The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures

Jens Dittrich
Saarland University
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

Joris Nix
Saarland University
Saarland Informatics Campus
joris.nix@bigdata.uni-saarland.de

Christian Schön
Saarland University
Saarland Informatics Campus
christian.schoen@uni-saarland.de

ABSTRACT

Index structures are a building block of query processing and computer science in general. Since the dawn of computer technology there have been index structures. And since then, a myriad of index structures are being invented and published each and every year.

In this paper we argue that the very idea of "inventing an index" is a misleading concept in the first place. It is the analogue of "inventing a physical query plan". This paper is a paradigm shift in which we propose to drop the idea to handcraft index structures (as done for binary search trees over B-trees to any form of learned index) altogether. We present a new automatic index breeding framework coined *Genetic Generic Generation of Index Structures (GENE).* It is based on the observation that almost all index structures are assembled along three principal dimensions: (1) structural building blocks, e.g., a B-tree is assembled from two different structural node types (inner and leaf nodes), (2) a couple of invariants, e.g., for a B-tree all paths have the same length, and (3) decisions on the internal layout of nodes (row or column layout, etc.). We propose a generic indexing framework that can mimic many existing index structures along those dimensions. Based on that framework we propose a generic genetic index generation algorithm that, given a workload and an optimization goal, can automatically assemble and mutate, in other words 'breed' new index structure 'species'. In our experiments we follow multiple goals. We reexamine some good old wisdom from database technology. Given a specific workload, will GENE even breed an index that is equivalent to what our textbooks and papers currently recommend for such a workload? Or can we do even more? Our initial results strongly indicate that generated indexes are the next step in designing index structures.

PVLDB Reference Format:
Jens Dittrich, Joris Nix, and Christian Schön. The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures. PVLDB, 15(3): 527 - 540, 2022.
doi:10.14778/3494124.3494136

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BigDataAnalyticsGroup/GENE.

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494136

1 INTRODUCTION

1.1 Problem 1: Indexes are considered monolithic entities

When we database researchers talk about indexes, we use the term *index* like referring to an entity of its own. But is that the case? Let's look at our good old B-tree: A *B-tree index* consists of two different node types: *inner nodes* and *leaves.* Inner nodes keep pointers to other nodes. The main purpose of an inner node is to route incoming lookups to other nodes. In addition, a B-tree index algorithmically preserves a couple of invariants, e.g. all paths from the root to a leaf have the same lengths, each node only has one parent node (i.e. nodes are structurally organized into a tree), and so forth. In addition, all nodes keep data in a specific layout (row or column layout, cache-and SIMD-efficient layouts, etc.) and define which search algorithm to use inside a node (binary search, interpolation, prediction, etc.). Since the publication of the original B-tree paper [6] almost 50 years ago, the physical organization of B-trees has been improved in a zillion different ways, e.g. [30, 42, 43, 46].

But what concretely is **the entity** "the index" in here? So far we only defined two different node types pointing to each other, we added a couple of constraints (fan-outs, tree-structure, concrete physical organization of inner nodes and leaves). We may also add heuristics for invariant maintenance (split and merge). But, if we change any aspect of this, do we receive a completely different index? When is it just a variant of an *existing* index? And when is it a *new* index? For instance, if we change constraints to allow nodes to have more than one parent, would that be a completely different index entity? Or is it just that one constraint that changes (with possible implications to other features of the index)?

In this paper, we will introduce the idea of logical and physical indexes. We will show that most existing indexes can be expressed as a specific *configuration* in a generic logical and physical indexing framework[1] including B-trees, radix-trees, learned indexes, and even extendible hashing. And those configurations can be combined almost arbitrarily *within the same configuration.* This opens the book for a myriad of hybrid "indexes". For instance, in our framework, one extreme of an index (say a single hash table) can smoothly be morphed into another extreme (say a B-tree style index with all kinds of different layouts and search algorithms inside its nodes).

[1]Note that we will not introduce this as a software framework as done in [11, 23] but rather on a conceptual level.

1.2 Problem 2: Two completely different methodologies to solve a similar problem

It is remarkable that there is quite a divide in databases when it comes to designing efficient components of a database system like index structures as opposed to designing query plans. For index structures, the historic and state-of-the-art approach is to define some performance goals, reason about complexities, design something on a blackboard, and then implement it. Like that an index (much like any other system component) has to be designed from scratch and then implemented. Eventually, we receive a piece of software that then (hopefully) serves the original purpose. In sharp contrast to this, since the 70s and the seminal Selinger paper [48] database researchers follow a completely different, and rather successful, design path when it comes to designing query plans: we automatically assemble complex plans from logical and physical operators.

So why follow two completely different design approaches if at the core these are similar problems? Once we are in the position to express an "index" as a configuration in a generic logical and physical indexing framework, there is one question left: Why should we configure indexes by hand anyways? Why should we handcraft which node type to use, which node-internal search algorithm to use, which data layout, tree-levels to use, etc.?

If we have different components of an index which can be interchanged freely, plus options to play with, well, then we have an optimization problem!

For this reason, in this paper, we will propose a genetic algorithm that, given a dataset and workload, will automatically determine a suitable logical and physical index configuration.

1.3 Problem Statement

We summarize the two principal problems discussed above into the following problem statement that we will investigate in this work:

(1) How can we generalize the most important index structures into a common conceptual indexing framework?
(2) How can we automatically breed index structures using (1).

1.4 Contributions

In this paper we make the following contributions:

(1) We introduce a generic index structure framework that makes a clear difference between a logical and a physical indexing framework. This is inspired by the split into logical and physical operators in relational and physical algebras/operators.
(2) We present a genetic algorithm which allows us to automatically generate (breed) efficient index configurations (aka indexes).
(3) We present an extensive experimental evaluation of our approach demonstrating that we can both rediscover existing, previously handcrafted indexes as well as new types of hybrid indexes.

The paper is structured as follows: in Section 2, we introduce our logical generic indexing framework. After that, in Section 3, we introduce our physical generic indexing framework. Both serve as the basis for Section 4 where we introduce our index breeding approach. Section 5 contrasts our approach to related work. Section 6 presents our experimental evaluation. We will conclude and point out a couple of exciting future research directions in Section 7.

2 GENERIC LOGICAL INDEXING FRAMEWORK

In this section we introduce our generic logical indexing framework. The physical indexing framework is explained in Section 3.

Descriptions of index structures tend to mix up logical (*what* is done) and physical aspects (*how* is that achieved). For instance, consider the following sentence taken from a popular textbook:

> "A sorted file, called the *data file*, is given another file, called the *index file*, consisting of key-pointer-pairs. A search key K in the index file is associated with a pointer to a data-file record that has search key K" [21, Section 13.1].

In this sentence the logical aspects of the index (black underlines, e.g. sorted, key, record) and the physical aspects of the index (red underlines, e.g. file, pointer) are introduced *at the same time* and thus mix up both aspects in the same explanation. In a way this violates physical data independence of the index structure. We want to clearly separate the logical and physical aspects of an index.

Basic Definitions. Any expression $\sigma_P(R)$ where P is a predicate defined on a relational schema $[R] : \{[A_1 : D_1, \ldots, A_n : D_n]\}$, i.e., a function $P : [R] \mapsto$ {true,false}, is called a *query* on R. The result of a query is $\sigma_P(R) \subseteq R$. Given $[R]$ with an attribute A_i with a corresponding non-categorical one-dimensional domain D_i, and two constants $l, h \in D_i, l \leq h$, $\sigma_{l \leq A_i \leq h}(R)$ is a *range query* on R. It selects all tuples $t = (a_1, .., a_i, .., a_n) \in R$ where a_i is contained in the interval $[l; h]$. A range query with $l = h$ is called a point query.

2.1 Logical Nodes and Logical Indexes

Definition 2.1. Logical Node. A logical node is a tuple (p, RI, DT):
(1) p : $[R] \to D$ is a **partitioning function** on the schema $[R]$ of the dataset to index, (p may be undefined),
(2) RI is the **routing information**. It is a function $RI : D \to \mathcal{P}(N)$ where N is a set of nodes and $\mathcal{P}(N)$ is the power set of N. In other words, each element of D (the target domain of p) is mapped to a subset of the nodes in N. For each outcome of the partitioning function p we can find a set of associated nodes or the empty set. Notice that the routing information does neither imply nor assume a specific physical organization including a sort order on its entries (like in B-trees). RI may be undefined. In the following, we use nodes(RI) for the set of nodes mapped to by RI.
(3) DT is the **data**. It is a set of tuples with relational schema $[R]$, DT may be empty[2].

Figure 1 visualizes the principal structure of a logical node. The partitioning function p computes $t.e$ mod 5 which yields a domain $D = \{0, 1, 2, 3, 4\}$. Here, only a subset of D is shown in the visualization of RI, i.e. 3 is not shown as it maps to the empty set. In addition, RI maps 2 and 0 to the same node. Moreover, the data part DT contains two tuples $(2, A)$ and $(1, B)$.

Definition 2.2. Complete Logical Index. Let LN be a set of logical nodes with $\forall_{n \in LN} : nodes(n.RI) \subseteq LN$. Then the graph $\lambda = (LN)$ is called a complete logical index.

[2]In principle, DT could also be defined as a similar function as RI the difference being that RI maps to nodes whereas DT maps to tuples.Also note that the DT-fields can be used to very naturally support buffer-tree-style indexes [3], bulkloading mechanisms [12] as well as any form of recursive partitioning algorithm.

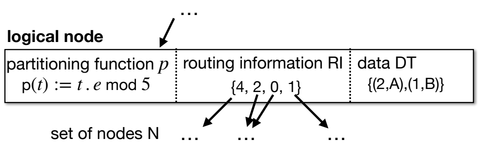
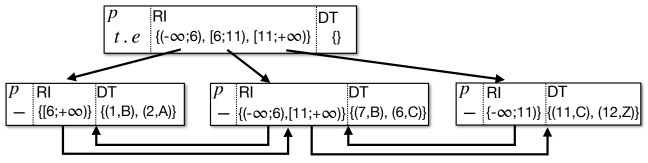
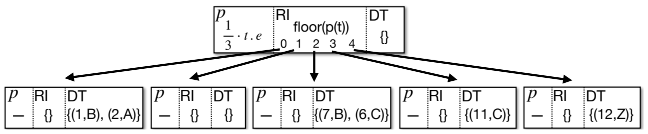


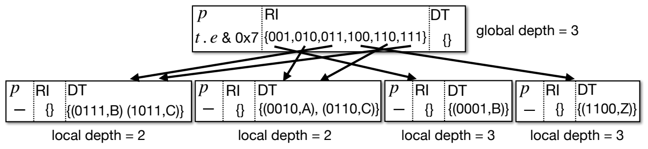
Figure 1: An example of a logical node with a hash-style partitioning function, four mappings in the routing information RI, and two tuples in the data part.



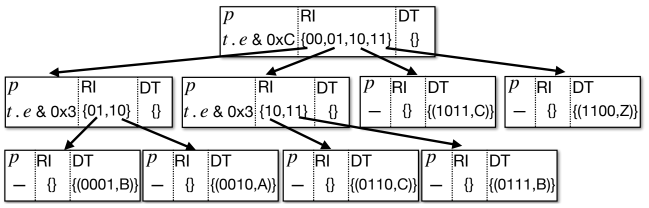
(a) **B-tree with ISAM:** Here the partitioning function returns $t.e$. The routing information maps ranges to nodes on the next level. This induces a B-tree-style partitioning. Notice that the common textbook explanation of B-trees showing k pivots and $k + 1$ pointers is already a specific physical implementation of this logical index. In addition, this index contains entries on the leaf-level for backward and forward chaining of leaves as in ISAM.



(b) **RMI:** Here the partitioning function is a linear function $p(t) = \frac{1}{3} \cdot t.e + 0$ that squeezes the data into a smaller range ([0;12] $\longrightarrow$ [0;4]). This is equivalent to a linear regression over the key space. RI groups the data into bins (corresponding to nodes on the next level). However, p and RI can be set to use any form of regression method and for any node independently.



(c) **extendible hashing:** Here the partitioning function only considers a suffix of the lowest three bits (&0x7) of $t.e$. This implies that it partitions exactly like an extendible hashing [16] directory with global depth of three. Note that there is no need to create entries for empty 'buckets'.



(d) **radix tree:** Here the partitioning functions partition the dataset on two adjacent bits each: the root-node partitions on the first two bits of the prefix, the next level on the next two bits. This induces a radix-partitioning. Note that in this example the index is configured to keep at most one tuple per leaf. This can of course be configured. So alternatively, we could force a two-level tree just partitioning on the first two bits. The second level would then keep multiple entries in their DT-fields.

Figure 2: The modeling power of our logical indexing framework for traditional indexes. Four special cases of possible logical indexes for the running example. All examples mimic existing and handcrafted (physiological) index structures.

In other words, only if all routing information in the nodes of LN points to nodes contained in LN, we call LN a complete logical index. At first, this definition sounds a bit trivial, but this definition makes an important observation that is frequently overlooked: a logical index **is-a** graph of logical nodes — and **nothing else**.

Running Example. Figure 2 illustrates the modeling power of our framework and shows *four possible* logical indexes for $[R] = \{[e : \text{int}, g : \text{char}]\}$ and $R = \{(2, \text{A}), (7, \text{B}), (1, \text{B}), (6, \text{C}), (12, \text{Z}), (11, \text{C})\}$. Note that in these examples the DTs are empty for inner nodes. The implications of non-empty DTs are future work. Figure 3 demonstrates how we can model arbitrary 'hybrid' logical indexes.

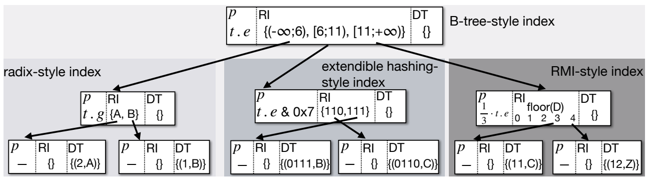


Figure 3: The modeling power of our logical indexing framework for any form of 'hybrid' index. The example combines properties from four different traditional index structures. Notice that there are countless examples: any node in this logical index may be exchanged by any other suitable logical node as long as the data in the index is partitioned in a way that all possible queries on the logical index return the correct result set. On this abstraction level it is still undefined *how* data is represented in the different nodes and in particular in the RI-function and the DT-set and *how* we search.

2.2 Logical Queries

Definition 2.3. RQ: Result of a Range Query on a Logical Index. Given a range query with predicate $P := l \leq A_i \leq h$, a logical index λ build upon a relation R and a non-empty start node-set $SN \subseteq LN$, the result set of the range query is given by:

$$\text{RQ}(P,SN) := \bigcup_{n \in SN} \left(\underbrace{\sigma_P(n.DT)}_{\text{data in } n} \cup \text{RQ}\Big(P, \bigcup_{t \in R, l \leq t.A_i \leq h} n.\text{RI}\big(n.p(t)\big) \Big) \right)$$

Notice that the set semantics will implicitly remove duplicates which in a physical graph-structured index (possibly not obeying set semantics) may result from visiting nodes multiple times.

Also note that this query will recursively traverse the graph for all qualifying nodes in the RI-fields. This is fine for a strictly tree-structured index, however, as soon as we do not have a tree-structure anymore but a more general DAG, it may become possible that, given a set of start nodes SN, certain nodes are reachable via *multiple* paths. For a general graph, the implementing algorithm has to be modified to not visit nodes multiple times.

Definition 2.4. Correctness of a Logical Index. Let $\lambda = (LN)$ be a complete logical index. Let SN be an arbitrary non-empty subset of start nodes: $SN \subseteq LN$. Let $DT_{\lambda:} = \bigcup_{n \in LN} n.DT$ be the data contained in λ. Let $\sigma_{P:=l \leq A_i \leq h}(R)$ be a *range query* on R. If

$$\forall_{l,h} : \sigma_{l \leq A_i \leq h}(DT_\lambda) = \text{RQ}(P, SN),$$

then λ is called a correct logical index w.r.t SN.

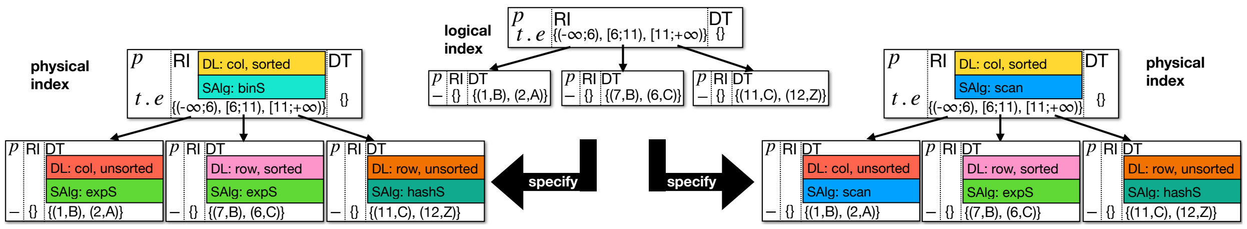


Figure 4: The arrows show some possible transitions from a logical to a physical index (we specify an algorithm and/or a data layout). Notice that neither the partitioning tree nor the assignment of data to nodes are changed in this process.

Notice that the correctness of an index depends on whether data is placed into the different DT-sets according to the properties of the different partitioning functions used at the various nodes. Furthermore, the start nodes *SN* must be chosen such that all qualifying data can be reached by the range query. For instance, in a tree-structured index picking the start node is trivial: we call it 'the root node'. In a general graph structure, which may even be disconnected, things can become more complex, i.e. we might have multiple 'root nodes', i.e. all nodes that cannot be reached from any other node of the index, or even no root nodes (in case of a cyclic graph). This discussion is beyond the scope of this paper and therefore in the following, we will only consider correct, DAG-structured indexes and assume that *SN* is chosen accordingly.

3 GENERIC PHYSICAL INDEXING FRAMEWORK

As we just have defined logical indexes (our counterparts to the logical relational algebra operators), now, we can proceed to devise physical indexes (our counterparts to physical operators).

For each logical node **and** *for each of its RI and DT-part* we eventually have to specify *how* to realize it. We do this by making a *physical* decision on the search algorithm (Section 3.1) and the data layout to use for that set (Section 3.2). Or, we delegate those decisions by using a nested index (Section 3.3).

Any index where for all its nodes the data layouts and algorithms are sufficiently specified, is called a *physical index*.

3.1 Specify Search Algorithm

We decide which search algorithm to use for searching (key/value)-pairs in RI and/or DT. Note that all search algorithms stop once a qualifying key was found, i.e. we found the corresponding entry in RI or we have an exact key match in DT. The principal options are as follows: **(1) scan:** linear search through all entries, for each key check if it qualifies, **(2) binS:** binary search **(3) intS:** interpolation search, iteratively compute slope and intercept, i.e. a linear function, for *left* and *right* key, predict key location *pred* and reduce search area to [left, pred] or (pred, right] respectively until *key* qualifies. **(4) expS:** exponential search, start with the first entry, increase exponent *i* for key position specified by 2^i until *key* is greater than the search value, use binary search (or any other suitable method) inside range $[2^{i-1}, \text{end}]$. **(5) hashS:** chained hashing (or any other suitable hashing variant), use the underlying hash function to compute the location of the *key* (and its associated mapping). **(6) linregS:** linear regression (or any other form of approximation and/or learning), compute slope and intercept, i.e. linear function, for all data points, compute error bounds, predict key location *pred* and use linear search (or any other suitable error correction method) inside [pred - lower error bound, pred + upper error bound]. **(7) hybridS:** any suitable hybrid algorithm (i.e. a composite of the former options).

3.2 Specify Data Layout

We decide which data layout to use for representing the data from RI and/or DT. To define a data layout, we have to specify the following: **(1) col vs row:** key/value-pairs are in row or col layout. **(2) func:** we use a function to specify the RI and/or DT-mapping, thus we do not need to represent pivots and/or data and therefore do not need a data layout. As discussed in Definition 2.1 already, we assume the DT-fields to be actual sets even though they could be modeled as a more general mapping as well. **(3) unsorted vs sorted:** the entries are (or are not) sorted by their key. **(4) comp:** the entries are compressed (and how exactly, i.e. which compression method). **(5) hybridDL** any suitable hybrid data layout (i.e. any composite of the former options). Notice that some of these data layout decisions cannot be made independently from the search algorithms to use, e.g. binary search implies a sorted data layout. Figure 4 shows an example of a logical index that by specifying the search algorithms and data layouts may be transformed into different physical indexes.

3.3 Specify by Nested Logical or Physical Index

We make a decision to specify RI and DT by a nested physical index. Notice that this is not equivalent to the recursively reachable set of nodes pointed to by one particular RI. Nesting is about representing the key/value-lookup search algorithms and data layout **inside** a node by another index. For instance, consider a physical binary search tree (BST). If we use such BST to represent and search RI, we basically have a nested physical index in our node. However, this is just a special case, so in theory we can allow for arbitrary nested indexes at this point.

4 GENETIC INDEX BREEDING

As we just have defined our logical and physical generic indexing frameworks, we proceed to present our genetic algorithm allowing us to automatically generate indexes. This is structured as follows:

(1) Core algorithm (Section 4.1),
(2) Initial population generation (Section 4.2),
(3) The set of applicable mutations describing possible changes to individual logical and physical index structures (Section 4.3), and
(4) The fitness function used to measure the performance of individual physical index structures (Section 4.4).

The major challenge with a generic indexing framework presented in Section 3 is the intractable search space. Therefore, we

Algorithm 1 Genetic Search Algorithm of GENE

```
1: function InitPopulation($DS$, $s_{\text{init}}$)
2:     $\Pi = \emptyset$                                  ▷ initialize population with empty set
3:     for ($i = 0; i < s_{\text{init}}; i++$) do          ▷ create $s_{\text{init}}$ initial indexes
4:         $\pi$ = buildAndPopulateRandomIndex($DS$)       ▷ build and populate index
5:         $\Pi = \Pi \cup \{\pi\}$                       ▷ add index to population $\Pi$
6:     end for
7:     return $\Pi$                                       ▷ return population $\Pi$
8: end function

9: function TournamentSelection($\Pi$, $s_T$, $W$)
10:    $T$ = sample_subset($\Pi$, $s_T$)                   ▷ draw random subset $T \subseteq \Pi$ of size $s_T$
11:    $\pi_{\min} = \arg\min_{\pi \in T} f(\pi, W)$       ▷ select fittest individual $\pi_{\min}$ in $T$ under $W$
12:    $\tilde{t}$ = median_fitness($T$)                   ▷ compute median fitness of all $\pi \in T$
13:    return ($\pi_{\min}$ , $\tilde{t}$)                 ▷ return fittest individual $\pi_{\min}$ and median fitness $\tilde{t}$
14: end function

15: function GeneticSearch($g_{\max}, s_{\text{init}}, s_{\max}, s_\Pi, s_T, s_{\text{ch}}, DS, MD, ND, W$)
16:    $\Pi$ = InitPopulation($s_{\text{init}}$, $DS$)     ▷ initialize population
17:    for ($i = 0; i < g_{\max}; i++$) do                 ▷ perform $r_{\max}$ iterations/generations
18:        ($\pi_{\min}$, $\tilde{t}$) = TournamentSelection($\Pi$, $s_T$, $W$)   ▷ run tournament selection
19:        for ($j = 0; j < s_{\max}; j++$) do             ▷ create $s_{\max}$ mutations
20:            $m$ = draw_mutation($MD$)                   ▷ draw from mutation distribution
21:            $n$ = draw_node($ND(\pi_{\min}, m)$)        ▷ draw from node distribution
22:            $ph$ = draw_phys($PD(m, n)$)                ▷ draw from phys distribution
23:            $\pi_{\text{mut}} = m(\pi_{\min}, n, ph)$   ▷ perform mutation
24:            if $f(\pi_{\text{mut}}, W) \leq \tilde{t}$ then   ▷ add $\pi_{\text{mut}}$ to $\Pi$ if fitter than median $\tilde{t}$
25:                if $|\Pi| \geq s_\Pi$ then              ▷ if capacity exceeded
26:                    $\Pi = \Pi \setminus \arg\max_{\pi \in T} f(\pi, W)$   ▷ remove unfittest individual
27:                end if
28:                $\Pi = \Pi \cup \{\pi_{\text{mut}}\}$   ▷ add index to population
29:            end if
30:        end for
31:    end for
32:    $\pi_{\min} = \arg\min_{\pi \in \Pi} f(\pi, W)$     ▷ return fittest individual of final population
33:    return $\pi_{\min}$
34: end function
```

need an optimization method that can cope with such a huge search space. Notice that an intractable search space does not imply that we cannot find a good solution. In fact, entire research communities work on these kind of problems including: planning, reinforcement learning, and genetic optimization. We decided to design our search algorithm based on genetic optimization. Genetic optimization algorithms have been developed for more than 40 years [24], but recently gained a lot of attention due to growing computational resources. They allow researchers to effectively explore larger search spaces. Recent surprising, and not widely-known, results include: *genetic algorithms can rediscover state-of-the-art machine learning algorithms*(!) [44]. Furthermore, they can devise yet unknown mathematical equations [9]. Genetic optimization tasks are very domain specific as possible mutations and the performance measure depend heavily on the concrete task.

4.1 Core Algorithm

The general design for our algorithm follows the principal of evolution which is known from nature: We start with the main function GeneticSearch (line 15). We start by initializing a *population* of individuals (line 16), in our case a set of physical index structures $\Pi := \{\pi | \pi \text{ is a physical index}\}$ (see function InitPopulation, line 1). To create the initial population, we build and populate s_{init} physical index structures (line 4) and add them to the population Π (line 5). This build process is described in more detail in Section 4.2. Now, we enter the central iteration: we perform $g_{\max}$ iterations in genetic search (lines 17–31). We start by tournament selection (line 18), see function TournamentSelection (line 9). We select a sample of size s_T of the current population Π (line 10) from which we select the fittest index $\pi_{\min}$ (line 11). We keep a trace of the fitness of physical indexes to never evaluate indexes multiple times. We compute the median fitness $\tilde{t}$ of sample T (line 12) and return both $\pi_{\min}$ and $\tilde{t}$ (line 13) to the GeneticSearch function (line 18). Then, we enter the mutation loop (line 19). The core idea is to compute $s_{\max} \geq 1$ mutations for index $\pi_{\min}$. We draw a random mutation m from a precomputed distribution of mutations MD (line 20). For the mutation m we draw a start node n to be used for this mutation (line 21) as well as a physical implementation ph (line 22). The mutations and distributions are described in detail in Section 4.3. Then, we perform the actual mutation on $\pi_{\min}$ (line 23) and receive π_{mut}. We originally also experimented with applying chains of mutations (lines 20 and 23) but it did not show any benefits. We check, whether the mutated index π_{mut} has a better fitness than the median $\tilde{t}$ (line 24). If it has a better fitness, we check if Π exceeds its capacity of maximum allowed physical indexes s_Π (line 25). If that is the case, we remove the physical index with the worst fitness from Π (line 26). Then we add π_{mut} to the population Π (line 28). Once the outer loop terminates, we determine the fittest index from Π (line 32) and return it.

Table 1: Symbols used.

Symbol	Meaning
λ	logical index
π	physical index
Π	population
s_{init}	initial size of the population
s_Π	maximum number of indexes in population
$g_{\max}$	number of generations
$s_{\max}$	number of mutations created and evaluated in a single iteration
s_T	size of sample in tournament selection
s_{ch}	maximum length of a mutation chain applied in one iteration
DS	dataset
$\pi_{\min}$	best individual in tournament selection
π_{mut}	mutated element
$\tilde{t}$	median fitness
MD	probability distribution of mutations
m	a single mutation
$ND(\pi, m)$	probability distribution of nodes
$PD(m, N)$	probability distribution of physical implementations
W	workload of queries
$f(\pi, W)$	fitness of a physical index

4.2 Initial Population Generation

What is a good start population Π for the genetic algorithm? In Algorithm 1, function InitPopulation (line 1), we need to define an initial population of individual index structures. There are several possible dimensions to consider. First, we can change the initial number s_{init} of indexes in Π. This basically defines how diverse the initial set of indexes may be. Second, we should determine how to actually build and populate the initial physical index with data from dataset DS (line 4). There are several options:

(1) We start with a single physical node that does not contain data, mutate it, and only then insert the actual data. We experimented with this approach initially but discarded it quickly due to its high training costs. Thus we do not support it in our algorithm anymore.
(2) We start with a single physical node containing all data. For data layout/search method we either randomly pick it or we pick one that we believe works well for the given workload.
(3) We use bottom-up bulkloading with the difference that for all nodes the search algorithms and data layouts are picked randomly.

In our current version we exclude hash nodes for inner nodes as we have not defined a radix-partition search method on this data layout yet. We will integrate this in future versions of our optimization framework. The resulting tree is logically similar to a standard B-Tree, the physical nodes however differ considerably.
(4) We start with a population containing a physical index that resembles a state-of-the-art hand-tuned index, i.e. we define the logical index (including its partitioning functions) as well as the physical nodes. Then we check whether we can still improve that index through our genetic algorithm.

Notice that for options from (1) to (4) increasing, we postulate that we take away load from GENE, using it increasingly as a refinement tool: The more we start with something already representing a very efficient (or fit, however fitness is defined) index, the more we expect that only small mutations will be performed by GENE. At least that is what we would believe. In fact, even if we (non-randomly) specify an initial physical index to start with, recall, that GENE has all degrees of freedom to pick mutations, and may surprise us by taking unexpected turns and make different decisions.

4.3 Mutations and their Distributions

In this section we introduce a suitable set of mutations and discuss how they are used in our algorithm.

Mutation. In our framework, a *mutation* is a function m : Index$\rightarrow$ Index. A mutation takes a single index as input, mutates it, and returns a modified index. By 'Index' we mean, that either a logical index (λ) *or* a physical index (π) is given and a mutated index is returned (λ_{mut} or π_{mut}). λ_{mut} and π_{mut} must preserve the correctness of λ and π. This is inspired by rewrite rules in classical query optimization: there we also only consider rules that are guaranteed to not change the query result. We will only consider mutations on tree-structured indexes. This is not a restriction of our generic framework but makes the following mutations a bit more digestible.

Mutation distributions. We use a probability distribution MD allowing us to assign different probabilities to the different mutations (line 20), e.g. we can prioritize certain mutations. Given a mutation m and a physical index $\pi_{\min}$ we draw from a second distribution $ND(\pi_{\min}, m)$ to determine the nodes N for this mutation (line 21). Now, we draw from a third distribution $PD(m, N)$ to determine which physical implementation to use for this mutation and nodes. Setting probabilities to zero within this distribution $PD(m, N)$ excludes invalid combinations of physical data layout and search method, e.g. binary search on unsorted data layouts. Note that these distributions can be created based on microbenchmarks.

Fundamental Mutations. Our goal is to implement a minimal set of mutations allowing to create a huge variety of physical indexes.

M$_1$ Change data layout: From n, we randomly select either its RI- *or* DT-part. Then we create a new physical node n' with data layout $n'.dl \neq n.dl$ drawn from $PD(m, N)$ with the same data and routing information as n: $n'.DT = n.DT \wedge n'.RI = n.RI$. The options for data layouts are described in Section 3.2. If n contains child partitions, we enforce the additional condition $n.dl' \neq$ hash, as our software framework does not (yet) support child partitions in nodes with a hash layout. In π, we replace n by n'. If $n'.s$ is incompatible with $n'.dl$, we draw a new method from $PD(m, N)$ to ensure correctness. Figure 5(a) shows an example: the input node n has a sorted column-layout. In the index, we replace n by n' which has a tree-layout.

M$_2$ Change search method: From n, we randomly select either its RI- *or* DT-part. Given the existing search method $n.s$, we draw an $s' \neq s$ from $PD(m, N)$. Then we create a new physical node n' with the new search method s' with the same data and routing information as n: $n'.DT = n.DT \wedge n'.RI = n.RI$. Figure 5(b) shows an example: the input node n uses a scan as search method. In the index, we replace n by n' which uses binary search.

M$_3$ Merge sibling nodes horizontally: We set node $n_{\text{parent}} := n$ whose RI maps to at least one other node in π, if not we abort this mutation. From the set of nodes mapped to by n_{parent} we randomly select a child node $n_{\text{target}} \in \text{nodes}(n_{\text{parent}}.\text{RI})$. We select a non-empty subset $N_{\text{sources}} \subseteq \text{nodes}(n_{\text{parent}}.\text{RI})$ of nodes to merge into n_{target} using the following restrictions: $n_{\text{target}} \notin N_{\text{sources}} \wedge \forall_{n \in N_{\text{sources}}} \; n.p = n_{\text{target}}.p$. This implies that the source domain of the routing information function D is equal for all nodes in $N_{\text{sources}} \cup \{n_{\text{target}}\}$. We then need to perform updates on two levels of the index: The node n_{target} that we merge with and the parent node n_{parent}. We start by describing the updates to the node n_{target}. First we update the data n_{target}.DT and set it to the union of all data within the merged nodes:

$$n'_{\text{target}}.\text{DT} = n_{\text{target}}.\text{DT} \;\; \cup \bigcup_{n \in N_{\text{sources}}} n.\text{DT}.$$

In the following, we also update the routing information function n_{target}.RI such that

$$\forall_{d \in D} n'_{\text{target}}.RI(d) = n_{\text{target}}.\text{RI}(d) \cup \bigcup_{n \in N_{\text{sources}}} n.\text{RI}(d),$$

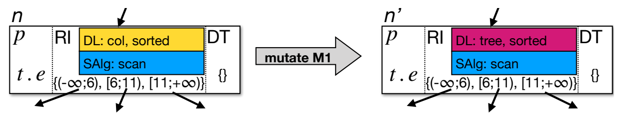
where D is the common domain of the RIs in $N_{\text{sources}} \cup \{n_{\text{target}}\}$. This ensures that our target node n_{target} now maps to all child nodes that any node $n \in N_{\text{sources}}$ previously mapped to, i.e. we can still reach all child nodes. For the parent node n_{parent} we have to update the routing information n_{parent}.RI such that

$$\forall_{d \in n_{D_{\text{parent}}}} \forall_{n \in N_{\text{sources}}} \; n \in n_{\text{parent}}.RI(d)$$

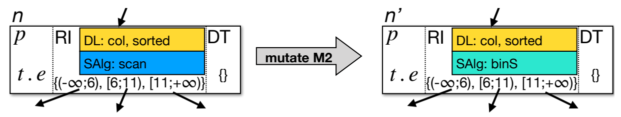
$$\Rightarrow n_{\text{parent}}.RI(d) = \{n_{\text{target}}\} \cup n_{\text{parent}}.RI(d) \setminus \{n\}.$$

In other words: We remove all mappings to merged nodes $n \in N_{\text{sources}}$ and replace them with a new mapping to the node n_{target}. Notice that the merge operation performed in B-trees is essentially just a specialized version of this general merge mutation. In a B-tree the number of merged nodes k is typically set to $k = 2$ and the nodes must be directly neighboring due to the sorted key domain. For our actual implementation, we also restrict ourselves similarly to merges where $|N_{\text{sources}}| = 1$. Merge operations with larger source-sets can easily be achieved by recursively executing the merge operation on the same node. Figure 5(c) shows an example: the set N_{sources} contains a single leaf that we want to merge into n_{target}. To achieve this we first merge all data contained in N_{sources}.DT into n_{target}.DT. As N_{sources}.RI is empty, we do not have to do anything here. In n_{parent}.RI, we need to remove the mapping to all nodes in N_{sources}, in this case the key-range $[2; 6) \subset D$ must be changed to map to n_{target}. For this example this is equivalent to merging the old entry $(-\infty; 2)$ with $[2; 6)$ into $(-\infty; 6)$. Now, all nodes in N_{sources} can be removed from the index.

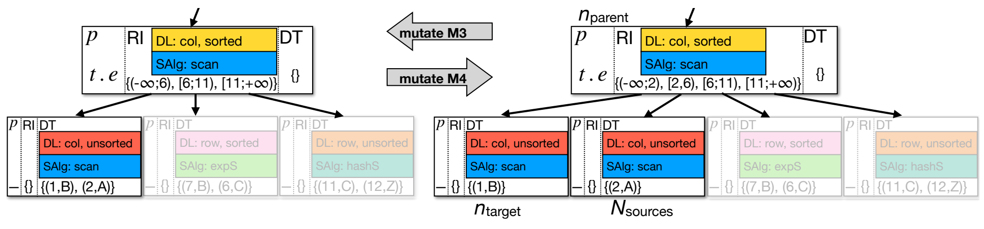
M$_4$ Split child node horizontally into k nodes: This is the inverse mutation of M$_3$. Figure 5(c) shows an example.



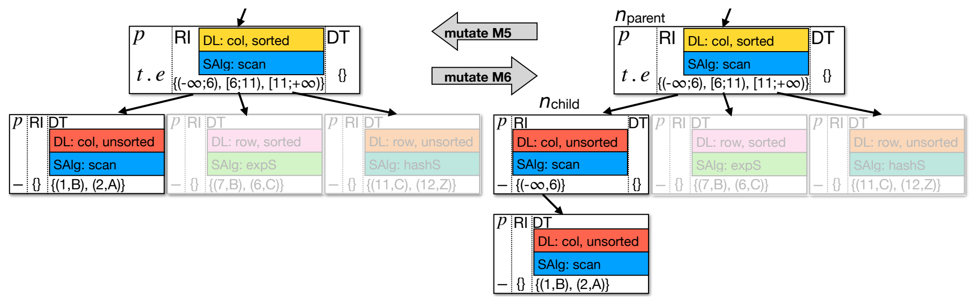
(a) M_1 **Change node type:** change data layout of RI.



(b) M_2 **Change search method:** change search of RI.



(c) M_3 **& M_4 Merge or split nodes horizontally:** merge left & middle child node (M3) or split leftmost child node (M4).



(d) M_5 **& M_6 Merge or split nodes vertically:** merge top-level node's left child (M5) or split it (M6).

Figure 5: Performing the mutations described in Section 4.3 on actual physical indexes.

M_5 Merge sibling nodes vertically: We set node $n_{\text{parent}} := n$ whose RI maps to at least one other node in π, if not we abort this mutation. From the set of nodes mapped to by n_{parent} we randomly select a child node $n_{\text{child}} \in \text{nodes}(n_{\text{parent}}.\text{RI})$ using the following restriction: $n_{\text{child}}.p = n_{\text{parent}}.p$. To merge n_{child} into n_{parent}, we then need to perform the following updates: First we need to move all data in n_{child}.DT to the parent node:

$$n_{\text{parent}}.\text{DT} = n_{\text{parent}}.\text{DT} \cup n_{\text{child}}.\text{DT}$$

In the following we need to move potential child nodes n' of n_{child} to the parent node n_{parent}:

$$\forall_{d \in D_{\text{parent}}}\ n_{\text{child}} \in n_{\text{parent}}.\text{RI}(d)$$
$$\Rightarrow n_{\text{parent}}.\text{RI}(d) = n_{\text{parent}}.\text{RI}(d) \setminus \{n_{\text{child}}\} \cup n_{\text{child}}.\text{RI}(d)$$

where D_{parent} is the domain of n_{parent}.RI. In other words: We remove all mappings to the merged node n_{child} and replace them with mappings to the child nodes of n_{child}. For our actual implementation, we restrict ourselves to the merge of a single parent-child-pair during a single mutation. Merge operations for longer chains of nodes can easily be achieved by recursively executing the merge operation on the same node. Figure 5(d) shows an example: We select the root node as n_{parent} and its left child node as n_{child} which we want to merge into the root node. To achieve this we first merge all data contained in n_{child}.DT into n_{parent}.DT. In n_{parent}.RI, we need to remove the mapping to n_{child} and replace them with mappings to the children of n_{child}. In this case, we remove the key-range $(-\infty; 6) \subset D$ and replace it with the corresponding entries of n_{child}.RI. For this example this is equivalent to re-inserting the entry $(-\infty; 6)$ into n_{parent}.RI.

M_6 Split child node vertically into k nodes: This is the inverse operation of M_5. Figure 5(d) shows an example.

4.4 Fitness Function

The fitness function is used to measure the performance of a single physical index and describes what to optimize by the genetic algorithm (either by minimizing or maximizing its value). Its definition can be chosen freely depending on the optimization goal. We have chosen to optimize our index structures for the runtime given a specific workload consisting of point and range queries. We therefore define the fitness function f: Physical Index $\times$ Workload$\rightarrow \mathbb{R}$ to be minimized in the following way: $f(\pi, W) = r(\pi, W)_c$. π denotes the physical index (the individual) to evaluate, W is a sequence of queries and denotes the workload of the specific experiment. $r(\pi, W)_c$ is the median runtime measured for this physical index on the workload over c runs. The fitness function can also easily be adapted to factor in other optimization goals like memory- or energy-efficiency. Other interesting extensions include regularization, i.e. index complexity could be punished (similar to model complexity in ML). Furthermore we could punish or incentivize the filling grade of leaves, e.g. if leaves are fully packed, this is beneficial for read-optimized indexes but for inserts can quickly lead to structural modifications of the tree. However, if leaves are only partially filled, many inserts can be handled by leaf-local changes. All these requirements can be modeled into the fitness function.

5 RELATED WORK

Handcrafted Indexes. Since the original B-tree-paper [6] in 1972, B-trees have become a workhorse in database systems. Since then a myriad of B-tree-variants and -improvements have been proposed [30, 42, 43, 46]. Other classes of handcrafted index structures include radix-trees like Judy-arrays [4] and its modern SIMDified incarnation ARTful [37]. Moreover, considerable work has been done in the past years to better understand the performance of hash tables which are widely used in query processing [2, 45].

Learned Indexes. The core task of a learned index [36] is to provide an index on a densely packed, sorted array. The main idea is to manually define an (outer) B-tree-like structure, typically a two-level tree (coined RMI by the authors). Then, inside each node, rather than performing a binary search on the keys contained in that node — as done in a textbook B-tree — a learned regression function is used to predict the position in the sorted array. Care has to be taken to avoid prediction errors. This is done through an error correction method: the prediction actually defines a range which must be post-filtered through a different algorithm like binary or interpolation search. The biggest advantage of a 'learned index' is that no space is required to store pivots in internal nodes thus allowing for high branching factors. Like our work, the original work was a read-only index. It bulkloaded the index top-down, but as with any other B-tree like structure, bottom-up bulkloading up is also possible [32] and actually easier. Later on different proposals were made to use different regression techniques [31] and support inserts and deletes [14, 17]. Also note that the RMIs make a couple of other assumptions that may not always hold in practice [10]. As illustrated in Figure 2(b) already, an RMI is just one special configuration in GENE: an RMI is (1) a logical index: classical B-tree (however, fixed number of layers, balancing enforced, high fan-out), (2) a physical index: node internal search constrained to use some form of linear regression. In other words, an RMI handcrafts its logical structure. Then, inside its nodes it uses a fixed physical regression method to learn a CDF. In contrast, we allow for optimizing the structure *and* the search methods and data layouts used inside nodes. Thus, we fully embrace the orthogonality of *learning a model only inside* a node vs *optimizing the entire index structure*. Our approach aims at optimizing the entire index structure not only learning weights in a handcrafted structure.

Periodic Tables and Data Calculator. The work by Stratos Idreos et.al. on semi-automatic data structure design is truly inspiring. In their vision paper [27] they aim at a complete dissection and classification of the individual primitives used to design data structures. They sketch the huge design space of indexes and conclude that many quadrants in that space are still unexplored. They also phrase the high-level vision to synthesize an index from a declarative specification. Their main idea is to use a fine-grained learned cost models to be able to cost the physical individual index primitives (like scans, binary search, etc.). However, they go not further to show how this can be achieved concretely. In addition, no split into logical and physical indexes is given which is the key enabler in our approach. The follow-up work [28] is another vision paper which goes into somewhat more detail in describing the problem space of this endeavor and proposing a workbench like "'Data Alchemist' architecture" which is a semi-automatic design tool. However, again no experiments and/or results are shown. Then, [29] explores a large set of physical index design primitives, benchmarks them, and uses the results to learn cost models for physical primitives. This is used to build synthesized cost models for the expected cost of a combination of those physical primitives. The authors show several indexes where these cost estimates match the actual runtimes very well. At the same time the paper emphasizes that many physical design primitives and their cost models are missing including compression, concurrency, updates, etc. In their most recent work [25], they present the concept of design continuums, which unify different data structure designs by introducing common parameters, rules, and domains necessary to describe the underlying individuals. Using this design continuum, they show how to transition between known data structures, exposing also hybrid designs, and how to extend the continuum by new designs. Their focus lies on the semi-automated construction of these design continuums which are supposed to support researchers and engineers in finding a close to optimal data structure for a given problem composed of workload and hardware by using it as an inference engine.

There are four important differences to our work: we focus on (1) fully automatic index structure construction, (2) we provide a clear separation into logical and physical index components, (3) we believe that the index design space is simply too big for a practical system to be comprehensively modeled by (learned) cost models one reason being that costs models of different physical primitives are often non-additive and hence not usable for an optimization process. (4) Optimization time is important but not as critical as in standard query optimization: recall that the creation of an index structure is an offline process (in contrast to the creation of an index instance at query time!). And therefore, it makes a lot of sense to define fitness via actual observed runtime measurements rather than cost models whenever possible.

Generic Frameworks. A couple of generic indexing frameworks have been proposed in the past, most notably GIST [23] and XXL [11]. Those frameworks also aimed at generalizing presumably different index structures into a common software framework. This in turn allowed architects to implement important database algorithms for the generic index. The specialized indexes could then relatively easily be adapted to use the generic algorithms. Prominent examples include generic bulkloading [13] and concurrency control [33]. Though that work was inspiring to us, we stress that in our paper we argue on a conceptual level rather than an object-oriented-level. Moreover, we are primarily inspired by the analogue separation into logical and relational operators without immediately specifying how physical operators get implemented (ONC, vectorization, SIMD, whatever) or even how software interfaces need to be defined, as that is a tertiary concern.

DQO. Recently, we proposed Deep Query Optimization [15]. The core idea is to break operators into smaller components which can then possibly be optimized using traditional query optimization technique. This paper is another inspiration of our work. However, that work does not go into any detail on how such an idea can be realized in the context of indexing. It neither details how traditional operators can be split nor how this can be turned into an optimization problem for automatic index creation. We fill that gap.

Index Selection. Index Selection [35, 38] operates on a completely different level as our approach. Instead of coming up with a concrete index structure, in index selection the goal is to determine a suitable set of attributes to index in order to improve the runtime of a workload. In contrast, in our work we consider how to devise efficient index structures in the first place — which could then be leveraged in index selection algorithms.

Adaptive Indexing. As index selection is NP-hard, an interesting strategy is to not consider indexing a binary decision but rather allow indexes to become more and more fine-grained over time. That is at the heart of adaptive indexing [26]. Several interesting proposals have been made in this space, see [47] for a survey. However, all these indexes are still handcrafted indexes. In future work, we are planning to revisit some of these techniques, as the DT-field of our logical nodes can be used to mimic many of those techniques.

Genetic Algorithms. Genetic algorithms are a long known search method for an infeasible search space and have been used in our database community for decades. Early work by Bennett et al. [7] applied a genetic algorithm to search for efficient plans in a query optimizer. Other papers used similar approaches to improve database testing [5] or to perform index selection [20, 34, 40]. We are however not aware of papers tackling the problem of index creation using a genetic algorithm and therefore try to further extend the application area of these algorithms.

Decoupling Logical and Physical Indexes. Early work on partitioning schemes was done by Hellerstein et al. [22]. They represent data as a set of partitions where each partition is then (redundantly) mapped to at least one physical replica. In contrast to our work, they do not consider partitioning trees as in our logical indexes and they also do not further detail how to physically implement each partition. In the field of structural indexing [1, 19, 41] introduce the idea to co-partition (or cluster) tuples in a relational schema using graph partitioning. These graph partitions can then be exploited to answer structural queries which could be difficult to compute using foreign key indexes only. Their work has a completely different goal: while we strive to create a single physical index, they strive to create a graph partitioning which can then be mapped to suitable existing indexes. Extending our logical index partitions to their graph co-partitions could be an interesting future extension to GENE. The GMAP project by Tsatalos et al. [49, 50] is another interesting work in the area of physical data independence and index design. In contrast to their work, we focus on the clear difference between a logical and physical index and not the schema and a physical index. Moreover, we automatically generate efficient index structures, while their work only allows the choice of one concrete physical index.

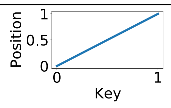
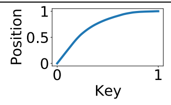
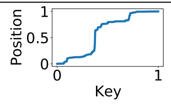
6 EXPERIMENTAL EVALUATION

In our experiments, we first determine a suitable set of hyperparameters for our genetic framework. Based on those hyperparameters, we then carefully evaluate GENE. We highlight the cost for training and the ability to automatically reach a certain performance baseline. Finally, we show the capability of GENE to match and even beat the performance of several state-of-the-art index structures.

System. All experiments were executed on a machine with an AMD Ryzen Threadripper 1900X 8-Core processor with 32 GiB memory on Linux. Our framework and the respective experiments are implemented in C++ and compiled with Clang 8.0.1, -O3. All experiments are run single-threaded and in main-memory.

Datasets. We use three types of datasets. All datasets consist of unique 64-bit uint keys and a 64-bit payload. In the following, we refer to the keys as *data.keys*. The payload represents the offset of the corresponding key into a sorted array. Therefore, we refer to the payload as *data.offset*. The datasets exhibit a variety of different characteristics like *distribution*, *density*, *domain*, and *size*. The first dataset uni_{dense} contains keys that are uniformly distributed in a dense domain. Concretely, uni_{dense} contains keys in the range [0, n) where n is the size of the dataset. The other two datasets, *books* and *osm*, represent real-world datasets with complex distributions and are taken from [31]. The datasets are sampled-down to our specific data size by uniformly drawing elements without duplicates. We have two main dataset sizes 100K and 100M, depending on the concrete experiment. Table 2 gives an overview of the datasets.

Table 2: Datasets used in the experiments.

Dataset	CDF	Properties
$\text{uni}_{\text{dense}}$		n := # elements (100K, 100M) 64-bit unique unsigned integers
books		n := # elements (100K, 1M, 10M, 100M) 64-bit unique unsigned integer Dataset taken from [31]
osm		n := # elements (100K, 100M) 64-bit unique unsigned integers Dataset taken from [31]

Workloads. We use three classes of workloads: point, range, and mixed point and range query workloads. For the moment, all our workloads are read-only, i.e. we do not consider *insert*, *delete*, or *update* statements. Note however, that our generic framework still supports insertions and deletions. In addition, *update* statements would not alter the structure of the index so we could easily integrate them into our framework. Table 3 summarizes the basic workload types. Point(data, idx_{min}, idx_{max}) represents a point query workload where the keys to lookup are taken from the keys in the dataset *data* by selecting indices in the subdomain $[\text{idx}_{\text{min}}, \text{idx}_{\text{max}}) \subseteq [0, n)$ with a uniform distribution. Likewise, Range_{sel}(data, idx_{min}, idx_{max}) describes a range query workload consisting of pairs specifying the lower bound and upper bound of the query. The lower bound is drawn with a uniform distribution in the index domain $[\text{idx}_{\text{min}}, \text{idx}_{\text{max}}$ - data.size * sel$) \subseteq [0, n)$ and the upper bound is set based on the dataset size and the given selectivity *sel*. If the domain is not explicitly specified, we assume it to cover the whole dataset. Mix(data, P, R) represents a mix of point and range queries with P and R being sets of point and range query workloads, respectively, based on *data*. Note, that in contrast to the datasets, our workloads may contain duplicates.

As already showcased in Sections 3 and 4, there is a huge search space in designing physical index structures. Consequently, in our experiments, we focus on the most important data layouts and search algorithms. We use the data layouts depicted in Table 4. As search algorithms, we use **scan**, **binS**, **intS**, **expS**, and **hashS** described in more detail in Section 3.1.

6.1 Hyperparameter Tuning

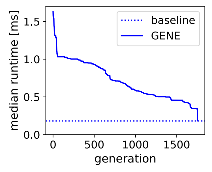
We use a uni_{dense} dataset of size 100K and vary five different parameters within this experiment: (1) number of mutations per

Table 3: Workloads used in the experiments.

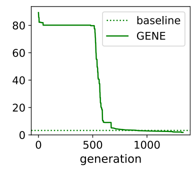
Workload	Characteristics	Parameters
Point(data, $\text{idx}_{\min}$, $\text{idx}_{\max}$)	point queries in index domain [$\text{idx}_{\min}$, $\text{idx}_{\max}$) with uniform distribution	[$\text{idx}_{\min}$, $\text{idx}_{\max}$) $\subseteq [0, n)$
Range_{sel}(data, $\text{idx}_{\min}$, $\text{idx}_{\max}$)	range queries in index domain [$\text{idx}_{\min}$, $\text{idx}_{\max}$) with uniform distribution and selectivity *sel*	[$\text{idx}_{\min}$, $\text{idx}_{\max}$) $\subseteq [0, n)$ sel $\in [0, 1]$
Mix(data, P, R)	mix of point and range query workloads with P and R being sets of respective workloads based on *data*	$P := \{p \mid p$ is Point(data, $\text{idx}_{\min}$, $\text{idx}_{\max}$)$\}$ $R := \{r \mid r$ is Range_{sel}(data, $\text{idx}_{\min}$, $\text{idx}_{\max}$)$\}$

Table 4: Data layouts used in the experiments.

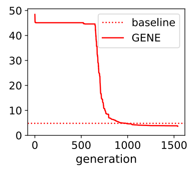
Data Layout	Characteristics	Implementation Detail
sorted_col	RI and DT have columnar layout for both keys and values. Sorted according to keys.	C++ standard library container `std::vector<Key>` and `std::vector<Value>`
hash	DT represents hash table mapping keys to their values. RI empty.	C++ standard library container `std::unordered_map<Key, Value>`
tree	RI and DT represent tree data structure mapping keys to their values. Sorted according to keys.	C++ standard library container `std::map<Key, Value>`



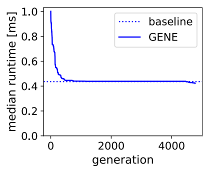
(a) **PQ, uni$_{\text{dense}}$**



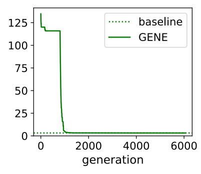
(b) **RQ, uni$_{\text{dense}}$**



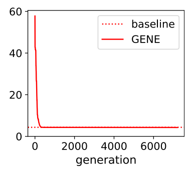
(c) **Mixed, uni$_{\text{dense}}$**



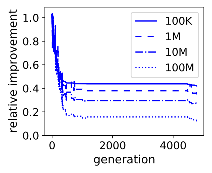
(d) **PQ, books**



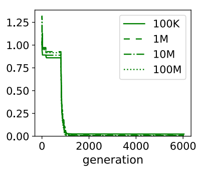
(e) **RQ, books**



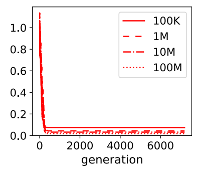
(f) **Mixed, books**



(g) **Upscaling, PQ, books**



(h) **Upscaling, RQ, books**



(i) **Upscaling, Mixed, books**

Figure 6: (a-f): GENE approaching handcrafted baselines on three different workloads: A point query only workload (PQ), a range query only workload (RQ) and a mixed workload consisting of 80% point and 20% range queries. (g-i): Relative improvement compared to the initial index structure after upscaling to dataset sizes of 100K to 100M.

generation ($s_{\max}$): $s_{\max} \in \{10, 50\}$, (2) maximum population size (s_{Π}): $s_{\Pi} \in \{50, 200, 1000\}$, (3) tournament selection size (s_T): $s_T \in \{10\%, 50\%, 100\%$ of population size$\}$, (4) initial population size (s_{init}): $s_{\text{init}} \in \{10, 50\}$, (5) population insertion criterion (q): Instead of taking the median of the subset drawn during tournament selection, we define a percentile q to be reached for a mutated individual to be inserted into the population: $q \in \{0\%, 50\%, 100\%\}$. For the 0% percentile, we always insert the mutated individual, for the 100% percentile we only add it if it is better than the previous best individual within the tournament selection subset.

Table 5: Best Genetic Search Configurations (over 5 runs)

Rank	$s_{\max}$	s_{Π}	s_T	s_{init}	q	median runtime [s]	mean runtime [s]
1	10	200	100%	50	0%	13.72	91.72
2	10	1000	50%	50	50%	14.58	26.10
3	10	1000	100%	10	50%	16.71	24.94
4	10	1000	100%	50	0%	16.87	94.48
5	10	1000	50%	10	50%	18.21	158.49

Table 5 shows the best configurations (based on the median of the 5 runs executed per configuration). Given a total number of mutations we want to perform, we conclude that it is more beneficial to use a smaller number of mutations per generation combined with a larger number of generations. As the population size has a limited influence, we decided to keep it very small to reduce the overhead to maintain the population. We therefore used the following default parameters for the experiments in the following sections: $s_{\max} = 10$, $s_{\Pi} = 50$, $s_T = 25$, $s_{\text{init}} = 10$ and $q = 50\%$.

6.2 Rediscover Suitable Baseline Indexes

In this experiment, we will demonstrate that our genetic algorithm is capable of reproducing the performance of various baseline index structures as known from textbooks. We consider two different datasets: *uni*$_{\text{dense}}$ and *books* of sizes 100K, 1M, 10M and 100M. We combine each of those two datasets with three different workloads of 10,000 queries each: Point(uni$_{\text{dense}}$), $\text{Range}_{0.001}$(uni$_{\text{dense}}$) and a Mix(uni$_{\text{dense}}$, P, R) workload, with P := {Point(uni$_{\text{dense}}$)} and R := {$\text{Range}_{0.01}$(uni$_{\text{dense}}$)} consisting of 80% point and 20% range queries. For each workload, we define a baseline within our generic framework of which we believe it has a decent performance: For the point query only workload, we assume a simple hash table to perform best which is implemented as an index structure with a single node having the hash data layout. For the range query only and mixed workload, we assume a B-tree-like structure to offer a decent performance. We initialize the tree to have 100 fully filled leaves, each containing 1,000 elements and a fan-out of 10 for the internal nodes. Each node is configured to use the `sorted_col` layout and `binS`. We configured GENE to allow nodes to contain up to 100,000 key-value-pairs or 100,000 child partitions (potentially leading to solutions consisting of a single node or solutions with one node per element assembled under a single root node). In the initial population trees were bulkloaded with 100 equally filled leaves and a fan-out of 10, but with randomized data layouts and search methods. Each experiment is conducted for 8000 generations. The genetic search was run on the smallest sample size of 100K elements. Each time we found a new, best individual, we checked if the results carry over to the larger datasets, i.e. we created new index structures using the same routing information and data layouts and search methods as found by GENE (i.e. using the exact same index structure), but bulkloaded them with the larger dataset, increasing leaf capacities if necessary. We then evaluated them using the exact same workload as used in the genetic search.

Figure 6 shows the results. Each plot in the first two lines compares the performance of the baseline to the performance of the genetic algorithm where we plotted the best individual of each generation. We plot the curves up to the point of the last improvement.

As we can clearly see, GENE rapidly approaches the baseline. This is mostly due to the fact that GENE can rather easily improve by mutating very inefficient nodes in the beginning. After getting close to the baseline, GENE only finds slight improvements, e.g. by changing search algorithms within nodes, which are hardly visible on the plot. The index structures found by GENE are very similar to the baselines: On the uni_{dense} dataset, GENE always returned a single node index structure. For the point query only workload, it came up with a single `hash` node containing all entries, i.e. exactly the baseline we defined beforehand. For the range query only as well as mixed workload, GENE also reduced the index to a single node, but with `sorted_col` data layout and `intS` search method. This difference is due to the fact that range queries can not be executed efficiently on a hash node. This result is reasonable as a uniformly distributed, dense dataset can easily be modeled by an array with a linear model as search method. Considering the *books* dataset, the point query workload resulted in a tree with 68 nodes in total, 66 of them being leaves. All but one leaf are direct children of the `sorted_col` root node, the remaining leaf has a single `tree` node between itself and the root. With 48 nodes, the vast majority of the leaves has a `hash` data layout. The remaining leaves are of `sorted_col` (15) or `tree` data layout (3). The dominating search method for non-hash nodes is `binS`, with only 3 exceptions that use `expS`. The resulting index structure reminds of a partitioned hash map, indicating that GENE indeed approached the expected baseline. For the range query workload, we obtained an index with similar size, having 44 nodes with `sorted_col` data layout in total, 40 of them being leaves. The index has a height of three with the majority of the leaves (38) situated at depth two and only two leaves being one level below. `BinS` is again the dominating search method for the leaves, with four nodes using `intS` and two using `expS` instead. The resulting index structure reminds of a shallow B-tree, indicating that GENE again approached the expected baseline. For the final mixed workload, the results are similar to the range only workload. We obtained an index of height three with 41 nodes in total (all with `sorted_col` data layout), 35 of them being leaves. The majority of the leaves is at depth two, with three leaves being one level above and one leaf being a level below. The dominating search method is again `binS`, with only 7 leaves using an `intS` instead. As for the range query only workload, GENE approached a shallow B-tree like index to match the performance of the baseline. The last line in Figure 6 shows the improvements of the scaled index structures for the books dataset. Each line represents the relative improvements compared to the best index structure of GENE's initial population, upscaled to the indicated dataset sizes of 100K (the size on which the search was conducted) up to 100M. We can clearly see that an improvement in the solid line representing the training data nearly always results in a very similar improvement for the upscaled index structures. The overall, relative improvement becomes even bigger with increasing dataset size, indicating that is most likely sufficient to run GENE on a sample of the data to obtain a decent index structure, highly reducing the necessary search time. If best possible performance is the ultimate goal, then GENE can again be applied to the upscaled index structure resulting from the sample to perform further fine tuning.

We also experimented with an additional, mixed workload again consisting of 10,000 queries with a 80% / 20% point to range query ratio, based on the uni_{dense} dataset. However, this time we chose the queries to be normally distributed around key 75,000 with a standard deviation of 10,000, i.e. the queries were mainly focused on the upper half of the key domain. Our GENE algorithm again decided to shrink the initial index structures considerably, however it stopped after 3500 generations returning a tree with 4 levels and 25 nodes in total, 17 of them being leaves. The nodes containing the upper half of the key domain were again using the `sorted_col` layout and either `intS` or `binS`. The total runtimes of GENE heavily depend on the concrete datasets and workloads. The fastest execution for uni_{dense} with point query only workload took less than 3 minutes until the last improvement was found. The longest run on the same dataset with range query only workload took about 122 minutes. Performing the additional upscaling steps further influenced the runtimes, leading to execution times of up to 30 hours for the *books* dataset in combination with range query only workload.

6.3 Optimized vs Heuristic Indexes

In this section, we will compare the performance of a GENE index with representatives of different prevalent heuristic index types. Table 6 gives an overview of the different index types and respective representatives. For the B+tree implementation we use the commonly used TLX baseline implementation by Bingmann [8]. In particular, we use the specialized B+tree template class `btree_map` implementing STL's map container. The ART implementation is taken from the SOSD benchmark [39] by Marcus et al. and concretely, we use the implementation `ARTPrimaryLB` that supports lower bound lookups. PGM [18] by Ferragina et al. provides multiple implementations that support a variety of different functionalities like insertion and deletion support or compression to reduce space usage. Since we are only interested in the lookup performance, we use the default `PGMIndex` implementation. We purposely exclude hash tables since they do not support range queries efficiently.

Table 6: Overview of different index types and representatives of each category.

Type	Index	Details
Tree	B-tree	TLX `btree_map` [8]
Radix	ART	SOSD `ARTPrimaryLB` [39]
Learned	PGM	PGM `PGMIndex` [18]

We conduct our performance evaluation on the three different datasets, uni_{dense}, *books*, and *osm*, each with a size of $n = 100$M data points. As for the workload, we are going to use a mixed workload consisting of multiple point and range query workloads. Concretely, the workload consists of 1M queries, divided in three point query workloads and one range query workload: Mix(data, P, R), with $P :=$ {Point(data, 0, 0.1 · n), Point(data, 0.1 · n, 0.85 · n), Point(data, 0.85 · n, n)} and $R :=$ {Range(data, 0.1 · n, 0.85 · n)}, where data ∈ {$\text{uni}_{\text{dense}}$, books, osm}. With that, the queried key domain is essentially split into three partitions at 10% and 85% of the data based on the different workloads. The first partition $[0, 0.1 \cdot n)$ exclusively receives point queries representing 20% of the total workload size. The second partition $[0.1 \cdot n, 0.85 \cdot n)$ receives a mix of both, 10% point and 20% range queries, and the third partition $[0.85 \cdot n, n)$ 50% point queries. Figure 7 illustrates the workload based on the *osm* dataset. Since each data point maps a key to its position in a sorted data array,

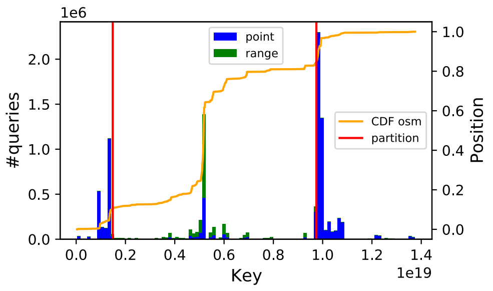


Figure 7: Visualization of the experimental setup. The *osm* dataset is shown as CDF while the point and range queries are illustrated as a stacked histogram. The red vertical lines highlight the partition borders.

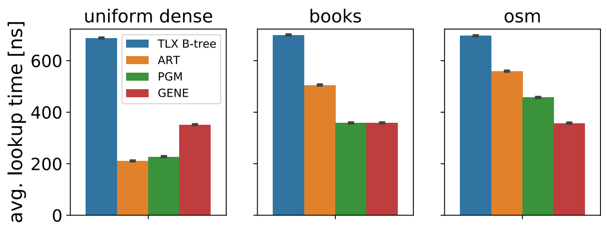


Figure 8: Average index lookup time comparison between three representative state-of-the-art index structures and our GENE index on three different datasets and workloads described in subsection 6.3. The small black bars indicate the standard deviation of five runs, which is negligibly small.

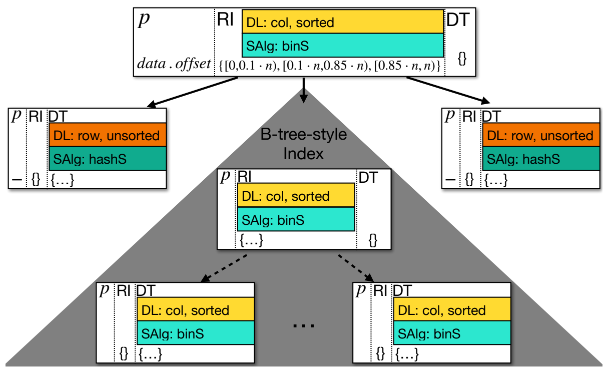


Figure 9: Physical index structure of the GENE index based on the workload partitioning.

range queries can be translated to finding the position of the lower bound in the index and subsequently scanning the data array. This scan is independent of the underlying index type and can therefore be neglected. Thus, a range query in our evaluation is equivalent to a lower bound lookup in the index. Our generic implementation allows us to easily replace specific parts of a physical index structure like the data layout or search method. However, this leads to a non-negligible performance overhead mainly due to repeated dynamic dispatches. To be competitive with the other baselines and state-of-the-art index structures, we provide an additional implementation that specifically contains the concrete physical index structures used in this experiment. Figure 9 shows the physical structure of our GENE index. Since the workload domain is split into three partitions with two exclusive point query regions, we bulkload our index structure accordingly. The first and third partition are hash nodes while the second partition represents a B-tree-style index. The root is a sorted array using binary search. We randomly shuffle the workload before each execution to avoid caching effects.

Figure 8 shows the results of the index structures for different datasets. We report the average index lookup time. Independent of the underlying dataset, the TLX B-tree requires around 700 ns and is not able to compete with the other indexes. On the uniform dense dataset, ART and PGM both achieve a lower lookup time than GENE. However, for both, a uniform dense dataset is close to the optimal use case. For the two real-world skewed and sparse datasets, our GENE index achieves a competitive or even faster lookup time than the other index structures of around 350 ns.

We are well aware that this is a very specific use case, however, it showcases that there are indeed scenarios where an optimized GENE index can outperform a state-of-the-art (heuristic) data structure. Expanding the covered design space by GENE, i.e. the available data structures and search algorithms, and automatically finding those scenarios is part of future work. In conclusion, our proof of concept emphasizes that there are use cases in which GENE is able to achieve a competitive or even superior performance than state-of-the-art index structures and therefore, confirms its validity.

7 CONCLUSION AND FUTURE WORK

Conclusions. This paper has opened the book for automatically generated index structures. We have proposed a powerful generic indexing framework on the logical and physical level analogue to logical and physical operators in query processing and optimization. We have shown that by clearly separating the logical and physical dimensions of an index, a huge number of existing (physical) indexes can be represented in our generic indexing framework. Furthermore, we introduced *Genetic Generic Generation of Indexes (GENE)*. Given a workload, GENE can come up with an efficient physical index structure automatically. Our initial experimental results outlines the potential and efficiency of our approach.

Future Work. This paper is obviously just a starting point and there are many possible exciting research directions ahead:

(1) code-generation, similar to generating code for the most efficient physical *plan* found, generate code for the most efficient *physical index structure* found,
(2) *The Index Farm*: we plan to open source our framework: the goal is that people submit a workload on a web page and the framework emits suitable source code for an index structure,
(3) runtime adaptivity: how to mutate structurally, this can also simulate the adaptive indexing family of index structures,
(4) updates: explore workloads with inserts, updates, and deletes,
(5) scalability: extend our scalability experiments to evaluate workloads only on subtrees affected by mutations using cost functions to prioritize expensive partitions when drawing nodes for mutations
(6) effects of non-empty DT-fields in internal nodes,
(7) extend GENE to support more data layouts, search algorithms, and hardware acceleration (SIMD).

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] Erik Agterdenbos, George H. L. Fletcher, Chee-Yong Chan, and Stijn Vansummeren. 2016. Empirical evaluation of guarded structural indexing. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT*. 714–715. https://doi.org/10.5441/002/edbt.2016.101

[2] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. 2015. A comparison of adaptive radix trees and hash tables. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 1227–1238. https://doi.org/10.1109/ICDE.2015.7113370

[3] Lars Arge. 1995. The Buffer Tree: A New Technique for Optimal I/O-Algorithms (Extended Abstract). In *Algorithms and Data Structures, 4th International Workshop, WADS '95, Kingston, Ontario, Canada, August 16-18, 1995, Proceedings (Lecture Notes in Computer Science)*, Vol. 955. Springer, 334–345. https://doi.org/10.1007/3-540-60220-8_74

[4] D. Baskins. 2004, (accessed November 8, 2021). *Judy arrays*. http://judy.sourceforge.net/

[5] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. 1243–1251. http://www.vldb.org/conf/2007/papers/industrial/p1243-bati.pdf

[6] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Informatica* 1 (1972), 173–189. https://doi.org/10.1007/BF00288683

[7] Kristin P. Bennett, Michael C. Ferris, and Yannis E. Ioannidis. 1991. A Genetic Algorithm for Database Query Optimization. In *Proceedings of the 4th International Conference on Genetic Algorithms*. 400–407.

[8] Timo Bingmann. 2018. TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers. https://github.com/tlx/tlx, accessed November 8, 2021.

[9] Miles Cranmer, Alvaro Sanchez-Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. 2020. Discovering Symbolic Models from Deep Learning with Inductive Biases. arXiv:2006.11287 [cs.LG]

[10] Andrew Crotty. 2021. Hist-Tree: Those Who Ignore It Are Doomed to Learn. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper20.pdf

[11] Jochen Van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider, and Bernhard Seeger. 2001. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 39–48. http://www.vldb.org/conf/2001/P039.pdf

[12] Jochen Van den Bercken and Bernhard Seeger. 2001. An Evaluation of Generic Bulk Loading Techniques. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 461–470. http://www.vldb.org/conf/2001/P461.pdf

[13] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. 1997. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 406–415. http://www.vldb.org/conf/1997/P406.PDF

[14] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 969–984. https://doi.org/10.1145/3318464.3389711

[15] Jens Dittrich and Joris Nix. 2020. The Case for Deep Query Optimisation. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p3-dittrich-cidr20.pdf

[16] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.* 4, 3 (1979), 315–344. https://doi.org/10.1145/320083.320092

[17] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175. http://www.vldb.org/pvldb/vol13/p1162-ferragina.pdf

[18] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[19] George H. L. Fletcher, Dirk Van Gucht, Yuqing Wu, Marc Gyssens, Sofia Brenes, and Jan Paredaens. 2009. A methodology for coupling fragments of XPath with structural indexes for XML documents. *Inf. Syst.* 34, 7 (2009), 657–670. https://doi.org/10.1016/j.is.2008.09.003

[20] Farshad Fotouhi and Carlos E. Galarce. 1989. Genetic Algorithms and the Search for Optimal Database Index Selection. In *Computing in the 90's, The First Great Lakes Computer Science Conference (Lecture Notes in Computer Science)*, Vol. 507. 249–255. https://doi.org/10.1007/BFb0038500

[21] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2002. *Database Systems - the Complete Book (International Edition)*. Pearson Education.

[22] Joseph M. Hellerstein, Elias Koutsoupias, Daniel P. Miranker, Christos H. Papadimitriou, and Vasilis Samoladas. 2002. On a model of indexability and its bounds for range queries. *J. ACM* 49, 1 (2002), 35–55. https://doi.org/10.1145/505241.505244

[23] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*. Morgan Kaufmann, 562–573. http://www.vldb.org/conf/1995/P562.PDF

[24] John Henry Holland. 1975. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

[25] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf

[26] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[27] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.* 41, 3 (2018), 64–75. http://sites.computer.org/debull/A18sept/p64.pdf

[28] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike S. Kester, Niv Dayan, Demi Guo, Minseo Kang, and Yiyou Sun. 2019. Learning Data Structure Alchemy. *IEEE Data Eng. Bull.* 42, 2 (2019), 47–58. http://sites.computer.org/debull/A19june/p47.pdf

[29] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 535–550. https://doi.org/10.1145/3183713.3199671

[30] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 339–350. https://doi.org/10.1145/1807167.1807206

[31] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *CoRR* abs/1911.13014 (2019). arXiv:1911.13014 http://arxiv.org/abs/1911.13014

[32] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*. ACM, 5:1–5:5. https://doi.org/10.1145/3401071.3401659

[33] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. 1997. Concurrency and Recovery in Generalized Search Trees. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. ACM Press, 62–72. https://doi.org/10.1145/253260.253272

[34] Marcin Korytkowski, Marcin Gabryel, Robert Nowicki, and Rafal Scherer. 2004. Genetic Algorithm for Database Indexing. In *Artificial Intelligence and Soft Computing - ICAISC (Lecture Notes in Computer Science)*, Vol. 3070. 1142–1147. https://doi.org/10.1007/978-3-540-24844-6_179

[35] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395. http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf

[36] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 489–504. https://doi.org/10.1145/3183713.3196909

[37] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[38] Vincent Y. Lum and Huei Ling. 1971. An Optimization Problem on the Selection of Secondary Keys. In *Proceedings of the 1971 26th Annual Conference (ACM '71)*. Association for Computing Machinery, New York, NY, USA, 349–356. https://doi.org/10.1145/800184.810505

[39] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[40] Priscilla Neuhaus, Julia Couto, Jonatas Wehrmann, Duncan Dubugras Alcoba Ruiz, and Felipe Meneguzzi. 2019. GADIS: A Genetic Algorithm for Database Index Selection (S). In *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE.* 39–54. https://doi.org/10.18293/SEKE2019-135

[41] François Picalausa, George H. L. Fletcher, Jan Hidders, and Stijn Vansummeren. 2014. Principles of Guarded Structural Indexing. In *Proc. 17th International Conference on Database Theory (ICDT).* 245–256. https://doi.org/10.5441/002/icdt.2014.26

[42] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK.* Morgan Kaufmann, 78–89. http://www.vldb.org/conf/1999/P7.pdf

[43] Jun Rao and Kenneth A. Ross. 2000. Making B$^+$-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.* ACM, 475–486. https://doi.org/10.1145/342009.335449

[44] Esteban Real, Chen Liang, David So, and Quoc Le. 2020. Automl-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning.* PMLR, 8007–8019.

[45] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *Proc. VLDB Endow.* 9, 3 (2015), 96–107. https://doi.org/10.14778/2850583.2850585

[46] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2009. k-ary search on modern processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009.* ACM, 52–60. https://doi.org/10.1145/1565694.1565705

[47] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *Proc. VLDB Endow.* 7, 2 (2013), 97–108. https://doi.org/10.14778/2732228.2732229

[48] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1.* ACM, 23–34. https://doi.org/10.1145/582095.582099

[49] Odysseas G. Tsatalos and Yannis E. Ioannidis. 1994. A Unified Framework for Indexing in Database Systems. In *Database and Expert Systems Applications, 5th International Conference, DEXA (Lecture Notes in Computer Science)*, Vol. 856. 183–192. https://doi.org/10.1007/3-540-58435-8_183

[50] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. 1996. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB J.* 5, 2 (1996), 101–118. https://doi.org/10.1007/s007780050018