Regrouping Metric-Space Search Index for Search Engine Size Adaptation

Khalil Al Ruqeishi and Michal Konečný

School of Engineering and Applied Science
Aston University, Birmingham, UK
{alruqeik,m.konecny}@aston.ac.uk

Abstract. This work contributes to the development of search engines that self-adapt their size in response to fluctuations in workload. Deploying a search engine in an Infrastructure as a Service (IaaS) cloud facilitates allocating or deallocating computational resources to or from the engine. In this paper, we focus on the problem of regrouping the metric-space search index when the number of virtual machines used to run the search engine is modified to reflect changes in workload. We propose an algorithm for incrementally adjusting the index to fit the varying number of virtual machines. We tested its performance using a custom-build prototype search engine deployed in the Amazon EC2 cloud, while calibrating the results to compensate for the performance fluctuations of the platform. Our experiments show that, when compared with computing the index from scratch, the incremental algorithm speeds up the index computation 2–10 times while maintaining a similar search performance.

1 Introduction

A typical search engine distributes its search index into multiple processors to achieve a sufficiently high throughput [7,6,5,4,13,8,16]. However, the workload of a search engine typically fluctuates. Therefore, it is desirable that a search engine adapts its size to avoid wasting resources when the workload is low and to avoid unacceptable delays when the workload is high. If the engine is deployed in an Infrastructure as a Service (IaaS) cloud, the cloud facilitates allocating or deallocating compute resources to or from the engine.

Such an adaptive search engine repeatedly determines the number of processors to use, appropriately regroups the search data to form a new search index, and re-deploys the data onto the processors according to the new index.

Fig. 1 illustrates the running of such an adaptive search engine obtained using our small-scale prototype deployed on Amazon EC2.

In this paper, we focus on an important part of our prototype engine, namely the mechanism for regrouping the search data for a small or larger number of processors. We propose an algorithm for this task and evaluate its effectiveness using controlled tests in the prototype engine. We observe that our algorithm speeds up this task 2–10 times when compared with computing these groups

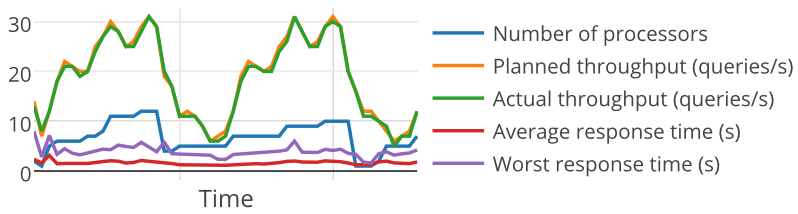


Fig. 1. Search engine updates number of processors whenever the workload changes.

from scratch (see Fig 6). In addition, the search performance does not deteriorate significantly when using our algorithm (see Fig 5).

The remainder of this paper is organized as follows. Section 2 recalls the background, in particular Subsection 2.1 describes the architecture of a search engine with a distributed metric space index. Section 3 reviews related work. Section 4 describes our algorithm for regrouping the search data. Section 5 presents the design and results of our experiments to validate and evaluate our algorithm. Section 6 concludes and outlines opportunities for further development.

2 Background

We build on previous research on distributing search data onto processors, in particular,

we use KmCol [7] for the initial grouping of search data. Let us recall the main components of KmCol because some of them feature in our incremental regrouping algorithm.

KmCol groups the search data in 3 steps, which leads to 3 levels of groupings. We adopt the following notation for the 4 types of object in these groupings:

- Data points: points in a metric space, representing the objects of the search
- LC-clusters: groups of nearby data points
- H-groups: groups of nearby LC-clusters, optimising for sample queries Q
- G-groups: groups of H-groups, one per processor

LC-clusters are computed using the List of Clusters (LC) algorithm [2,7,6,5]. LC-clusters are created to reduce the number of objects that feed into the next, more resource-intensive algorithm.

H-groups are formed from LC-clusters using K-means with the metric d_Q, derived from the set of sample queries Q, effectively optimising the engine for queries similar to Q. Due to the nature of K-means, H-groups have varying sizes.

G-groups are computed from H-groups using a procedure we call Group-Balanced, which attempts to balance their sizes.

The metric d_Q is defined using the *query-vector* model proposed in [14].

The metric d_Q makes pairs of points that are near in the natural metric seem far away from each other if they are close to many queries from Q. Conversely, the metric d_Q makes pairs of faraway points seem almost identical if they are not

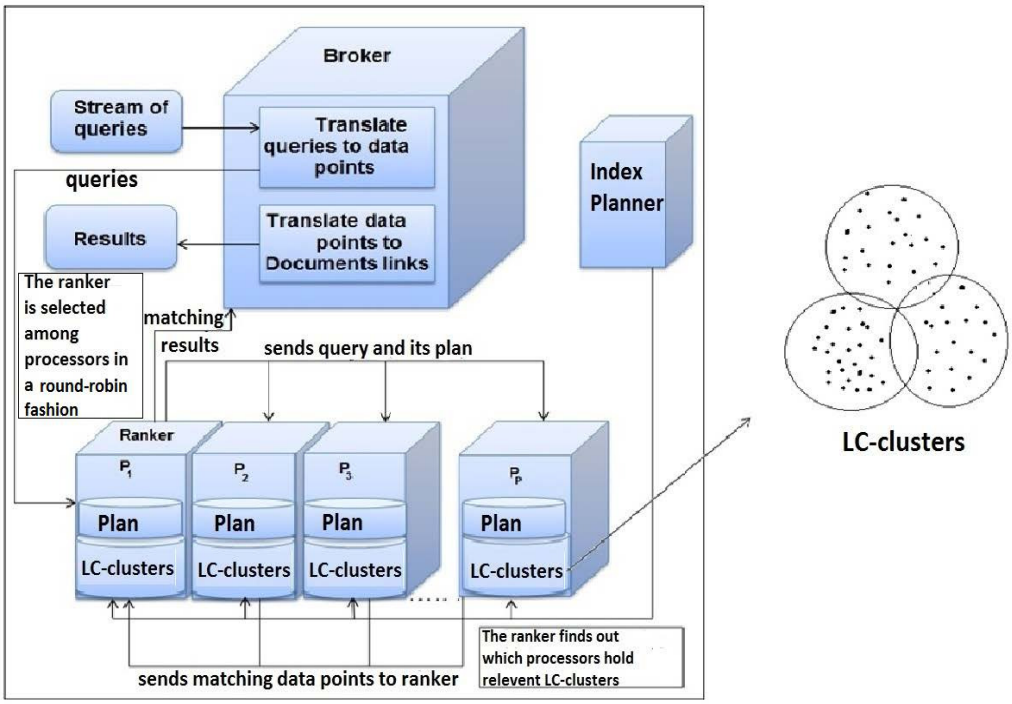


Fig. 2. Searching using distributed metric space index

near any of the queries from the set Q. This means that H-groups finely separate those LC-clusters that are more likely to contain results of queries, leading to a good balance of load among processors.

2.1 Search Engine Distributed Architecture

We utilise the search engine parallel processing architecture outlined in Fig 2. This architecture is analogous to that used in [7,5,4,13]. The Index Planner node is responsible for calculating G-groups and distributing them to the processors. It sends each processor not only its LC-clusters, but also an **index plan**, which is a map indicating for each LC-cluster on which processor it is. The index plan is used by the processor when it acts as a ranker for a query to determine which processors to contact regarding the query.

While search engines typically receive **k-nearest neighbor** (kNN) queries, i.e., "find k nearest objects to a specified object q for a small k" [7], search engines would translate such queries to **range queries** (q, r), i.e., "find all objects within distance r from q", because they are easier to distribute and process. Our engine also adopts this approach.

The ranker processor calculates the distance among the query and all of the centers across processors and formulates a *query plan*, namely the set of LC-clusters that intersect the ball of the range query (q, r).

The ranker sends the query and its query plan to the processor p_i that contains the first cluster to be visited, namely, the first LC-cluster that intersects the query ball. Then p_i processes all LC-clusters that intersect (q, r). For each such cluster, p_i compares (q, r) against the data points stored inside. The processor p_i then returns to the ranker the objects that are within (q, r) and passes the query and its plan to the next processor specified in the plan. This continues until all the processors in the query plan have returned their results to the ranker. The ranker sorts the query answers and passes the best k back to the broker as shown in Fig. 2. Each processor acts both as a processor in charge of processing a subset of LC-clusters and as a potential ranker.

Note that the architecture in Fig. 2 uses the Global Index Global Centers (GG) strategy because it uses a single node (i.e., the Index Planner) to compute the whole index. According to [6], such a global strategy performs better than local indexing strategies.

3 Related Work

According to [5], distributed metric space query processing was first studied in [12]. This work was extended in [6] for the LC-based approach, studying various forms of parallelization. As we said earlier, this study concluded that the GG strategy performs better than local indexing strategies.

An attractive feature of schemes without a global index is that they lend themselves to Peer-to-Peer (P2P) processing, which naturally supports resizing in response to load variations. For example, [11] presents a distributed metric space index as a P2P system called M-index. Unfortunately, such schemes tend to lead to a reduced search performance. Moreover, M-index is based on a pivot partitioning model, which has a high space complexity. For further related work using P2P metric space indexing see e.g. [9,10,3,15].

We note that [5,1] address the related problem of performance degradation when query load becomes unbalanced across processors. The query scheduling algorithm proposed in [5] balances the processing of queries by dynamically skewing queries towards explicit sections of the distributed index. On the other hand, [1] proposes dynamic load balancing based on a hypergraph model, but it is not concerned with multimedia search and does not use metric space index.

4 Adapting search engine size

An adaptive search engine will repeatedly re-evaluate its load and, when appropriate, switch over from p active processors to a different number of active processors. Recall that the initial H-group and G-groups were computed using the Km-COL algorithm as described in Sect. 2. Each switchover comprises the following steps:

1. Determine the new number of processors p' based on recent load.
2. (Re-)compute H-groups and G-groups (i.e., the index plan) for p' processors.

3. Distribute the index plan and the relevant LC-clusters onto each processor.
4. Pause search.
5. Switch to new LC-clusters and plan, de/activating some processors.
6. Resume search.

Our main contribution is an algorithm for step 2 and experimental evidence of how different ways of implementing step 2 impact search performance after the switchover. To allow us to focus on step 2 and the resulting search performance, we perform the switchovers while the engine is inactive, omitting steps 4 and 6. We also skip step 1 as p and p' will be determined by our experiment design.

4.1 Computing H-groups and G-groups

We compute G-groups from H-groups in the same way as in the KmCol algorithm. We therefore focus on the computation of H-groups for p' processors from H-groups for p processors. We introduce the following three methods (called **transition types**):

TT-R: Compute H-groups from scratch using K-means, like KmCol.
TT-S: Reuse the H-groups from previous configuration.
TT-A: Increase the number of H-groups using Adjust-H (Algorithm 1).

Algorithm 1 Adjust-H(d)(H, new_size)

Tuning Parameters: d: a metric on C
Input: H: a set of H-groups partitioning C,
new_size: the target number of H-groups
(new_size $> |H|$)

Output: updated H with $|H|$ = new_size

1: H_{sorted} = sort_by_decreasing_size(H)
2: **while** size(H_{sorted}) $\neq$ new_size **loop**
3: largest_group = H_{sorted}.getFirst()
4: new_groups = K-means(d)(largest_group, 2) // split
5: H_{sorted}.insert_sorted(new_groups)
6: H_{sorted}.delete(largest_group)
7: **end loop**
8: return H_{sorted}

Notice that the number of H-groups will never be decreased. This is appropriate because,

as we show in Sect. 5, reducing the number of H-groups does not improve search performance.

Adjust-H takes as parameters the number new_size ($= p' \cdot w$) and the old H-groups. On line 1, it starts by arranging the H-groups in an ordered collection, with the largest group first. On lines 2–7, the number of H-groups is increased by repeatedly splitting the largest H-group into two using K-means, until there are new_size many of them. Thanks to the following observation, we do not need to study the effect of repeated TT-A on search performance:

Proposition 1 (Repeated TT-A is equivalent to a single TT-A). *For any set H and sequence $|H| < p_1 < p_2 < \ldots < p_n$, it holds:*
Adjust-H$\big(\ldots$ *Adjust-H*$($*Adjust-H*$(H, p_1), p_2), \ldots, p_n\big)$ = *Adjust-H*(H, p_n)

Proof. A repeated execution of Adjust-H results in successive executions of the loop that forms the algorithm. There are no commands to change the H-groups between the successive executions of the loop. Thus the result of the repeated loop executions is the same as running the loop only once with new_size set to the final value p_n. □

To pursue our goal to speed up switchovers while keeping a good search performance, we will test the search performance implications of the three transition types TT-R, TT-S and TT-A.

Based on preliminary observations, we formed the following hypotheses:

H1 The time it takes to compute H-groups grows significantly with the number of these H-groups.

H2 Increasing the number of H-groups does not reduce search performance. Equivalently, when reducing p, TT-S does not lead to a worse search performance than TT-R.

H3 Computing a number of H-groups and then splitting them up using TT-A does not impair search performance when compared to computing the same number of H-groups directly using TT-R.

We provide experimental evidence supporting these hypotheses in Sect. 5.

Using these hypotheses, on the assumption that they are correct, we propose the algorithm Regroup (Algorithm 2) to decide which of the three transition types to use.

The algorithm takes as parameters the numbers $w_{\min}$ and w_{init}. TT-R uses w_{init} to compute H-groups from scratch, while $w_{\min}$ is used by TT-A to recompute H-groups. Due to hypothesis H2, the values of these tuning parameters do not significantly affect search performance. We therefore use the fairly low values $w_{\text{init}} = 2$ and $w_{\min} = 1.5$ in our experiments in order to reduce the time it takes to compute the H-groups. At the beginning, if a new Q is provided, it is necessary to update the metric d_Q and recompute the H-groups from scratch (TT-R, lines 2 and 3). If the number of H-groups is smaller than $p' * w_{\min}$, the number of H-groups is increased (TT-A, line 5). If there is no change in Q and $p > p'$, then H is reused (TT-S). Finally, on line 7, new G-groups are computed from the H-groups, using Group-Balanced, an algorithm borrowed from KmCol.

Algorithm 2 Regroup$(w_{\text{init}}, w_{\text{min}})(p', H, d_Q, Q)$

Tuning Parameters: w_{init}, $w_{\text{min}} \geq 1$
Input: p': new number of processors,
H: a set of H-groups partitioning C (optional, needed if Q absent),
d_Q: a metric on C (optional, needed if Q absent),
Q: sample set of queries (optional, needed if H absent)

Output: G: a partition of C with $|G| = p'$,
updated H and d_Q

1: **if** Q is provided **then**
2: $d_Q :=$ Query-Vector-Metric(C, Q)
3: $H := K$-means$(d_Q)(p' * w_{\text{init}}, C)$ // TT-R
4: **elseif** $|H| < p' * w_{\text{min}}$ **then**
5: $H :=$ Adjust-H$(d_Q)(H, p' * w_{\text{min}})$ // TT-A
6: **end** // TT-S: the if block not executed
7: $G :=$ Group-Balanced(H, p')
8: return G, H, d_Q

5 Experimental Evidence Supporting Hypotheses

In the experiments, the three transition types are compared in terms of their effect on *search performance* and the time it takes to compute H-groups for the new number of processors. (a component of *switch-over performance*). The performance is influenced by the following parameters:

1. **Search engine size evolution (SE)**: We consider only a single switchover at a time and write it as $p \rightarrow p'$. E.g., 5→8 encodes a single switchover from 5 to 8 processors. In our experiments, we use increasing or decreasing transitions of the sequence $2, 3, 5, 8, 12, 18$ and contrast sets of transitions sharing a similar ratio p/p' or sharing the same p'.
2. **Dataset (D)**: A dataset represents the set of objects that needs to be searched. In our experiments, we used a randomly selected set of 1,000,000 objects from the CoPhIR Dataset[1]. Each object comprises 282 floating-point number coordinates.
3. **Sample queries (Q)**: As explained in Sect. 2, the set defines the metric d_Q which is used to partition LC-clusters into H-groups. In our experiments, we used as Q a randomly selected set of 1,000 objects from the CoPhIR Dataset.
4. **Query profile (QP)**: Query profile simulates how users send queries to the search engine. It is determined by a sequence of queries and the timing when each query occurs. In our experiments, we use as queries 100,000 randomly

[1] http://cophir.isti.cnr.it/

selected objects from the CoPhIR Dataset. We fire the queries at a constant query rate. This rate is not a parameter of the experiment because it is determined automatically in the process of measuring maximum throughput as described below.

Search performance is measured using *maximum throughput* defined as follows. The current output throughput of a search engine is the rate at which answers to queries are sent to clients. This is equal to the input throughput, i.e., the rate at which the queries are arriving, except when the queries are accumulating inside the engine.

Maximum throughput is the highest output throughput achieved when flooding the input with queries. We have observed that the network stack efficiently facilitates the queuing of queries until the engine is able to accept them. Each of the search engine nodes (Fig. 2) was deployed on a separate Amazon EC2 medium virtual machine instance. In each experiment, we used the following steps to obtain sufficiently reliable throughput measurements despite significant performance fluctuations of the Amazon cloud platform:

- Conduct two speed tests: an initial and a final test. The two tests are identical. Each test comprises 4 repetitions of a fixed task based on distributed searching.
- If the speed variation within these 4 repetitions is over 5%, the cloud is not considered sufficiently stable.
- Also if the initial and final speed measurements differ by over 2%, the cloud is not considered sufficiently stable.
- The average of the speed measurements in the initial and final tests is used to calibrate the maximum throughput measurements obtained in the experiment to account for longer-term variations in the cloud performance.

When the stability tests failed repeatedly, we relaxed the thresholds and took the average of the measurements obtained from 3 repetitions of the experiment.

We observed that in many experiments, the throughput fluctuates at the beginning and then stabilises. To discount the initial instability, we run each search experiment as a sequence of blocks of 100 queries and we wait until there are four consecutive blocks with a performance variation below 30%. We discount the preceding blocks that have a higher variance.

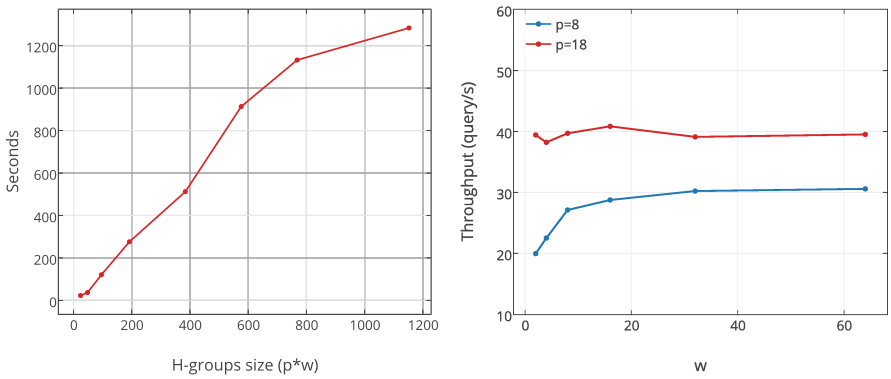
We artificially slowed down all processors by a factor of 5 to compensate for the slow network in the EC2 cloud (around 240 MBits/s), simulating a faster network, which would be found in typical clusters (around 1 GBit/s).

The full code for our experimental search engine and the experiments described in this section is available on `http://duck.aston.ac.uk/ngp`.

5.1 The Number of H-groups

Experiment E1. To test hypothesis H1, we computed different numbers of H-groups and observed how the computation time grows with size while the remaining parameters are fixed.

The results (Fig. 3(a)) confirm hypothesis H1.



(a) TT-R computation time grows (b) Throughput is not significantly affected

Fig. 3. Impact of increasing the number of H-groups ($= w * p$) on performance.

Experiment E2. In a similar setup as experiment E1, we checked whether the extra computation time spent creating more H-groups translates to better search performance, in contradiction to hypothesis H2. We have done this for $p = 8$ and $p = 18$ and the same values of w as for E1. The results of $E2$ in Fig. 3(b) show that the throughput is not significantly affected by w, confirming H2.

5.2 Search Performance of TT-S

Experiments E3 and E4. Reusing H-groups for $p' < p$ (TT-S) is much faster than recomputing H-groups (TT-R). The alternative phrasing of hypothesis H2 states that this speed up does not come at a cost to the search performance. Here we report on experiments that confirm hypothesis H2 in the alternative phrasing: The same switchover $p \to p'$ is performed using TT-R and independently using TT-S and the resulting search performance is measured.

These two experiments differ in the set of switchovers considered as follows:

- E3 varies p' and fixes the ratio p/p'.
- E4 varies the ratio p/p' and fixed p'.

The results of these experiments shown in Fig. 4 support H2: TT-S does not lead to worse search performance than TT-R when switching over to a smaller number of processors.

5.3 Comparing TT-A and TT-R

Experiments E5 and E6. In this section, we test hypothesis H3 by comparing the results of experiments that measure the search performance after computing

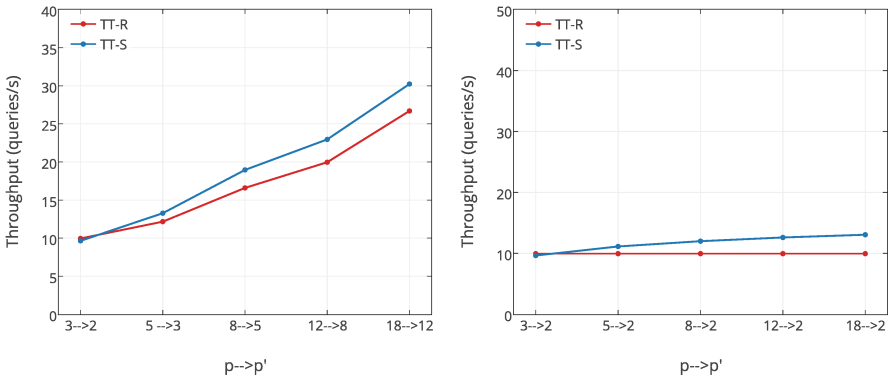


Fig. 4. TT-S and TT-R produce similar throughput, measured separately for increasing p' (E3) and increasing p/p' (E4).

H-groups using TT-R and TT-A. Moreover, we capture the computation time of the transitions to measure the speed-up of TT-A over TT-R.

As with E3 and E4, the experiments differ in the set of switchovers considered as follows:

- E5 varies the ratio p/p' and fixed p'.
- E6 varies p' and fixes the ratio p/p'.

The results of experiments $E5$ and $E6$ in Fig. 5 support hypothesis H3, namely they show that the maximum throughputs after TT-A is similar to, sometimes even better than the maximum throughput after TT-R. Plots in Fig. 6 show that in this context the speed-up of TT-A versus TT-R is 2–10 times.

6 Conclusions

We have proposed a new algorithm for planning an incremental regrouping of a metric-space search index when a search engine is switched over to a different size. This algorithm is inspired by the results of a set of experiments we conducted. These experiments also indicate that our algorithm facilitates 2–10 times faster switchover planning and leads to a similar search performance when compared with computing the index from scratch.

In this work, we studied only the re-computation of the metric-space index when the search engine changes size. We plan to develop and study the remaining aspects of an adaptive search engine, such as determining when and how to change the engine size and re-distributing the search data among processors processors according to the newly computed search index while keeping the engine responsive.

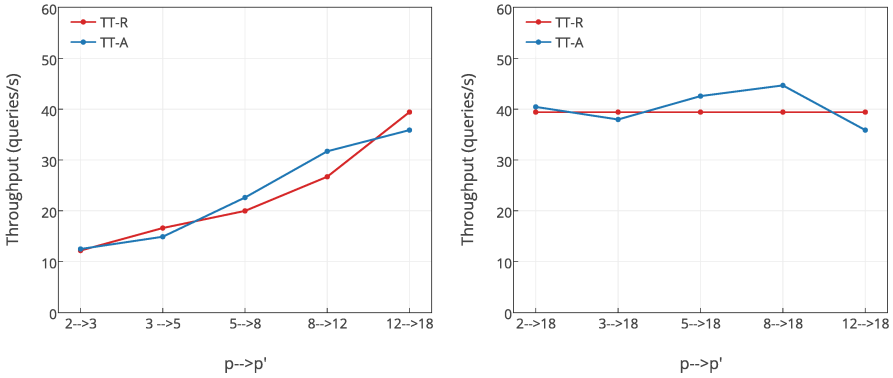


Fig. 5. TT-A and TT-R lead to a similar maximum throughput after switchovers with various p' and with various ratios.

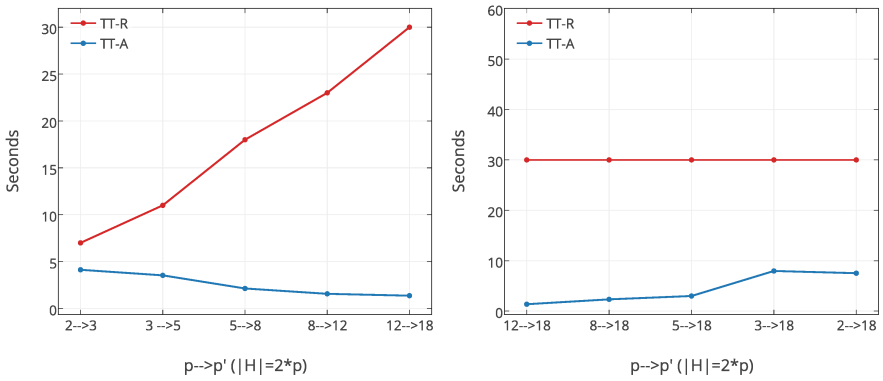


Fig. 6. TT-A is faster than TT-R in switchovers with with various p' and various ratios.

References

1. Catalyurek, U.V., Boman, E.G., Devine, K.D., Bozdağ, D., Heaphy, R.T., Riesen, L.A.: A repartitioning hypergraph model for dynamic load balancing. Journal of Parallel and Distributed Computing 69(8), 711–724 (2009)
2. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters 26(9), 1363–1376 (2005)
3. Doulkeridis, C., Vlachou, A., Kotidis, Y., Vazirgiannis, M.: Peer-to-peer similarity search in metric spaces. In: Proceedings of the 33rd international conference on Very large data bases. pp. 986–997. VLDB Endowment (2007)
4. Gil-Costa, V., Marin, M.: Approximate distributed metric-space search. In: Proceedings of the 9th workshop on Large-scale and distributed informational retrieval. pp. 15–20. ACM (2011)
5. Gil-Costa, V., Marin, M.: Load balancing query processing in metric-space similarity search. In: Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on. pp. 368–375. IEEE (2012)
6. Gil-Costa, V., Marin, M., Reyes, N.: Parallel query processing on distributed clustering indexes. Journal of Discrete Algorithms 7(1), 3–17 (2009)
7. Marin, M., Ferrarotti, F., Gil-Costa, V.: Distributing a metric-space search index onto processors. In: Parallel Processing (ICPP), 2010 39th International Conference on. pp. 433–442. IEEE (2010)
8. Marin, M., Gil-Costa, V., Bonacic, C.: A search engine index for multimedia content. In: Euro-Par 2008–Parallel Processing, pp. 866–875. Springer (2008)
9. Marin, M., Gil-Costa, V., Hernandez, C.: Dynamic p2p indexing and search based on compact clustering. In: Similarity Search and Applications, 2009. SISAP'09. Second International Workshop on. pp. 124–131. IEEE (2009)
10. Novak, D., Batko, M., Zezula, P.: Metric index: An efficient and scalable solution for precise and approximate similarity search. Information Systems 36(4), 721–733 (2011)
11. Novak, D., Batko, M., Zezula, P.: Large-scale similarity data management with distributed metric index. Information Processing & Management 48(5), 855–872 (2012)
12. Papadopoulos, A.N., Manolopoulos, Y.: Distributed processing of similarity queries. Distributed and Parallel Databases 9(1), 67–92 (2001)
13. Puppin, D.: A search engine architecture based on collection selection. Ph.D. thesis, PhD thesis, Dipartimento di Informatica, Universita di Pisa, Pisa, Italy (2007)
14. Puppin, D., Silvestri, F., Laforenza, D.: Query-driven document partitioning and collection selection. In: InfoScale '06: Proceedings of the 1st international conference on Scalable information systems. ACM Press, New York, NY, USA (2006)
15. Yuan, Y., Wang, G., Sun, Y.: Efficient peer-to-peer similarity query processing for high-dimensional data. In: Web Conference (APWEB), 2010 12th International Asia-Pacific. pp. 195–201. IEEE (2010)
16. van Zwol, R., Rüger, S., Sanderson, M., Mass, Y.: Multimedia information retrieval: new challenges in audio visual search. In: ACM SIGIR Forum. vol. 41, pp. 77–82. ACM (2007)