Erschienen in: Advances in Intelligent Data Analysis XVI Proceedings / Adams, Niall; Tucker, Allan; Weston, David (Hrsg.). - Cham : Springer, 2017. - (Lecture Notes in Computer Science ; 10584). - S. 87-98. - ISBN 978-3-319-68764-3
https://dx.doi.org/10.1007/978-3-319-68765-0_8

Bucket Selection: A Model-Independent Diverse Selection Strategy for Widening

Alexander Fillbrunn[1,2(✉)], Leonard Wörteler[1], Michael Grossniklaus[1], and Michael R. Berthold[1,2,3]

[1] Department of Computer and Information Science, University of Konstanz, 78457 Konstanz, Germany
{alexander.fillbrunn,leonard.worteler, michael.grossniklaus,michael.berthold}@uni-konstanz.de
[2] Konstanz Research School Chemical Biology (KoRS-CB), Konstanz, Germany
[3] KNIME AG, 8005 Zurich, Switzerland

Abstract. When using a greedy algorithm for finding a model, as is the case in many data mining algorithms, there is a risk of getting caught in local extrema, *i.e.*, suboptimal solutions. Widening is a technique for enhancing greedy algorithms by using parallel resources to broaden the search in the model space. The most important component of widening is the selector, a function that chooses the next models to refine. This selector ideally enforces diversity within the selected set of models in order to ensure that parallel workers explore sufficiently different parts of the model space and do not end up mimicking a simple beam search. Previous publications have shown that this works well for problems with a suitable distance measure for the models, but if no such measure is available, applying widening is challenging. In addition these approaches require extensive, sequential computations for diverse subset selection, making the entire process much slower than the original greedy algorithm. In this paper we propose the *bucket selector*, a model-independent randomized selection strategy. We find that (a) the bucket selector is a lot faster and not significantly worse when a diversity measure exists and (b) it performs better than existing selection strategies in cases without a diversity measure.

1 Introduction

An important component of many algorithms that need to traverse a large model space is greedy search. Data mining algorithms such as hierarchical clustering, rule set induction, and decision tree learning all rely on its ability to find good enough solution states quickly. In this setting, the increasingly available parallel computing resources can be leveraged in two ways. First, proposals have been made to use these resources to improve the *runtime performance* of greedy search algorithms, *i.e.*, to find the same results faster [2,6]. A second use of parallel computing resources that has been studied is to improve the *task-based performance* of greedy algorithms by addressing their shortcomings, namely their tendency to

get caught in local extrema. These techniques propose to search the model space in parallel, maintaining and improving multiple diverse (possibly incomplete) solutions until a better final model is found.

In this paper, we focus on this second class of greedy search algorithms and, in particular, on the technique of *widening* [1], which aims to maintain diverse models with little or, ideally, no communication between the parallel workers. In order to prevent workers from pursuing similar solutions because they show promise at first, widening relies on a distance measure and a selection strategy that together enforce diversity. However, deciding which distance measure and selection strategy to use for a given problem can be challenging itself.

On the one hand, designing a measure that selects good and diverse models from a large model space is a multi-objective optimization problem, which, in general, is difficult to solve. Nevertheless, widening has shown very promising results for problems that have a suitable model-dependent distance measure. Examples of such problems include the set cover problem [9], KRIMP [12], Bayesian networks [13], and hierarchical agglomerative clustering [7].

On the other hand, the state-of-the-art selection strategies either require a distance matrix for the models or the models to be sorted according to their quality. These preconditions put severe limits on the independence of individual parallel workers as they have to exchange their models frequently, resulting in a large increase in runtime over the sequential greedy algorithm.

In order to address these two challenges, we propose a model-independent diverse selection strategy for widening, the so-called *bucket selector*. We argue that our selection strategy is a powerful technique to enable widening for problems that have no straightforward distance measure or that require more independence of workers than state-of-the-art selection strategies can provide. As an example of such a greedy search problem, we use the set covering problem as a demonstrator and the ordering of joins during query optimization in relational database systems [14].

In summary, this paper makes the following contributions:

- We propose the bucket selector, a novel selection strategy that uses randomized partitioning in order to facilitate exploration of the model space while still exploiting promising areas (Sect. 4).
- In our evaluation, we demonstrate that (a) the performance of the bucket selector is not significantly worse for problems that have a well-known model-dependent distance measure, (b) it outperforms existing selection strategies for problems without such a measure and (c) in both cases, it finds results faster than other strategies (Sect. 5).

We begin in Sects. 2 and 3 by giving an overview of related work and an introduction to widening, respectively. Concluding remarks are given in Sect. 6.

2 Related Work

Work focusing on the diversity of models has been done mostly in the areas of ensemble learning and genetic algorithms, but interesting approaches can also

be found in satisficing planning, where the objective is to find a *good enough* solution for a given problem.

A very popular and powerful method of utilizing parallel resources is the learning of multiple predictors concurrently, *i.e.*, ensemble learning, resulting in a group of models that leverages the *wisdom of the crowd* by combining the individual models' predictions, *e.g.*, by (weighted) majority vote. *Bagging* [4] is a method that is often used to ensure that the learners specialize in different areas of the feature space. In ensemble learning, promoting diversity among the individual members is an important part of building the joint classifier, as multiple very similar predictors would not contribute any additional knowledge. In general the diversity among members of an classifier ensemble is perceived to be high if the predictors' outputs differ. This makes sense for an ensemble, as a single model does not necessarily need to be good for every part of the input space, but overfit and focus on a fraction of it instead. When the goal is to select one generally good model in the end, this notion of diversity does not suffice.

Genetic algorithms provide a more fitting approach here, as they usually maintain a population, but the goal is to find one best solution. In order to avoid premature convergence of the population the selection of individuals to take over into the next iteration often consists of the fittest and some less fit individuals. Additionally, techniques such as fitness sharing [8] can be employed to enforce exploration of the model space.

Other approaches presented in previous widening publications use various distance metrics to enforce diversity. For set covering Ivanova *et al.* [9] use the Jaccard coefficient, Sampson *et al.* use the Frobenius Norm of the difference of the graphs' Laplacians to compare Bayesian networks [12] and an optimization based on p-dispersion-min-sum for KRIMP [13]. Fillbrunn *et al.* [7] compare incomplete hierarchical clustering trees by using the Robinson-Foulds metric. Most selection strategies presented in those publications are either computationally too expensive to be feasible for use in greedy algorithms due to them having to sort the models or build distance matrices, or they require extensive communication between workers.

3 Widening

Widening is an improvement for greedy algorithms that learn a model by iteratively refining it. Formally, such a sequential algorithm performs the following update on the model $m \in \mathcal{M}$ in each iteration:

$$m' = s(r(m)). \tag{1}$$

Here, r is the refinement function that builds all possible successors for a model and s is a selection function that selects a single model, usually based on some heuristic $\psi : \mathcal{M} \mapsto \mathbb{R}$ for the model's quality. The update rule is applied either until the model does not change anymore, cannot be refined further, or is deemed good enough. For a widened and parallel algorithm the update

rule is extended for multiple models and the selection function is split into a thread/worker local and a global part:

$$\{m'_1, \ldots, m'_k\} = s_{global}\left(\bigcup_{i=1}^{k} s_{local}(r(m_i))\right), \tag{2}$$

where k is the number of models held by parallel workers and refined individually. The selection function is applied locally to each worker and then globally to the union of these selected refinements to produce the next k models. We call k the *width* of the algorithm. Instead of a greedy search through the model space, this approach resembles a beam search, enabling broader exploration of the model space while still exploiting promising areas thoroughly. For $k = 1$ a widened algorithm behaves greedily and if k is large enough the model space is searched exhaustively. If the selection operator simply selects models based on ψ, the solutions found by the individual workers are likely to either converge or be very similar, leading to inefficient use of compute resources and a lack of exploration. We call this simple and fast approach *top-k*. Widening aims to counteract this behavior by making the workers choose sufficiently different models $m'_1, \ldots, m'_k$. But at the same time, the models must be good according to the heuristic, so that the workers do not optimize solely based on diversity. The solution to this multi-objective optimization problem is the responsibility of the selection function s.

Ivanova *et al.* [9] introduced the diverse top-k selector which selects the best k models that are also sufficiently diverse from one another. While the top-k selection strategy does not enforce diversity at all, diverse-top-k cannot be run as efficiently because in each operation the models have to be sorted by quality. Additionally, it relies on a model-specific distance measure that is not always easy to find.

In the next section, we present the bucket selector, a randomized selection strategy that does not require a distance function, is easily parallelizable with little communication between the workers, and shows promising results in our initial performance study.

4 The Bucket Selector

The bucket selector enhances diversity by selecting not just the best models, but by partitioning the models into buckets and selecting the best model from each bucket. In which bucket a model is placed is determined by a hash function $h : \mathcal{M} \mapsto [0, k)$. The ith model selected by s_{local} and s_{global} is therefore:

$$m'_i = \max_{m:h(m)=i} \psi(m).$$

Because the same hash function is used locally and globally, $s_{local} = s_{global}$ as is also the case for the top-k selector.

The choice for the hash function is obviously problem-dependent. In addition to the usual properties of hash functions (even distribution of hash values, same hash value for identical models) it would also be desirable to guarantee an even distribution of hash values *in each step* of the algorithm and possibly also equal hash values for *semantically* equivalent models. We plan to investigate these issues in subsequent work.

Using the bucket selector, we achieve diversity by using the hash function, while also ensuring that the same model is not selected twice. By fixing the bucket number for which a worker is responsible, we can further limit the necessary inter-worker communication to the transfer of the best model in a bucket to the respective worker node, avoiding the central collection of models altogether. For $k = 3$ this means that Worker 1 sends the best model in Bucket 2 to Worker 2 and the best model in Bucket 3 to Worker 3. Workers 2 and 3 do the same for Buckets 1 and 3 or 1 and 2, respectively. As a consequence, $k \cdot (k-1)$ models have to be transferred in each iteration.

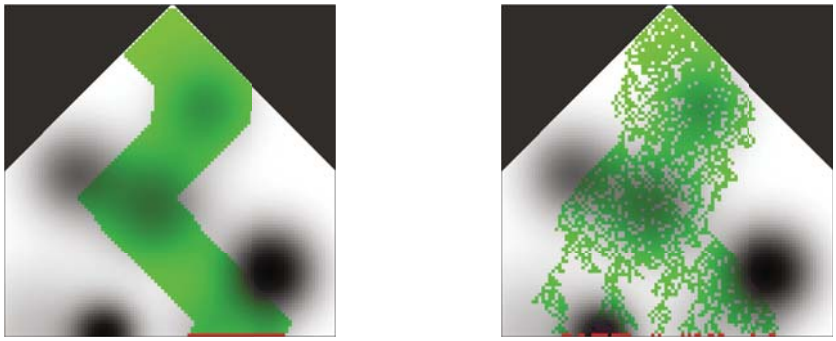
Given k workers and n models, the time complexity of the hash bucket selector is $\mathcal{O}(n + k^2)$: n for finding the best model in each bucket locally and k^2 because each worker gets all models in its bucket and must find the best one.

Partitioning the models based on their hash value ensures that the same models produced by different workers still end up in the same bucket and the selection step is deterministic. A downside of the hashing is the fact that the models may not be evenly distributed across buckets or buckets may even stay empty if k is large and the number of refinements is small. To ensure that approximately the same number of models competes for the top spot in each bucket, we can randomly shuffle the model list and assign a model m_j at index j to a bucket using the function $h \mapsto j \bmod k$. In contrast to hash partitioning the shuffle approach is sensitive to duplicate models: a good model that is sorted into multiple buckets may also be the best model in all of them and therefore reduce the diversity of the selected models. Adding explicit deduplication increases the time complexity to $\mathcal{O}(nk+k^2)$ because for each model that is put at the top spot of a bucket, the top spots of all other buckets have to be checked for a duplicate.

If determinism is important, it is also possible to hash the models into $l > k$ buckets and iteratively merge the smallest buckets until k buckets are left. The complexity for this approach is $\mathcal{O}(n(l-k) + k^2)$ and its exploration behavior compared to that of top-k is shown in Fig. 1. It should be noted that if the number of refinements is much larger than k, the probability of empty buckets is very small and bucket merging is not necessary.

5 Evaluation

In this section, we present the results of an initial study that we carried out by applying the bucket selector to the set cover problem and the join ordering problem for query optimization in relational database systems. The set cover problem serves as a baseline to evaluate the bucket selector on a problem with a known good distance measure, whereas the join ordering problem is used as an



(a) Top-k selector (b) Hash bucket selector (l=60)

Fig. 1. Exploration behavior of the top-k selector and the hash bucket selector with bucket merging. The space is searched from top to bottom. One pixel denotes one model that is refined by moving to one of the 3 pixels below it. The darker a pixel, the lower is the model's cost.

example of a problem that has no straightforward distance measure. For both of these problems, we analyze and compare different selectors w.r.t. the quality of the solutions they find and their runtime performance with different grades of parallelization.

5.1 The Set Cover Problem

The set cover problem is an NP-hard problem from combinatorics [11, p. 414]. Given a collection of sets $\mathcal{S}$ whose union spans a universe $\mathcal{U}$, the goal is to find the smallest subset $\mathcal{C} \subseteq \mathcal{S}$ that still spans the whole universe.

An iterative algorithm for solving the problem greedily has been given by Johnson *et al.* [10]. The algorithm starts with an empty temporary cover $\hat{\mathcal{C}}$ and iteratively adds the set from $\mathcal{S}$ that covers the largest number of currently uncovered elements of $\mathcal{U}$. Given a model with temporary cover $\hat{\mathcal{C}}$ and the set of sets that are not yet taken $\mathcal{S}' = \{s | \forall s : s \in \mathcal{S}, s \notin \hat{\mathcal{C}}\}$, we can refine it and find set $\mathcal{R}$ of all possible new temporary covers with:

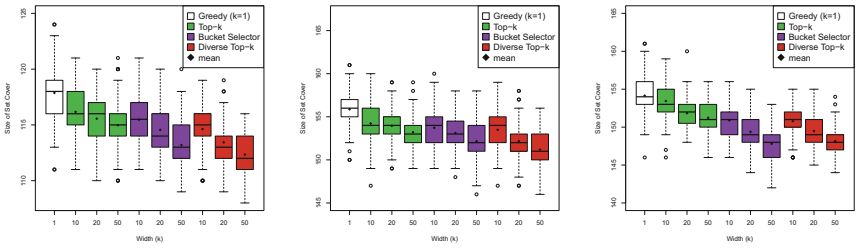
$$\mathcal{R} = \bigcup_{t \in \mathcal{S}'} \hat{\mathcal{C}} \cup t \ .$$

From this set we can then select the k covers that cover the largest part of $\mathcal{U}$ for the next iteration. For a heuristic estimation of a cover's quality we can therefore simply use the size of the union of the sets contained within it, *i.e.* the size of the partial cover.

We compute the hash values of our models based on the bit set containing the indices of the sets contained in the cover. In our implementation we make use of the `java.util.BitSet#hashCode()` method.

5.2 Set Cover Results

We evaluated the set covering performance of the bucket selector on three benchmark data sets: *rail507*, *rail516*, and *rail582* [3], all of which are from a real-life train-crew scheduling problem. We ran each experiment 200 times, shuffling the order in which we presented the data to the algorithm each time. When models are sorted by quality, ties between two models are broken based on the smallest set-index that is contained in the cover[1]. For the diverse top-k selector we used a threshold based on the *jaccard index* as described by Ivanova *et al.* [9].



(a) rail507 (b) rail516 (c) rail582

Fig. 2. Results of solving the set cover problem with different selectors, data sets and k. The bucket selector uses a random partitioning.

As can be seen in Fig. 2, increasing the number of models maintained in parallel has a positive effect on the final size of the found set cover. Diverse Top-k with $k = 10$ even achieves a slightly better mean set cover size than Top-k with $k = 50$. We also see that the bucket selector performs better than the top-k algorithm in all datasets, but worse than the diverse top-k approach for *rail507* and *rail516*. While the improvements the bucket selector brings over Top-k seem small, it has to be noted that this comes at no additional computational cost, as will be shown in Sect. 5.5.

5.3 The Join Ordering Problem

In contrast to the set cover problem, the join ordering problem for query optimization in a relational database system does not have a representation with a straightforward distance measure. We therefore demonstrate for this problem how join queries can be optimized using widening and the bucket selector.

The join ordering problem has the form $\langle \mathcal{T}, \mathcal{P} \rangle$, where $\mathcal{T}$ are the tables to be joined and $\mathcal{P}$ are predicates of the form $\langle A, B, s \rangle$, connecting tables A and B with

[1] Due to the nature of the set cover problem and the chosen heuristic, models with the same score occur frequently and other tie-breaking methods may be feasible. This is, however, out of scope of this work.

selectivity s. The selectivity denotes the fraction of rows retained from the cross-product $A \times B$, so that the resulting table has the cardinality $|A \bowtie B| = |A| \cdot |B| \cdot s$.

In order to find an efficient execution plan for a database query comprising multiple joins, the query optimizer needs to enumerate all orders in which the elements of $\mathcal{T}$ can be joined together into one join tree, *i.e.* all binary trees where the leaves are elements of $\mathcal{T}$ and the inner nodes are joins with zero or more predicates from $\mathcal{P}$. We disregard commutations (*i.e.*, $A \bowtie B \equiv B \bowtie A$), which reduces the number of candidates to inspect. For N tables there are $1 \cdot 3 \cdot 5 \cdot \ldots \cdot (2N-3) = (2N-3)!! \in \Omega\left(4^N/N^{3/2}\right)$ distinct trees. For each of these join orders, the query optimizer estimates the cost of the corresponding query execution plan based on a cost model that represents CPU and I/O overhead. For a large number of tables ($N > 15$) the exhaustive enumeration of all possible table combinations is infeasible and query optimizers typically need to restrict the search space or resort to greedy or genetic algorithms to find good plans. As we focus exclusively on the ordering of joins in this paper, the sizes of the original and intermediate tables is the main factor that influences the overall cost of a query execution plan.

In this section, we show that widening with the bucket selector performs favorably compared to the top-k selector and the simple greedy algorithm, suggesting that diversity is an important factor in the search for good join orders.

In the widening framework a model for a join optimization problem has the form $\langle \mathcal{R}, \mathcal{P} \rangle$ where $\mathcal{R}$ are relations (tables or joins) and $\mathcal{P}$ are the remaining predicates. In the initial model each element of $\mathcal{R}$ is a table ($\mathcal{R} = \mathcal{T}$). To refine an unfinished model $m = \langle \mathcal{R}, \mathcal{P} \rangle$ we take each predicate $P = \langle R_1, R_2, s \rangle$ with $R_1, R_2 \in \mathcal{R}$ from $\mathcal{P}$ and create a new model $m' = \langle R_1 \bowtie R_2 \cup \mathcal{R} \setminus R_1, R_2, \mathcal{P} \setminus P \rangle$. The cost of m' is then: $cost(m) + |R_1| \cdot |R_2| \cdot s$.

In the selection step we can use this formula for our quality heuristic and select the k models with the lowest cost estimation, *e.g.*, $\psi(m) = \frac{1}{cost(m)}$.

The hash function for join trees is computed by first assigning a unique random number to every table and calculating hash values of inner nodes in a bottom-up fashion using the formula $h(a \bowtie b) = 31 \cdot (31 \cdot j(a \bowtie b) + h(a)) + h(b)$, where $j(\cdot)$ returns the number of joins in a join (sub-)tree. The hash value of the whole tree is that of its root node.

5.4 Join Ordering Results

We evaluate our bucket selector on the join optimization problem with randomly generated tables and join predicates. For this evaluation, we use a random bucket selector that first puts one random model in each bucket and then assigns random buckets to the remaining models, *i.e.*, each bucket always has at least one model, given the number of models is greater or equal k. The hash bucket selector uses a hash code based on the join tree structure.

We limit our evaluation to joins in a snowflake schema, a topology that is common in data warehouses. A snowflake schema consists of a large *fact table*, which is connected to multiple *dimension tables*, which in turn have connections to further, smaller tables. We selected this use case because it is known to be

challenging: in a snowflake schema, optimal plans often involve cross-products of the smaller tables, which are typically not enumerated by exact optimizers to reduce the search space.

In our experiments the fact table has a cardinality between 500,000 and 1,000,000 rows, the dimension tables have a cardinality between 500 and 1000 rows and the outer tables have a cardinality between 10 and 100 rows. The predicates for the joins between the fact and the dimension tables have a selectivity between 0.0001 and 0.0005 and the dimension tables are connected to the outer tables by predicates with a selectivity between 0.01 and 0.05. Each experiment is repeated 1000 times with random predicate selectivities and cardinalities within the given bounds. We also allow cross products, which are treated as joins with a predicate that has a selectivity of 1.0.

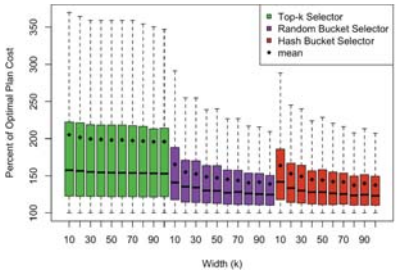
For a total of 16 tables arranged in a snowflake topology (1 fact table, 3 dimension tables and 12 outer tables) we can compare our results to the optimal plan. As can be seen in Fig. 3a both the random bucket and hash bucket selector outperform the top-k selector in terms of plan costs. We also see that as k increases the bucket approaches show greater improvement: while the top-k selector's mean plan costs improve from 205% at $k = 1$ to 196% at $k = 100$, the random bucket selector improves from 165.3% to 139% and the hash bucket selector from 163.8% to 137.5%.

For a larger number of tables the calculation of the optimal plan becomes infeasible. In order to evaluate the bucket selector on such an example, we follow a similar approach as described by Bruno *et al.* [5] and compare the plans found by the bucket selector to the plans found by the top-k selector. Figure 3b depicts the plan costs of the bucket selector relative to the top-k selector. Both the random and hash bucket selector perform better, with a mean improvement of about 10%. The higher the width k of the search is, the larger is the benefit the bucket selector has. With $k = 10$ the hash bucket selector produces on average plans with 89.9% of the top-k selector's cost and with $k = 100$ this cost sinks to 83.9%.

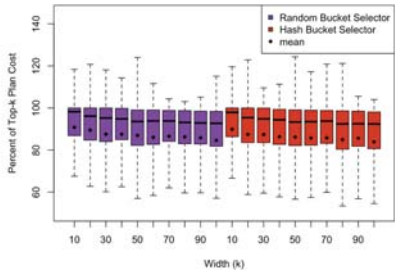
In both experiments, we see that the hash bucket selector produces slightly better plans than the random bucket selector. This is due to the imperfect handling of duplicate models as explained in Sect. 4: a model that occurs multiple times in the refinements can be the top of multiple buckets if they are assigned randomly, thus resulting in less than k selected models. As we can see in the next section, the random bucket selector in return offers a slightly better runtime without explicit deduplication.

5.5 Notes on Runtime

Apart from solution quality, the runtime of the search algorithm plays just as important a role. This especially concerns join plan generation, where the latency with which a database query optimizer can return a result to the user is the most crucial factor. In the following section, we evaluate the time the optimization algorithm needs to find a solution for different selectors. All benchmarks are

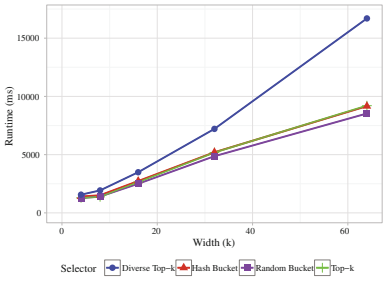


(a) 1 fact table, 3 dimension tables and 12 outer tables; comparison with optimal plan

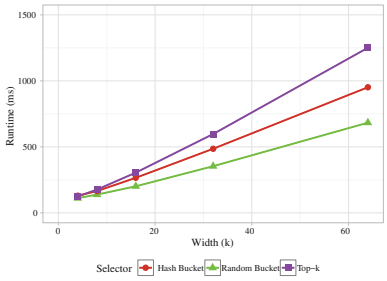


(b) 1 fact table, 5 dimension tables and 25 outer tables; comparison with Top-k

Fig. 3. Plan costs achieved with different selectors on join order optimization with 16 (a) and 31 tables (b).



(a) Set covering on rail507



(b) Join order optimization on snowflake topologies with 31 tables

Fig. 4. Runtime for various values of k in set covering and join order optimization.

performed on a machine with 64 GB of main memory and an Intel Core i7-5930K CPU with 6 physical and 6 virtual cores. For our experiments, we use the Java Microbenchmarking Harness (JMH)[2] to minimize the influence of Java's Just-in-Time compiler on the measurements. We run each experiment 200 times in batches of 20 iterations, creating a new virtual machine and performing five warm-up iterations for every batch.

For the set covering problem we run experiments with the *rail507* data set and compare the top-k, diverse top-k, hash bucket, and random bucket selectors. In Fig. 4a we see that diverse top-k is considerably slower than the other selectors and that the random bucket selector slightly outperforms the hash bucket and the top-k selectors in terms of runtime. That the hash bucket selector is as fast as the top-k selector, despite theoretically better runtime complexity, is likely a result of the rather time consuming calculation of the hash function.

Our runtime evaluation of the join order optimization problem is conducted on randomly generated snowflake topologies with 1 fact table, 5 dimension tables

[2] http://openjdk.java.net/projects/code-tools/jmh/ (1/26/2017).

and 25 outer tables. We compare the top-k selector and the hash and random bucket selectors. As depicted in Fig. 4b, both bucket selection strategies are faster than the top-k approach, the random bucket selector finding a solution almost twice as fast for $k = 64$.

6 Conclusion

We have proposed the bucket selector—enforcing diversity with a random or hash-based partitioning of intermediate models—as a favorable alternative to simple top-k selection, providing better results in terms of model quality and runtime. We also found that bucket selection cannot beat the solution quality of the diverse top-k selection strategy in the set covering problem, where a good distance measure for the models exists. We thus see the bucket selector as a middle ground, being better than simple top-k and providing the benefit of model independence and improved runtime over diverse top-k selection.

As an example for a problem without a known good model distance measure, we presented our results for the heuristic optimization of query plans for snowflake joins. This problem seems to benefit from diversity enhancement during search, as we can quickly generate query plans that have on average 83.9% of the costs of the plans found by the top-k approach. Our results motivate further work in the area of join order optimization concerning tests with other table topologies and the implementation of parallel query optimizers.

Acknowledgements. This work was partially funded by BMBF (grant 031A535C) and the Konstanz Research School Chemical Biology.

References

1. Akbar, Z., Ivanova, V.N., Berthold, M.R.: Parallel data mining revisited. Better, not faster. In: Hollmén, J., Klawonn, F., Tucker, A. (eds.) IDA 2012. LNCS, vol. 7619, pp. 23–34. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34156-4_4
2. Amado, N., Gama, J., Silva, F.: Parallel implementation of decision tree learning algorithms. In: Brazdil, P., Jorge, A. (eds.) EPIA 2001. LNCS, vol. 2258, pp. 6–13. Springer, Heidelberg (2001). doi:10.1007/3-540-45329-6_4
3. Beasley, J.E.: OR-Library: distributing test problems by electronic mail. J. Opl. Res. Soc. **41**(11), 1069–1072 (1990)
4. Breiman, L.: Bagging predictors. Mach. Learn. **24**(2), 123–140 (1996)
5. Bruno, N., Galindo-Legaria, C.A., Joshi, M.: Polynomial heuristics for query optimization. In: Proceedings of International Conference on Data Engineering (ICDE), pp. 589–600 (2010)
6. Zhihua, D., Lin, F.: A novel parallelization approach for hierarchical clustering. Parallel Comput. **31**(5), 523–527 (2005)
7. Fillbrunn, A., Berthold, M.R.: Diversity-driven widening of hierarchical agglomerative clustering. In: Fromont, E., De Bie, T., van Leeuwen, M. (eds.) IDA 2015. LNCS, vol. 9385, pp. 84–94. Springer, Cham (2015). doi:10.1007/978-3-319-24465-5_8

8. Goldberg, D.E., Richardson, J.T.: Genetic algorithms with sharing for multimodal function optimization. In: Proceedings of International Conference on Genetic Algorithms (ICGA), pp. 41–49 (1987)
9. Ivanova, V.N., Berthold, M.R.: Diversity-driven widening. In: Tucker, A., Höppner, F., Siebes, A., Swift, S. (eds.) IDA 2013. LNCS, vol. 8207, pp. 223–236. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41398-8_20
10. Johnson, D.S.: Approximation algorithms for combinatorial problems. J. Comput. Syst. Sci. **9**(3), 256–278 (1974)
11. Korte, B., Vygen, J.: Combinatorial Optimization. Algorithms and Combinatorics. Springer, Heidelberg (2013)
12. Sampson, O., Berthold, M.R., Widened, K.: Better performance through diverse parallelism. In: Proceedings of International Symposium on Intelligent Data Analysis (IDA), pp. 276–285 (2014)
13. Sampson, O.R., Berthold, M.R.: Widened learning of Bayesian network classifiers. In: Boström, H., Knobbe, A., Soares, C., Papapetrou, P. (eds.) IDA 2016. LNCS, vol. 9897, pp. 215–225. Springer, Cham (2016). doi:10.1007/978-3-319-46349-0_19
14. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of International Conference on Management of Data (SIGMOD), pp. 23–34 (1979)