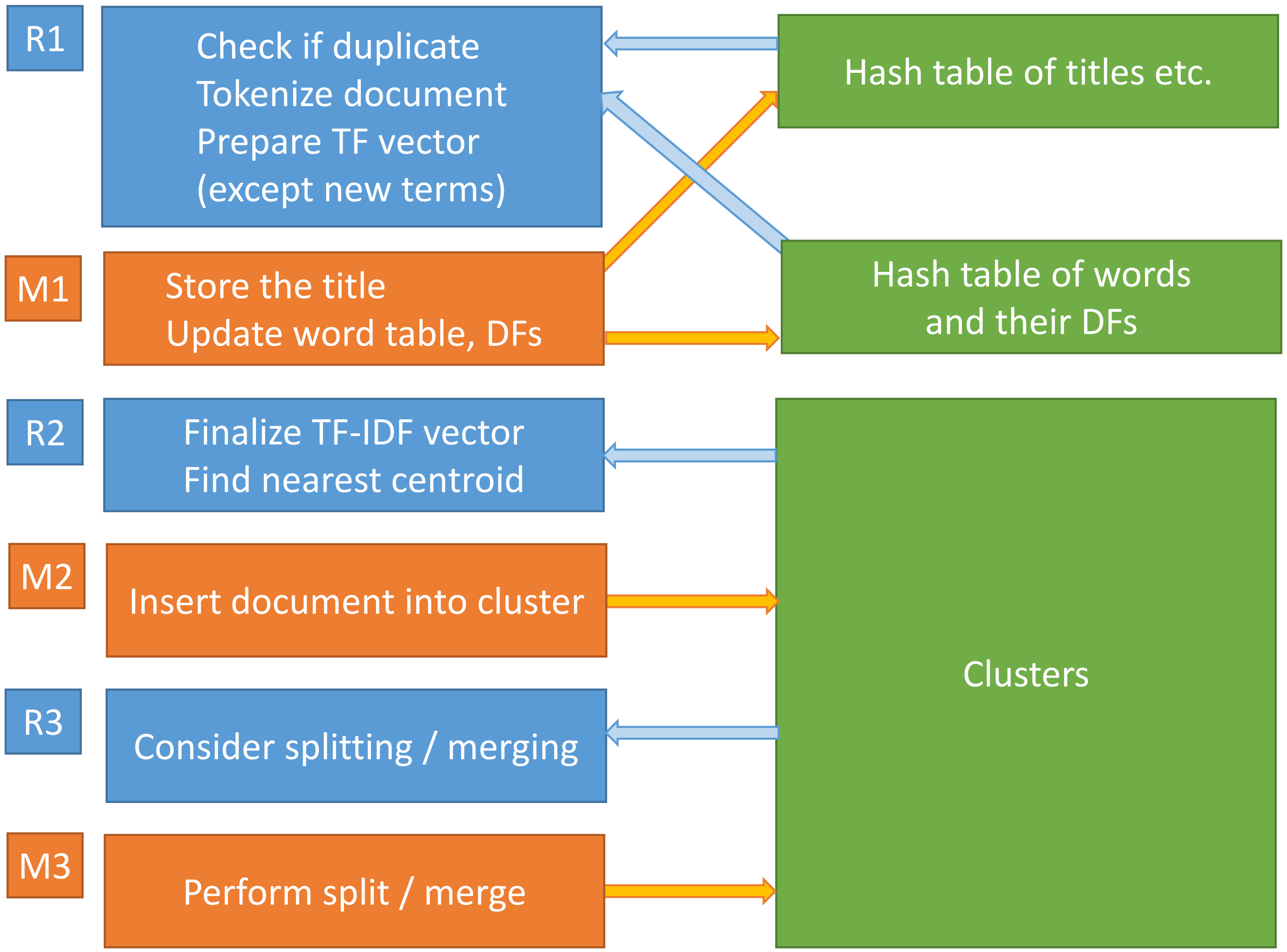

Thread 1
Thread 2
Thread 3
Thread 4
Cluster C
Computes cosine between C and document d_1 (for insertion) or between C and centroid C' (for merging)
Inserts d_2 into C, updates centroid etc.
Trying to split C into C_1 and C_2
Trying to merge C with C_3

Avoiding cluster-level locking

- If we want to avoid per-cluster locking:
  - No thread may modify a cluster
  - While some other thread is looping through clusters
    (to find the nearest centroid or to find merge candidates)
    - But each thread spends 95% its time doing this
  - And yet each thread will need to modify a cluster at some point
    (to insert the new document into it)
    - If it has to wait for all other threads to finish looping through clusters, that's a lot of waiting
    - And it doesn't help much with the problem of having multiple threads trying to do different things to the same cluster

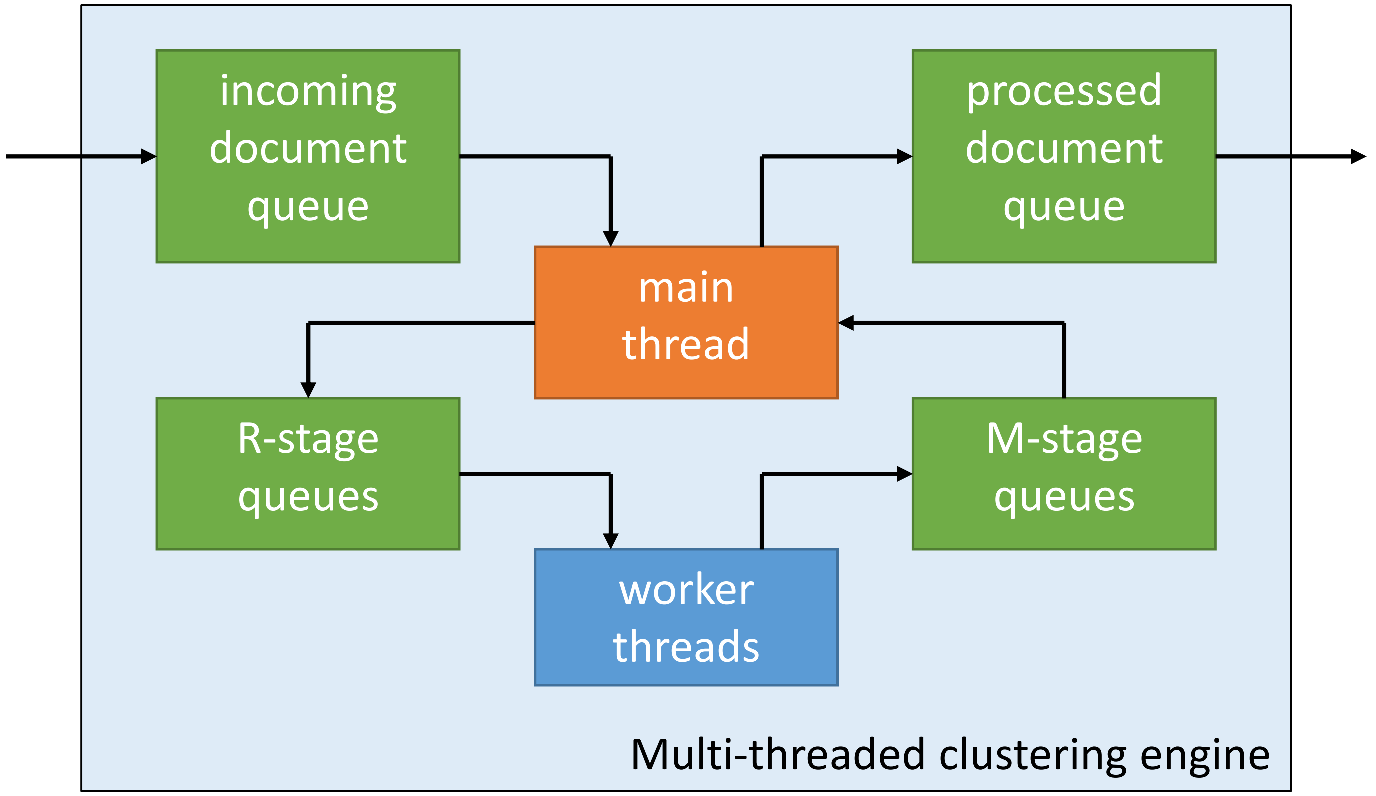
Read and write stages

- We can rephrase our approach a little to make it clearer which steps need to modify shared data structures



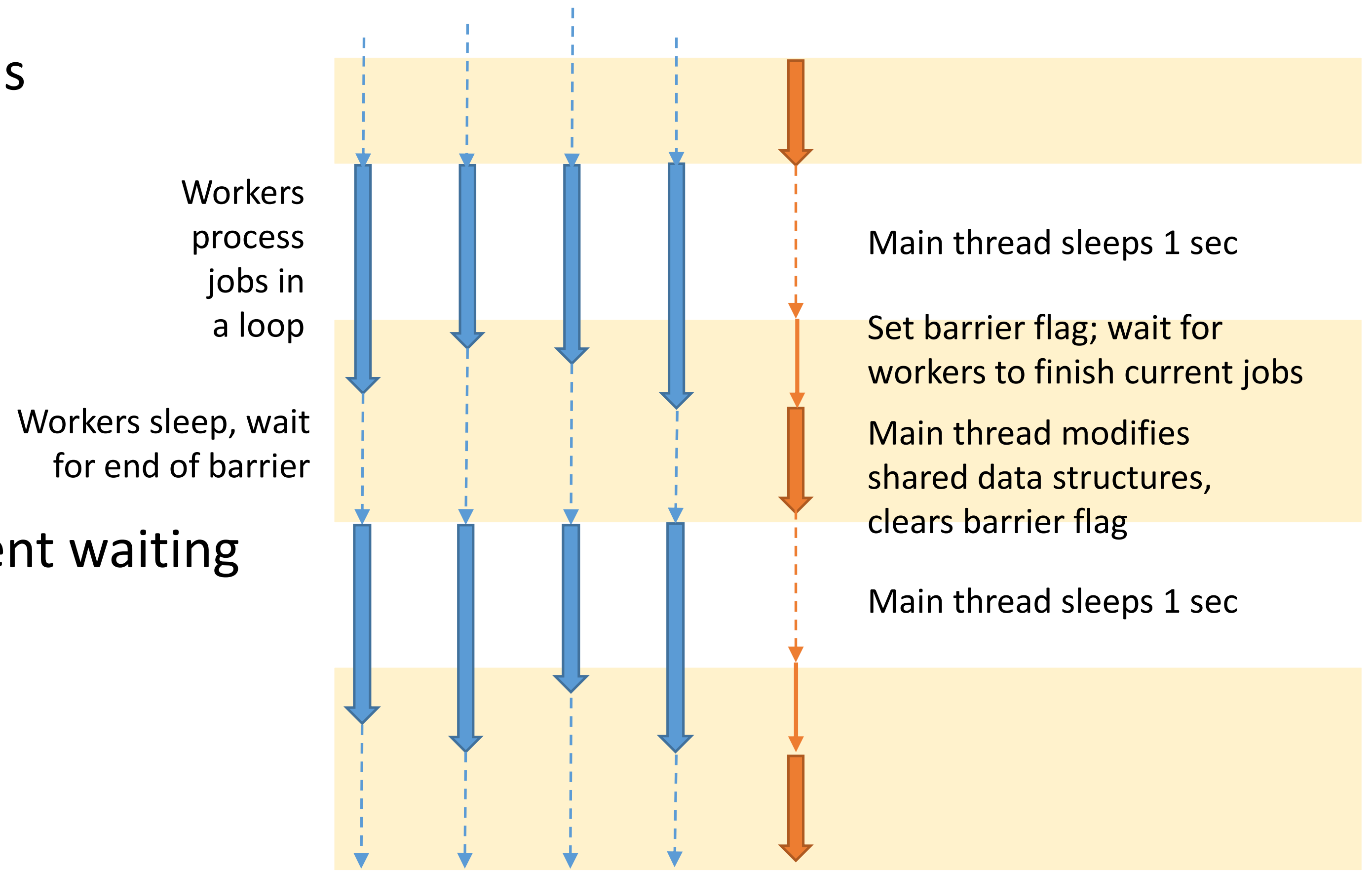
Main vs. Worker threads

- There is no reason why all the steps should be done by the same thread
  - *N* worker threads: perform read stages
  - 1 main thread: performs modify stages for all the requests
  - Only the main thread modifies any global data structures



Barrier-based parallelization

- Main thread needs to block all worker threads while it updates the shared data structures
  - Set a "barrier" flag to stop issuing new jobs to worker threads
  - Wait for all worker threads to finish their current job
  - Main thread can now modify shared data
  - Clear the barrier flag so the worker threads can resume
- We have a barrier once per sec
  - To reduce the amount of time spent waiting
  - This means each requests needs 3 seconds to be fully processed

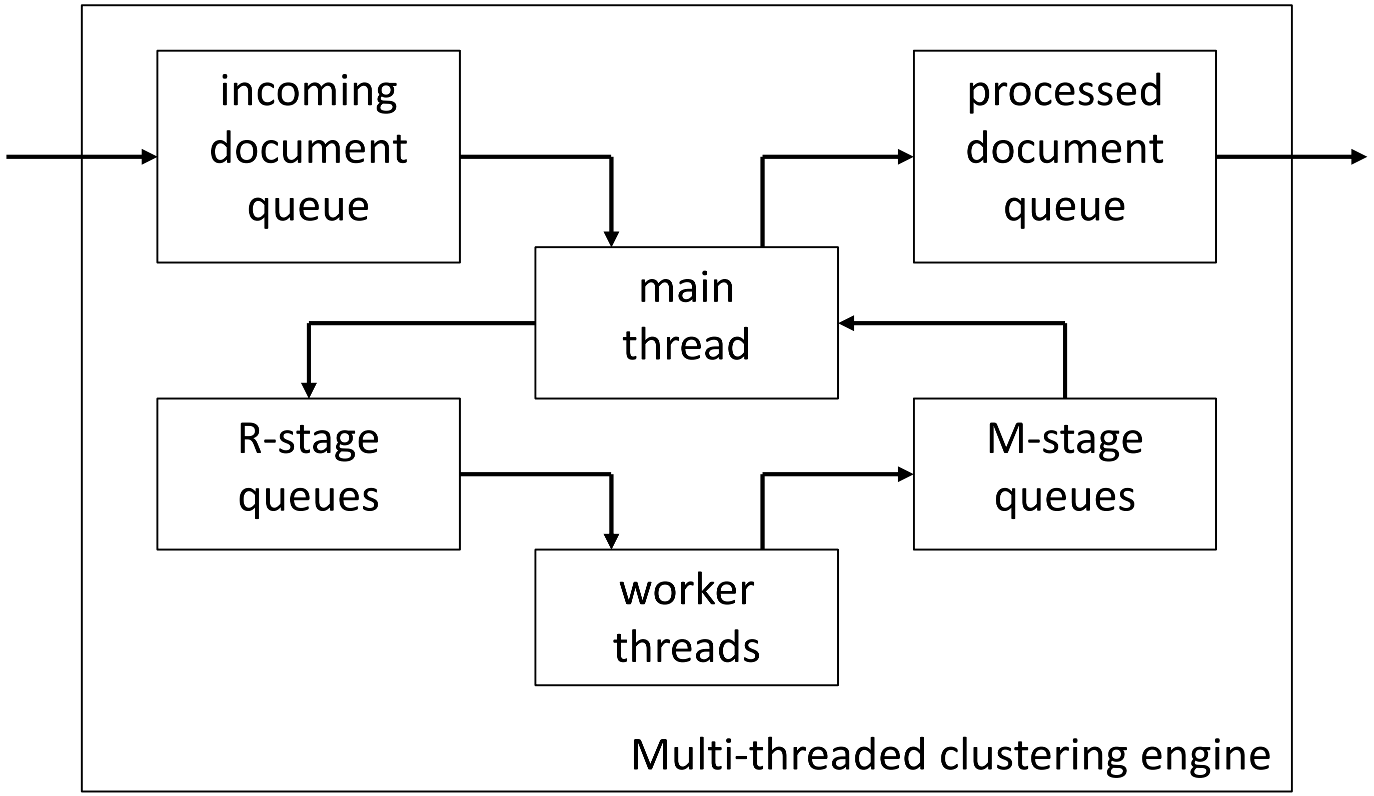

Workers process jobs in a loop
Workers sleep, wait for end of barrier
Main thread sleeps 1 sec
Set barrier flag; wait for workers to finish current jobs
Main thread modifies shared data structures, clears barrier flag
Main thread sleeps 1 sec

Barrier processing in the main thread

- For each request that went through an R-stage since the previous barrier, we now have to perform the corresponding M-stage
    - M1: add new titles, terms to shared hash tables, update DFs
    - M3:
        - In the R3 stages, the worker threads determined whether splits/merges should be done, and how exactly the documents should be split/merged
        - The main thread now carries out these split/merge proposals, ignoring those that would clash with already-processed proposals
    - M2:
        - In the R2 stages, the worker threads recommended where to insert which new document
        - The main thread now carries out these insertions and updates cluster statistics
        - While taking into account the splits/merges just carried out in M3

Conclusions and future work

- Multi-threaded clustering approach
  - Makes good use of parallel processing within a single computer
  - Low amount of locking and waiting
  - High throughput at the cost of high latency
  - Further paralellism comes from processing documents for different languages separately
- Possible future extensions
  - Use random projections to speed up the computation of cosine similarities
  - Hierarchical clustering
  - Distributed processing


incoming document queue
processed document queue
main thread
R-stage queues
M-stage queues
worker threads
Multi-threaded clustering engine