Empirical Evaluation of Strategies to Process Range Queries of Numeric Sequences in Batch-Mode

Luiz F. A. Brito[1], **Marcelo K. Albertini**[1]

[1]Faculty of Computing – Federal University of Uberlândia (UFU)
Av. João Naves de Ávila, 2121 – Santa Mônica – 1A236 – Uberlândia – MG – Brazil

{luiz.brito,albertini}@ufu.br

Abstract. *Tree structures are largely used to index and search sequences on secondary memory. In some situations, many range queries are processed almost simultaneously and the resulting number of disk accesses can be high. In order to reduce the number of disk accesses, similar sequences can be grouped and spanned as a single query. A simple strategy is to unify all sequences into a single group. However, other strategies for grouping sequences can also be used. In this paper, we present and empirical evaluation of* 5 *common grouping strategies for* R*-trees and* M*-trees. Our results indicate that for inputs modelled as a random walk distribution the overall best implemented strategy for grouping queries is indeed the one unifying all queries in a single group.*

1. Introduction

Fast retrieval of similar sequences are essential in many applications. In query-by-humming systems for example, a hum is sent as a query to retrieve songs with similar passages [Kotsifakos et al. 2015]. In protein similarity search, large indexes containing DNA sequences allow querying for known genomes [Camoglu et al. 2003].

The problem of range query for sequences is defined as follows: given a query sequence, find sequences in database that are similar within a distance ϵ [Faloutsos et al. 1994]. The naive strategy would retrieve all sequences from secondary memory and evaluates them sequentially. However, it is costly since every sequence in database may be considered.

Index structures such as R-trees [Guttman 1984] and M-trees [Ciaccia et al. 1997] are commonly used to answer range queries efficiently. These indexes organize similar sequences in order to discard tree branches not containing candidates during a range query.

In problems where multiple range queries can be processed in batch, we can use query grouping strategies to improve performance. These strategies group similar query sequences so that we can use a smaller number of range queries to search the index structure. This reduces the number of disk accesses because similar queries would traverse almost identical paths in the tree.

A simple strategy is to group all queries together and to search through the index only once [Moon et al. 2001]. This can improve the search when queries are similar. However, when many query sequences are dissimilar the resulting range query can cover most of the index. In the worst case, the search is degraded to scanning all sequences. Therefore, the choice of query grouping strategies impacts directly on the performance of batch-mode range queries.

The problem of grouping sequences is mentioned in literature. For example, [Faloutsos et al. 1994] compares grouping techniques for finding R-tree hyperrectangles during insertion to reduce the size of indexes. In [Fu et al. 2008] the authors perform multiple queries using a single hyperrectangle motived by its simplicity. Although the impact of grouping strategies in performance of range queries may be high, this subject is usually little discussed.

In this paper, we empirically evaluate 5 strategies to group sequences. They are described in Section 3. Some of these strategies, originally described for R-trees [Faloutsos et al. 1994, Moon et al. 2001, Fu et al. 2008], were adapted to the M-tree structure.

In our experiments, we model sequences by a random walking distribution. We use this model because often sequences can be transformed into random walks by using difference operations. We have performed two types of tests: using $\epsilon = 0$ to check if a sequence is in the database and using small values of $\epsilon > 0$ to retrieve only a few similar sequences. Our results show for small $\epsilon > 0$, R and M-trees number of disk accesses is significantly reduced by batch-mode strategies.

In the next section, we describe R and M-tree operations to index sequences. Next, we detail the query grouping strategies implemented for our tests and how they are applied to R and M-trees. In Section 4, we describe our methodology and discuss our results. Finally we draw our conclusions regarding the usage of query grouping strategies.

2. Background

R and M-trees are data structures optimized to secondary-memory storage. Often, they are employed to index sequences for fast retrieval through queries such as range queries. In R-trees, sequences are treated as d-dimensional points where similar points are grouped in the same node limited by a d-dimensional hyperrectangle, also known as Minimum Bounding Rectangle (MBR). M-trees can index any kind of object as long as its representation is comparable by a metric function. Differently from R-tree, M-tree nodes are delimited by hyperspheres.

There are two types of nodes in these structures: directory nodes, containing objects of directory type, and leaf nodes, containing objects of leaf type. Each directory object contains an id and a `container` representation, which is a hyperrectangle in a R-tree and is a sequence in a M-tree, along with a radius distance. Each directory also has a reference to a child node and, for M-tree nodes, a precomputed distance to the parent object in order to speed up calculations by using triangle inequality [Orchard 1991]. Each leaf object contains an id; a sequence, which works as a key added by an insertion, and a value to be associated with the key.

In an insertion, given a new sequence to be stored, the procedure for both trees can be described as follows: at the root level, if the node is not a leaf then choose the directory object that has the `container` that best fits or is closer to the new sequence. If the sequence does not fit entirely inside the `container`, then enlarge it and retrieve the child node referenced by the directory object. This step is applied recursively for child nodes until a leaf node is found. Finally, it adds a new leaf object containing the new sequence.

During the insertion, if a node is already full then a split policy is applied in order to redistribute its objects between itself and a new node. When a leaf node is full, the insertion algorithm firstly creates a new node and calls the split policy to redistribute the leaf objects. Then it creates two new `containers` in order to fit the two nodes and inserts them in the parent's object list, replacing the old `container`. In case the full node is a directory, its objects are also redistributed and the new `containers` are created from other `containers`. For M-trees the algorithm also needs to update parent distance of modified objects. This splitting process is applied recursively up to the root node. If the root is also full, then a new root node is created and the tree grows one level.

In a range query, given a sequence and the maximum distance allowed ϵ, first the algorithm creates a query `container`. This `container` has the center equals to the query sequence and each dimension width equals to 2ϵ. The search operation begins at root level and descends every branch that has possibility to be inside the container. This process repeats recursively until all possible leaves are reached. Finally, the algorithm retrieves every sequence reached within a distance ϵ to the query sequence.

3. Strategies for Processing Range Queries in Batch-Mode

In the literature, the predominant approach, named `Single Grouping`, is to group all queries together and search through the index just once [Moon et al. 2001, Fu et al. 2008].

In order to verify the importance of grouping queries to process range queries in batch-mode, in this paper we evaluate the following strategies: `k-Medoids Grouping`, `Single Grouping`, `n-Random Grouping`, `Maximum Capacity Grouping`, and `Adaptive Grouping` [Fu et al. 2008, Park and Jun 2009, Faloutsos et al. 1994].

`Single Grouping` (SG) strategy puts every query in a single group and performs just one range query using a single query `container`. For R-tree, this strategy is the same as the `n-Random Grouping` strategy with $n = 1$ since any query can be chosen at the beginning of the grouping process and it would still result in the same `container`. However, for M-trees is important to choose the query that is centralized. In this case, our algorithm chooses the query that has the lowest distance sum to all other queries.

`n-Random Grouping` (NRG) strategy selects n queries randomly and assigns them to different groups. They are enclosed by an initial `container`. It assigns each remaining query to the best-fitting container. If a query does not fit entirely in the `container`, then enlarge it. Finally, for each group enlarge its `container` by ϵ and process a range query.

In the `Maximum Capacity Grouping` (MCG) strategy, given the maximum number of sequences allowed per group m, the algorithm first creates a new group enclosing an initial `container`. Then it assigns the next m queries to the current group, enlarging the `container` when necessary. If the current group becomes full, then the algorithm creates a new group. This process continues until every query have been assigned to a group.

`k-Medoids` (KMG) strategy finds the best representative queries to form k clusters, namely medoids, such that the sum of distances among every object and its medoid is minimized. In this paper we used the algorithm presented in [Park and Jun 2009] because

it outperforms the well known clustering algorithm Partition Around Medoids (PAM) despite its simplicity.

The algorithm to find the k medoids first computes the distance matrix of all objects. Then, it selects k initial medoids based on lowest sum of normalized distances computed as $v_j = \sum_{i=0}^{n-1} (d_{ij} / \sum_{l=0}^{n-1} d_{il})$, where j is the target object, i and l can be any object and d_{xy} is the distance between objects x and y. In next step, this algorithm assigns the remaining objects to clusters that have the closest medoids and it calculates the cost of the current configuration. For the following steps, similar to partitioning clustering algorithms, the algorithm iteratively updates medoids such that the total distance inside cluster is minimized and reassigns the remaining objects to clusters. This algorithm finishes when objects stop changing groups.

Our `Adaptive Grouping` (AG) strategy is adapted from [Faloutsos et al. 1994], originally introduced to group d-dimensional points in MBRs during insertion. They proposed to group sequences at insertion time but not at querying time. We have adapted this strategy to the context of querying. AG calculates the marginal cost for assigning a query in the current group. The marginal cost mc for R-tree is calculated by: $mc = \prod_{i=1}^{d}(L_i + 0.5)/k$, where L_i is ith dimension width of the resulting MBR and k is the number of queries in resulting group. The strategy described in [Faloutsos et al. 1994] says if the marginal cost for a given query is lower than the previous cost then it assigns the query to the current group and enlarges the `container` as necessary. Instead, in our implementation we also look for other groups to check for lower maginal costs. It assigns the current query to a new group if its cost is larger than the current cost for all groups.

For M-trees, we propose to compute the marginal cost as: $(r + 0.5)^d/k$, where r is the radius of resulting hypersphere. The marginal cost for M-tree is justifiable because its value increases proportionally to the volume of the query hypersphere. We note the data needs to be normalized in range $[0, 1)^d$ in order to compute these marginal costs.

4. Experiments

In order to compare the query grouping strategies previously discussed we have built indexes using R and M-tree structures for dimensions ranging from 2^2 to 2^{10}. For each dimension we generated a dataset containing 10^5 sequences using the random walk distribution. Sequence values were normalized to the interval $[0, 1]$ using the formula $S_i^{norm} = (S_i - min(S))/(max(S) - min(S))$, where S_i is a value in sequence S, treated by the indexes as a dimension, and, $min(S)$ and $max(S)$ are the lowest and highest values of S, respectively.

In our experiments, for each dimension we used the same indexes to test different grouping strategies in order to prevent potentially effects related to the index construction and data randomness. We organize our experiments as following: for each index we processed batch-mode range queries containing 25, 100 or 1000 sequences. For `n-Random Grouping` and `k-Medoids Grouping` strategies we chose the number of groups to be the base 2 logarithm and the square root of the batch size. For the `Max Grouping` we chose the maximum allowed number of sequences such that the number of resulting groups was almost the same as NRG and KMG strategies.

The experiments were divided in two categories. In the first category, we processed

batch-mode range queries with $\epsilon = 0$ in order to check the existence of sequences in indexes. In the second category, we computed ϵ based on the dataset average energy. The mean energy E of a dataset is computed by $E(D) = (\sum_{i=0}^{n-1} \sum_{j=0}^{d-1} |D_{ij}|^2)/n$ where D_i is a sequence of size d and n is the number of sequences in the dataset. The final formula was $\epsilon = \frac{1}{4}\sqrt{E(D)}$. This value for ϵ allows retrieval of roughly the same number of results given a batch-mode range query.

In order to assess the grouping strategies we measured the ratio of the number of disk accesses per query when batch-mode was not used divided by the total disk accesses used per query by a given strategy, and a similar formulation of ratios for distance calculations. Ratio values higher than 1 indicate improvement of performance when using batch-mode strategies. The naive strategy, which process a range query for each sequence independently, was the baseline for our experiments and we refer as `No Grouping` (NG) strategy. Figure 1 shows the results regarding disk accesses. Each experiment considers: data structure, grouping strategy, ϵ value, amount of sequences per batch, and sequence dimensionality.

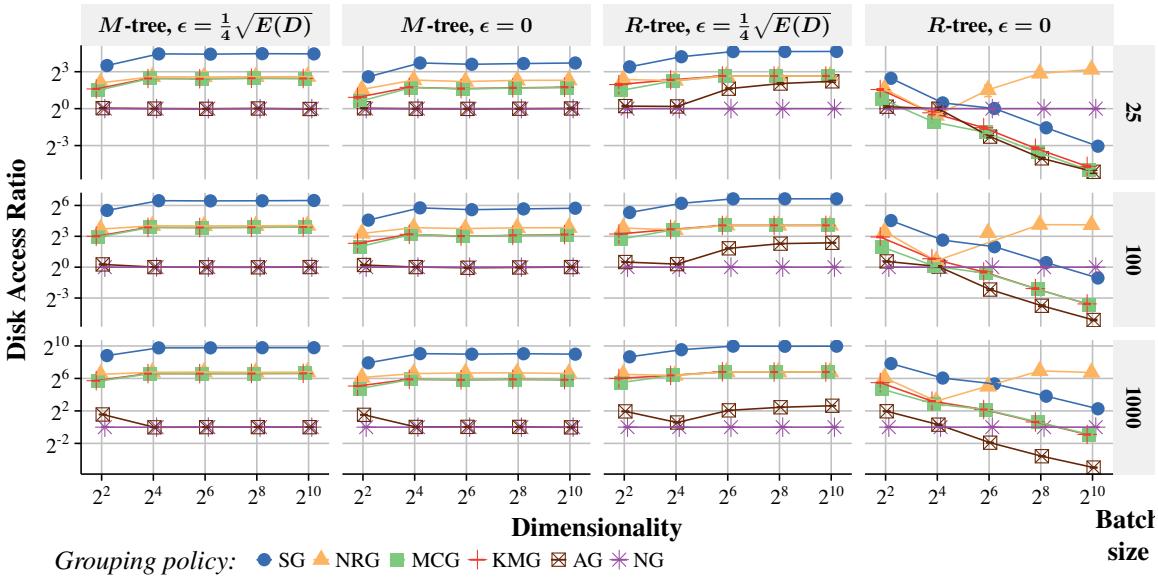

Figure 1. Disk accesses ratios for query grouping strategies using R-trees and M-trees.

In Figure 1, we noted the `Single Grouping` strategy was the best overall strategy. This strategy used less disk accesses in order to answer range queries. The `n-Random Grouping`, `Max Capacity Grouping` and `k-Medoids Grouping` strategies were similar and obtained better results than the `Adaptive Grouping` strategy. However, for R-tree and $\epsilon = 0$, the NRG strategy showed improvement in higher dimensionality. In our tests we noted the adaptive strategy usually found too many groups thus this approach was similar to performing non-batched range queries.

Results also shown large batches of sequences have positive impact on the number of disk accesses. For example, the combination of R-tree, batch of size 1000 and `Single Grouping` strategy used approximately 2^{10} times less accesses to disk for range queries with $\epsilon = \frac{1}{4}\sqrt{E(D)}$. The same behavior happened when using M-tree indexes.

We also analysed the ratios of distance calculations [1]. These results have shown when $\epsilon = 0$ the distance calculations increase considerably and correlates with dimensionality. At the other hand, the number of distance calculations reduces when $\epsilon = \frac{1}{4}\sqrt{E(D)}$ using R-tree.

5. Conclusions

We evaluated 5 query grouping strategies in order to verify whether its use could be employed to improve performance of batch-mode range queries using R and M-trees. Our results suggest the best strategy is the one which groups every sequence in a single group and performs a single range query. For range queries that retrieves considerable amount of results, the use of grouping strategies greatly improves its performance by using any of the addressed trees. We suggest this is due to the ratio of disk accesses per query is very high and the number of calculations needed is similar to performing the naive approach. We noted the number of sequences in a single batch is an important factor to perform multiple range queries simultaneously. The larger the batch, the less disk accesses are performed. Considering all results, we suggest usage of single grouping strategy for all scenarios except for querying R with $\epsilon = 0$.

References

Camoglu, O., Kahveci, T., and Singh, A. K. (2003). Towards index-based similarity search for protein structure databases. In *Computational Systems Bioinformatics*, pages 148–158. IEEE.

Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *VLDB '97*, pages 426–435. M. K. Publishers Inc.

Faloutsos, C., Ranganathan, M., and Manolopoulos, Y. (1994). Fast subsequence matching in time-series databases. In *SIGMOD Rec.*, volume 23, pages 419–429. ACM.

Fu, A. W.-C., Keogh, E., Lau, L. Y., Ratanamahatana, C. A., and Wong, R. C.-W. (2008). Scaling and time warping in time series querying. In *The VLDB Journal*, volume 17, pages 899–921. Springer-Verlag New York, Inc.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *SIGMOD Rec.*, volume 14, pages 47–57. ACM.

Kotsifakos, A., Karlsson, I., Papapetrou, P., Athitsos, V., and Gunopulos, D. (2015). Embedding-based subsequence matching with gaps-range-tolerances: A query-by-humming application. In *The VLDB Journal*, volume 24, pages 519–536. Springer-Verlag New York, Inc.

Moon, Y.-S., Whang, K.-Y., and Loh, W.-K. (2001). Duality-based subsequence matching in time-series databases. In *Proc. 17th Int. Conf. Data Engineering*, pages 263–272. IEEE.

Orchard, M. T. (1991). A fast nearest-neighbor search algorithm. In *ICASSP 91: 1991 Int. Conf. Acoustics, Speech, and Signal Processing*, volume 4, pages 2297–2300. IEEE.

Park, H.-S. and Jun, C.-H. (2009). A simple and fast algorithm for k-medoids clustering. In *Expert Systems with Applications*, volume 36, pages 3336–3341. Elsevier.

[1] Figure is not shown due to space constraints