Function Interpolation for Learned Index Structures

Naufal Fikri Setiawan, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic

School of Computing and Information Systems,
University of Melbourne, Australia
[naufal.setiawan, brubinstein, renata.borovica]@unimelb.edu.au

Abstract. Range indexes such as B-trees are widely recognised as effective data structures for enabling fast retrieval of records by the query key. While such classical indexes offer optimal worst-case guarantees, recent research suggests that average-case performance might be improved by alternative machine learning-based models such as deep neural networks. This paper explores an alternative approach by modelling the task as one of function approximation via interpolation between compressed subsets of keys. We explore the Chebyshev and Bernstein polynomial bases, and demonstrate substantial benefits over deep neural networks. In particular, our proposed function interpolation models exhibit memory footprint two orders of magnitude smaller compared to neural network models, and 30-40% accuracy improvement over neural networks trained with the same amount of time, while keeping query time generally on-par with neural network models.

Keywords: Indexing · Databases · Function Approximation.

1 Introduction

Databases use indexes to organise data for fast data retrieval, with B-trees and variants offering optimal worst-case lookup being the most popular. Viewed through the lens of machine learning, querying a B-tree is analogous to a model prediction, wherein a specific query key—an instance feature vector—is mapped to a record's location—a predicted label. Recent research has introduced the tantalising possibility of replacing classical range indexes with a model learned from data (*e.g.*, a neural network [16]) to perform queries on a pre-sorted set of records, with the aim being to either reduce index space requirements or to improve average-case query time. More broadly, significant efforts have explored the possibility to develop approximate and data-aware structures for specialised purposes [7, 17].

For large datasets, maintaining a B-tree can become resource intensive in terms of I/O operations and space requirements. While the space complexity of a B-tree [2] is $\mathcal{O}(N)$ in the number of database records, query time is $\mathcal{O}(\log N)$—space costs are justified given that B-tree's perfect retrieval accuracy minimises subsequent access times. By comparison, deep neural networks [11] have proven capable of generalising in a variety of problem domains: even large network models are compact relative to dataset size, presenting an opportunity to strike desirable trade-offs between the space and time complexity for use in databases, assuming prediction accuracy can be controlled. One cost of such learners is construction time, in that their effective use requires hyperparameter tuning (*i.e.*, grid search over learning rates, network topology, activation functions, and regularisation strategies). While GPUs can be used to accelerate the training of deep neural networks in domains such as computer vision, commodity database servers rarely have access to such hardware [19].

The reduction of index construction and key lookup to supervised learning is via 1) pre-sorting record locators by the key in an auxiliary data structure such as an array or B-tree, and 2) learning to predict a rank from the key. Normalised by the dataset size, this corresponds to fitting a monotonic function with co-domain the unit interval. A core observation in [16] is that such functions are cumulative distribution functions. Based on the association with probability theory, they apply tools from machine learning in supervised regression.

This paper introduces classical polynomial interpolation methods as an alternative approach to supervised regression when learning index models. We note that not only is there no need to generalise from training data to unseen test data in such a case—the typical requirement in statistical learning theory [23]—but that there is no latent population distribution on keys, rather a set (with uniform measure). As such we argue for a function approximation view [5] as opposed to a statistical learning view.

We examine both Chebyshev and Bernstein polynomial bases for function approximation. We find that these methods can outperform single neural network models in terms of query time. Polynomial models also generally occupy less space than neural networks. Fitting of a polynomial model, in most observed cases, also requires fewer resources and less time than do training of neural networks. A key computational requirement of neural networks, that can go unreported, is hyperparameter tuning which also contributes to learned index construction cost. On the contrary, the only hyperparameter needed to create a polynomial model is just the specified degree m.

In the following we discuss related work pertaining to learned and classical indexes. Section 3 summaries necessary background in data structures and polynomial interpolation methods. We describe our main contribution, an approach to index structures based on polynomial approximation, in Section 4. Experimental results are presented in Section 5, and Section 6 offers our conclusions.

2 Related Work

Indexes have for decades attracted the interest of database researchers and practitioners due to substantial query processing speed-ups on offer. We here discuss recent efforts pertaining to reducing the size of indexes through data-, workload- and hardware-aware optimisations or recently introduced learned models.

Space-Aware Indexes. Since traditional B+trees consume a substantial amount of memory space, many alternative approaches have explored techniques to reduce their size, including prefix or suffix truncation, or key normalisation [10, 12]. The past decade has also witnessed an increasing number of techniques where the index structure is adjusted to fit the properties of modern hardware, such as CSB+ trees, FAST trees, and Adaptive Radix Trees [21, 18, 15]. Partial and adaptive indexes similarly aim to reduce the memory footprint of indexes by building an index over a subset of data only, driven by user queries [24, 14]. Such techniques are orthogonal to (and could be extended with) our approach that uses function interpolation to approximate the positions of the keys within the leaves of the index.

Learning Indexes. When it comes to reducing the size of indexes the closest to our work is the line of research on approximating indexes with statistical machine learning models, such as learning indexes [16], fitting trees [7], or hybrid models such as sandwiched bloom filters or interpolation friendly B-trees [13, 20]. While the former two use learned models to actually replace the index structure to some extent (they still may use small indexes at the lowest levels to improve approximation accuracy), the latter two focus on improving the index performance by extending indexes with learned models as "helper functions".

3 Background

We now briefly overview the framing of range indexes as approximating a cumulative distribution function, and summarise key results from approximation theory.

3.1 Range Indexes as Cumulative Distribution Functions

The B-tree, and range indexes in general, can be seen as a model that maps a key to a position on disk with perfect accuracy. As records stored in a B-tree require ordering on a column, the positions of records are proportional to a cumulative distribution function (CDF): a monotonic increasing function mapping key space into $[0, 1]$. Suppose we have N records in the database and we are querying a specific key k, then we may conclude that the requested records position as:

$$pos = N \times \Pr(x \leq k) \ . \tag{1}$$

Any model that can approximate the function $f(k) = \Pr(x \leq k)$ can replace the role of a B-tree, provided that an error correction step follows predictions in order to retrieve the data when the model guesses an inaccurate position. This observation was made in [16], where deep neural networks (DNNs) were proposed as a means to approximate cumulative distribution functions.

The modelling choice of (regularised) supervised learning as in a typical DNN [16] presumes that the task of fitting $f(k)$ is one of *inductive learning* wherein the ultimate goal is to minimise risk as measured by expected loss of predictions on random draws of labelled examples from a latent population distribution.

We argue further in Section 4 that on fixed datasets[1] the problem of indexing does not require the model to extrapolate outside of existing data. That is, making good predictions on future unseen data is irrelevant to approximating $f(k)$ on existing and known keys.

3.2 Polynomial Interpolation

Interpolation uses a family of functions with uniform domain $\mathcal{B} = \{f_1, f_2, \cdots\}$ (a *basis*) such that any function to be approximated ψ that shares the same domain as the f_i satisfies $\psi(x) \approx \lim_{n\to\infty} \sum_{n=0}^{\infty} \alpha_n f_n(x)$ for some α_i coefficients. Compare this situation with supervised regression. Function approximation seeks accurate reconstruction over the entire domain of target ϕ, while supervised regression trains to fit on a small finite sample of training instances and aims to generalise to any likely inputs in the future.

Since function approximation approaches adopt specific but fixed function bases, only the vector of the coefficients $\langle \alpha_0, \alpha_1, \cdots, \alpha_N \rangle$ need be stored per target ϕ. This allows an interpolation polynomial to act as a lossy compression of a B-tree. The number of coefficients that need to be stored depends on the rate of convergence of the interpolation function and the accuracy that is desired. Most polynomial approaches require that target ϕ be continuous and piecewise smooth in order to provide theoretical guarantees on accuracy, but not in order to yield some approximation. To implement interpolation as a compressed learned index, we simply interpolate over the function ψ that maps a numeric key to a position in an auxiliary array (as the database may be sorted based on different keys rather than the one we are using for the index).

We next summarise the two major polynomial interpolation methods used in the remainder of this paper.

Chebyshev Interpolation Method The Chebyshev polynomials of the first kind are used in numerical methods in which good approximations and error bounds are needed [4], for example in the solution of least-squared problems. Chebyshev interpolation is regarded as accurate, due to its ability to minimise Runge's phenomenon as with other polynomials that sample from the Chebyshev nodes (the oscillation behaviour that can occur between sample points of the function being interpolated) [3].

Definition 1 (Chebyshev Polynomial). *The Chebyshev polynomials of the first kind are defined by the recurrence relation*

$$T_n(x) = \begin{cases} 1 \ , & \textit{if } n = 0, \\ x \ , & \textit{if } n = 1, \\ 2xT_{n-1}(x) - T_{n-2}(x) \ , & \textit{o.w.} \end{cases} \quad . \tag{2}$$

[1] It is sufficient but not necessary to prohibit insertions/deletions as done in [16].

Proposition 1. *An equivalent expression for the Chebyshev polynomials is* $T_n(x) = \cos(n \arccos x)$ *for* $x \in [-1, 1]$*.*

To express other functions in terms of the Chebyshev polynomials we project into the basis by performing the discrete Chebyshev transformation on the function ψ that we want to interpolate [9].

Definition 2 (Discrete Chebyshev Transform). *The coefficients of the Chebyshev polynomial of the first kind (of degree* N*) that interpolates* ψ *are given by:*

$$\alpha_i = \frac{p_i}{N} \sum_{k=0}^{N-1} \psi\left(-\cos\left(\frac{\pi}{N}\left(k+\frac{1}{2}\right)\right)\right) \cos\left(\frac{m\pi}{N}\left(N+k+\frac{1}{2}\right)\right) \quad , \tag{3}$$

where $p_0 = 1$ *and* $p_i = 2$ *when* $i > 0$*.*

The Chebyshev interpolation method is used to estimate the **erf** function that is evaluated as part of the cumulative distribution functions of the normal and log-normal distributions [22]. This fact will prove convenient later as the keys in datasets may be distributed in one of these ways.

Bernstein Interpolation Method The Bernstein polynomials form another important basis. They were originally used in a constructive proof of the Stone-Weierstrass approximation theorem which states that any continuous function can be uniformly approximated by a polynomial. They are also used as the basis for Bézier curves and privacy-preserving function release [1].

Definition 3 (Bernstein Polynomials). *The* v*-th Bernstein polynomial of order* n *is defined by the expression*

$$B_v^n(x) = \binom{n}{v} x^v (1-x)^{n-v} \quad , \tag{4}$$

which corresponds to the Binomial probability mass function representing the probability of observing v *heads out of* n *i.i.d coin flips each with heads probability* x*.*

Definition 4 (Bernstein Interpolation). *Let* ψ *be an arbitrary continuous function with domain* $[0, 1]$*. The order* n *Bernstein interpolation of* ψ *is defined by*

$$B_n[\psi](x) = \sum_{i=1}^{n} \binom{n}{i} \psi\left(\frac{i}{n}\right) x^n (1-x)^{n-i} \quad , \tag{5}$$

which corresponds to the expectation $\mathbb{E}_{V \sim \text{Binom}(n,x)}[\psi(V)]$*.*

4 Indexes by Function Approximation

We next detail the construction and application of indexes based on polynomial interpolation.

As observed in Section 3, the existing literature on learned indexes leverages inductive learning: fitting models on existing (training) data to minimise loss on future (test/population) data.

To see why inductive learning is an inappropriate formulation of the range index problem, consider the B-tree and its classical structure variants. The B-tree is prevalent in database systems despite it being an (efficient) lookup table. It does not generalise to new, unseen keys, in that an existing B-tree cannot be used to 'guess' the locations of keys of records not yet stored or encountered.

The astute machine learning reader may then wonder whether our goal should be one of *transductive* learning [8] in which one seeks to minimise loss on specific, given, test cases. While this appears closer to our task, the only test keys we seek to accurately query are in the training set. Further, there is no randomness in the locations of stored records, as would warrant supervised learning. Therefore we advocate for learning without accounting for population sampling or label randomness—pure function approximation.

4.1 Interpolant Construction

To obtain from a database D, a cumulative distribution function for structure construction, we must first extract the (sorted) set of keys. Suppose we choose a column K to be summarised with a range index, then an intermediary array is needed that allows us to map the key that we choose with the actual stored position of the record. Start by generating an array A of pairs $\langle key, pos \rangle$ that is sorted on *key*. Refer to Figure 1(a). The position of the entry $\langle key, pos \rangle$ in A will then define an unnormalised cumulative distribution function.

Formally, cumulative distribution functions are right continuous, monotonic and have range space minimum 0 and maximum 1, as depicted in Figure 1(b). Both polynomial interpolation methods considered require the target function ψ to be continuous and piecewise smooth (*i.e.*, accuracy of the approximation $\psi(x) \approx \lim_{i \to \infty} \sum_{n=0}^{\infty} f_i(x)$ will not typically be guaranteed when ψ is not piecewise smooth and continuous). For this reason, we transform the step-function cumulative distribution function seen in Figure 1(b) to a piecewise linear, continuous function as in Figure 1(c). In this step, we also rescale the key from the domain of $[K_{\mathrm{MIN}}, K_{\mathrm{MAX}}]$ to the preferred domain of the interpolation function $[I_{\mathrm{MIN}}, I_{\mathrm{MAX}}]$. In this paper, this is taken as the unit $[0, 1]$ for the Bernstein interpolation method, and $[-1, 1]$ for the Chebyshev interpolation method.

Lastly we choose a degree n and interpolation method (either Chebyshev or Bernstein) for application to the smoothed cumulative distribution function to obtain the parameters α_i for the chosen polynomial basis coefficients. The dashed line $\psi(x)$ in Figure 1(d) is the original piecewise linear function that passes through all the data points, the (orange) interpolated polynomial example here is $B_5[\psi]$.

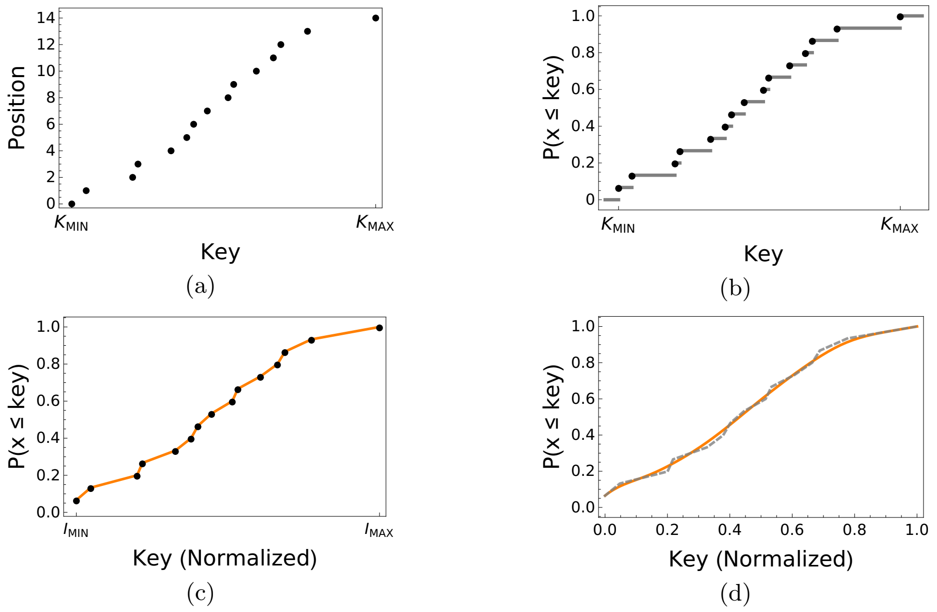

Fig. 1. Transforming (a) key positions to (b) normalised but step-function CDF, (c) piece-wise linear continuous smoothed CDF, (d) polynomial-interpolated CDF.

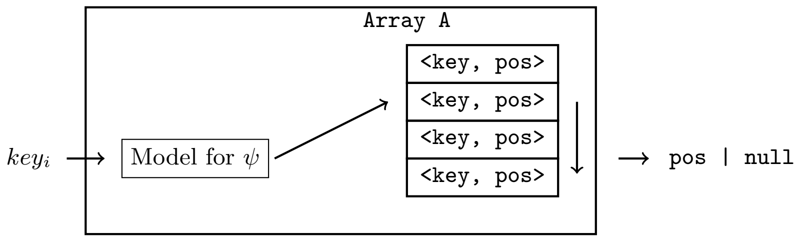

Fig. 2. The polynomial model creates a prediction $\hat{pos}_A$, then the linear seek error correction algorithm starts looking for the entry with *key* starting at $\hat{pos}_A$.

4.2 Query Processing

The lookup process of any record in the database then consists of three steps depicted in Figure 2 and described as follows:

1. Prediction. First we predict the position of a query *key* as a location inside the auxiliary index array A. We call this prediction $\hat{pos}_A$.

2. Error Correction. As predictions are only approximate, some correction/follow-up search might be necessary. Any incorrect prediction is corrected by linearly seeking through the auxiliary array until the requested key is found or it can be concluded that the key does not exist.

3. Retrieval. After the $\langle key, pos \rangle$ mapping is found in the auxiliary array we can retrieve the record from its actual position on disk.

5 Experimental Results

To investigate how polynomial interpolation performs compared to conventional indexes such as the B-tree, and neural network learned indexes, we performed several experiments reported here. We first explore the model creation time, memory footprint, and query accuracy and time, and then present a sensitivity analysis of interpolation methods with respect to their interpolation degree.

The neural network is set to have 2 hidden layers and 32 neurons. We opted not to use a GPU to reflect the setup of most common database servers—all training and computation was performed by CPU. Neural networks were implemented using the `torch` package [6] and the B-Tree data structure was implemented using the `OIBTree` Python module.

Datasets. We use three different datasets each containing two million entries and build a neural network index on top of it as well as a regular B-tree. These three datasets are created randomly to follow the **uniform**, **normal** and **log-normal** distributions.

Hardware. All experiments are performed on a commodity laptop with 4× Intel i7 CPU cores and 16 GB of RAM. For our implementation we use Linux 5.2.14 and `CPython 3.7.4` running on `gcc 9`.

5.1 Model Creation Time

In our first experiment, we benchmark the time needed to create each polynomial model from degree 1 to degree 50. We use the notation B_n and C_n for Bernstein and Chebyshev polynomials of degree n respectively. We have repeated the experiment on each dataset ten times in succession and report the average time across all, since the overall observed discrepancy was less than $100ms$. Neural networks are not included in this experiment since they are able to be trained further for better accuracy while interpolation models have their accuracy and parameters 'set in stone' after creation.

Table 1. Model creation time (in seconds) results for B-tree, and Polynomial indexes.

Number of Entries	B-tree	B_5	B_{10}	B_{15}	B_{20}	B_{25}	C_5	C_{10}	C_{15}	C_{20}	C_{25}
500,000	7.102	0.786	0.773	0.762	0.765	0.783	0.527	0.581	0.634	0.727	0.820
1,000,000	14.953	1.570	1.551	1.583	1.557	1.547	1.093	1.205	1.362	1.551	1.760
1,500,000	23.611	2.357	2.357	2.350	2.366	2.357	1.661	1.850	2.081	2.395	2.691
2,000,000	34.575	3.279	3.286	3.371	3.277	3.366	2.324	2.631	2.942	3.245	3.809

The results in Table 1 show that on existing data, the polynomial models are able to be created significantly faster than B-trees (**by a factor of 10**) for a specified number of entries. This is due to operations performed when assembling the B-tree, since rebalancing a B-tree can be expensive. The time complexity of creating a fresh, full B-tree index is $\mathcal{O}(n \log n)$, where n is the number database records. On the contrary, for m degree of the polynomial approximation, creation of the Bernstein polynomial is fixed $\mathcal{O}(n)$, improvable to $O(m \log n)$, while creation of the Chebyshev polynomial is $\mathcal{O}(m^2 \log n)$.

5.2 Memory Footprint

We next evaluate the size of the polynomial models in comparison to the B-tree and neural network structures, across multiple datasets. We obtain consistent results for all three datasets and hence report only the average results. While conducting this experiment, it is important to note that B-trees do not need to store the $\langle key, pos \rangle$ pairs inside an auxiliary array as these pairs are already stored in the leaves of the B-trees themselves. In this experiment, we refer to the storage of the pair as the 'data segment'. In the case of a B-tree, the term 'data segment' refers to the leaf slots where the $\langle key, pos \rangle$ is stored. In the case of the polynomial models, 'data segment' refers to the auxiliary array.

Table 2. Memory footprint results for B-tree, and Neural Network (in megabytes MB and kilobytes KB).

Dataset Entries	B-tree (MB)	Neural Network (KB)
500K	33.034	210.73
1M	66.126	210.73
1.5M	99.123	210.73
2M	132.163	210.73

Table 3. Memory footprint results for the polynomial index models (in bytes B).

Dataset Entries	B_5	B_{10}	B_{15}	B_{20}	B_{25}	C_5	C_{10}	C_{15}	C_{20}	C_{25}
500K	1016	1240	1632	1832	1400	1632	1882	1352	1392	1442
1M	1016	1240	1632	1832	1400	1632	1882	1352	1392	1442
1.5M	1016	1240	1632	1832	1400	1632	1882	1352	1392	1442
2M	1016	1240	1632	1832	1400	1632	1882	1352	1392	1442

Table 2 and Table 3 show the memory footprint of the alternatives with the data segment stripped off. This metric shows how much memory overhead is added for the index structure. The overhead introduced by B-tree scales with the size of the data as pointer overheads are needed. The polynomial models require the storage of coefficients only. Similarly, neural networks have a fixed number of parameters, causing them to be independent of the size of database.

According to Table 3, the size of polynomials does not increase with the size of the dataset. This property is also exhibited by the neural networks, although the memory footprint of the polynomial indexes is still **2 orders of magnitude lower**. The higher memory footprint in the case of neural networks is attributed to the larger number of parameters that a neural network has.

5.3 Query Accuracy and Time

We next examine the retrieval accuracy using the polynomial models, and compare them against the classical B-Tree and Neural Network. We present the model prediction time, root mean squared error (RMSE) of the prediction, and the total query time (involving all three steps discussed in Section 4.2).

Table 4. Prediction time (in *ns*), prediction root-mean-squared-error (RMSE) and total query time (in *ns*) for Normal (No), Uniform (Un) and Log-normal (Ln) datasets.

Dataset	Prediction Time (ns)			RMSE			Query Time (ns)		
Model Type	No	Un	Ln	No	Un	Ln	No	Un	Ln
B_5	133	46.6	148	25805.22	109.55	80214.85	23500	133	92200
B_{10}	158	71.9	189	18014.59	84.90	70304.07	16500	111	65300
B_{15}	196	103	238	14155.66	70.70	57078.78	12400	133	58500
B_{20}	237	133	288	11687.71	68.70	47333.16	9780	151	48100
B_{25}	277	166	336	9973.57	62.58	39566.59	8080	192	40200
C_5	17.9	9.87	27.2	1430.38	57.92	12779.95	10.6	56.3	11800
C_{10}	20.1	11.0	28.4	166.03	52.71	2137.22	11.4	51.6	1860
C_{15}	22.8	12.8	29.2	65.02	45.031	1224.74	68.8	92.2	1020
C_{20}	22.9	14.6	30.4	60.39	28.53	951.37	62.0	48.1	751
C_{25}	25.9	16.4	31.7	57.14	26.39	474.905	62.1	40.2	415
B-Tree	24.4	41.5	40.1	N/A			31.5	56.3	46.0
Neural Network	433	148	806	105.84	22.67	711.12	402	516	1100

The time taken to predict a key is generally very low in the scenarios examined, being less than 500ns for all models. However, the total query times for all of the models are always far higher than the time taken simply to predict a key, since most of the cost of retrieving a key is in the error correction. Referring to Table 4, we see that the Bernstein polynomials are far less accurate compared to the Chebyshev polynomials, causing them to take a greater amount of time. A majority of the Bernstein polynomials take longer than our baseline models (*i.e.*, the B-tree and neural network models).

Some of the higher-order Chebyshev models are however significantly faster than the neural networks. Starting at C_{10} onwards the Chebyshev polynomials are able to outperform the neural networks in terms of speed. While the error is generally larger, the Chebyshev polynomial models do not have the operation overhead that neural networks have (attributed to activation functions and matrix multiplication) and are able to outperform the neural network as a result.

5.4 Rate of Convergence of Polynomial Models

The results presented in Figure 3 show the sensitivity of the polynomial models with respect to the degree increase.

For the normal dataset, the Bernstein polynomial converges more slowly than the Chebyshev polynomial model, and the Chebyshev polynomial of lower degrees performs far better than the Bernstein polynomial in lower degrees in

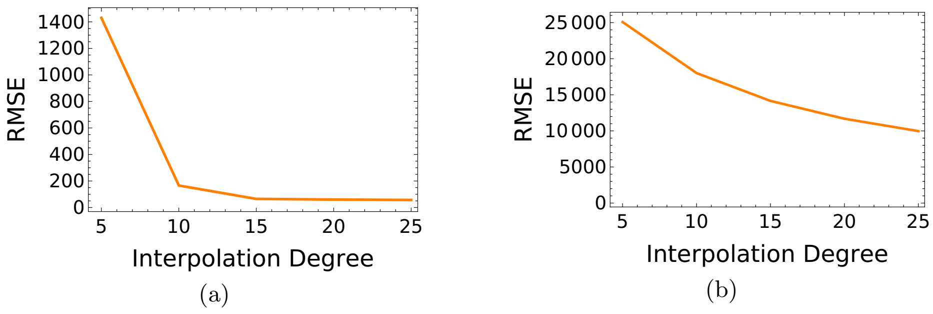

Fig. 3. Rates of convergence of the (a) Chebyshev and (b) Bernstein interpolations for the normally distributed dataset.

terms of accuracy. This, consequently leads to them being faster as there is less work required during error correction.

In the log-normal dataset (not presented here due to lack of space), both polynomial models have high errors relative to their performance in the other datasets (see Table 4). The Chebyshev model still performs better than the Bernstein model and still converges faster. To illustrate, C_{10} is 83% faster than C_5 while B_{10} is only 12% faster than B_5.

The uniform distribution is a special case where the Chebyshev model does not converge as fast as it does with the other data sets. However, errors are already minimal even with low-degree models and again, the Chebyshev model still performs better than the Bernstein model.

As seen from Figure 3(a), the fast rate of convergence of the Chebyshev polynomials allows us to accurately model the cumulative distribution function using a much smaller memory footprint. The Chebyshev interpolation method converges to an interpolant function at an exponential rate [3].

6 Conclusion

We advocate for a function approximation approach to range indexes as an alternative to learned indexes typified by deep neural networks. We argue that supervised learning approaches unnecessarily avoid overfitting in favour of generalisation, and unnecessarily model uncertainty in ground-truth labels. In the range index problem of databases, such considerations are inappropriate.

The two methods introduced in this paper—polynomial bases with corresponding interpolation/fitting operators—have lightweight overhead in construction time and memory footprint compared to neural networks. Moreover polynomial approximation techniques are far simpler to implement. As such, our methods represent feasible options as replacement models for learned indexes, and a tantalising direction for further investigation.

References

1. Aldà, F., Rubinstein, B.I.P.: The Bernstein mechanism: Function release under differential privacy. In: AAAI. pp. 1705–1711 (2017)
2. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. In: SIGFIDET. pp. 107–141 (1970)
3. Boyd, J.P., Ong, J.R.: Exponentially-convergent strategies for defeating the Runge phenomenon for the approximation of non-periodic functions, Part I: Single-interval schemes. Communications in Computational Physics **5**(2-4), 484–497 (2009)
4. Brisebarre, N., Joldeş, M.: Chebyshev interpolation polynomial-based tools for rigorous computing. In: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation. pp. 147–154. ACM (2010)
5. Cheney, E.W.: Introduction to approximation theory. McGraw-Hill (1966)
6. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A Matlab-like environment for machine learning. In: BigLearn NIPS workshop (2011)
7. Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., Kraska, T.: Fiting-tree: A data-aware index structure. In: SIGMOD. pp. 1189–1206 (2019)
8. Gammerman, A., Vovk, V., Vapnik, V.: Learning by transduction. In: UAI. pp. 148–155 (1998)
9. Gil, A., Segura, J., Temme, N.M.: Numerical Methods for Special Functions. Society for Industrial and Applied Mathematics (2007)
10. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: ICDE (1998)
11. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)
12. Graefe, G., Larson, P.A.: B-tree indexes and cpu caches. In: ICDE. pp. 349–358 (2001)
13. Hadian, A., Heinis, T.: Interpolation-friendly b-trees: Bridging the gap between algorithmic and learned indexes. In: EDBT. pp. 710–713 (2019)
14. Idreos, S., Kersten, M.L., Manegold, S.: Database cracking. In: CIDR. pp. 68–78 (2007)
15. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: Fast: Fast architecture sensitive tree search on modern cpus and gpus. In: SIGMOD. pp. 339–350 (2010)
16. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: SIGMOD (2018)
17. Kubica, J.M., Moore, A., Connolly, A.J., Jedicke, R.: Spatial data structures for efficient trajectory-based queries. Tech. Rep. CMU-RI-TR-04-61, Carnegie Mellon University (2004)
18. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: ICDE. pp. 38–49 (2013)
19. Microsoft: Hardware and software requirements for installing SQL server
20. Mitzenmacher, M.: A model for learned bloom filters, and optimizing by sandwiching. In: NIPS. pp. 462–471 (2018)
21. Rao, J., Ross, K.A.: Making b+-trees cache conscious in main memory. In: SIGMOD. pp. 475–486 (2000)
22. Schonfelder, J.: Chebyshev expansions for the error and related functions. Mathematics of Computation **32**(144), 1232–1240 (1978)
23. Shalev-Shwartz, S., Ben-David, S.: Understanding machine learning: From theory to algorithms. Cambridge university press (2014)
24. Stonebraker, M.: The case for partial indexes. SIGMOD Record **18**(4), 4–11 (1989)