BF-Tree: Approximate Tree Indexing

Manos Athanassoulis
École Polytechnique Fédérale de Lausanne
Lausanne, VD, Switzerland
manos.athanassoulis@epfl.ch

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne
Lausanne, VD, Switzerland
natassa@epfl.ch

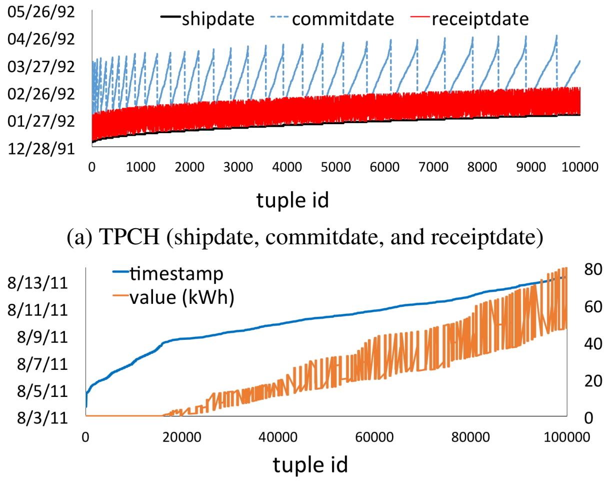
ABSTRACT

The increasing volume of time-based generated data and the shift in storage technologies suggest that we might need to reconsider indexing. Several workloads - like social and service monitoring - often include attributes with implicit clustering because of their time-dependent nature. In addition, solid state disks (SSD) (using flash or other low-level technologies) emerge as viable competitors of hard disk drives (HDD). Capacity and access times of storage devices create a trade-off between SSD and HDD. Slow random accesses in HDD have been replaced by efficient random accesses in SSD, but their available capacity is one or more orders of magnitude more expensive than the one of HDD. Indexing, however, is designed assuming HDD as secondary storage, thus minimizing random accesses at the expense of capacity. Indexing data using SSD as secondary storage requires treating capacity as a scarce resource.

To this end, we introduce approximate tree indexing, which employs probabilistic data structures (Bloom filters) to trade accuracy for size and produce smaller, yet powerful, tree indexes, which we name Bloom filter trees (BF-Trees). BF-Trees exploit pre-existing data ordering or partitioning to offer competitive search performance. We demonstrate, both by an analytical study and by experimental results, that by using workload knowledge and reducing indexing accuracy up to some extent, we can save substantially on capacity when indexing on ordered or partitioned attributes. In particular, in experiments with a synthetic workload, approximate indexing offers 2.22x-48x smaller index footprint with competitive response times, and in experiments with TPCH and a monitoring real-life dataset from an energy company, it offers 1.6x-4x smaller index footprint with competitive search times as well.

1. INTRODUCTION

Database Management Systems (DBMS) have been traditionally designed with the assumption that the underlying storage is comprised of hard disks (HDD). This assumption impacts most of the design choices of DBMS and in particular the ones of the storage and the indexing subsystems. Data stored on the secondary storage of a DBMS can be accessed either by a sequential scan or by using an index for randomly located searches. The most common types of indexes in modern database systems are B$^+$-Trees and hash indexes [37]. Other types of indexing include bitmap indexes [33].



(a) TPCH (shipdate, commitdate, and receiptdate)

(b) Smart home dataset (timestamp, aggregate energy)

Figure 1: Implicit clustering

Tree structures like B$^+$-Trees are widely used because they are optimized for the common storage technology - HDD - and they offer efficient indexing and accessing for both sorted and unsorted data (e.g., in heap files). Tree structures offer logarithmic, to the size of the data, number of random accesses (and, as a result, lookup time) and support ordered range scans. Hash tables are very efficient for point queries, i.e., for a single value probe, because, once hashing is completed the search cost is constant. Indexing in today's systems is particularly important because it needs only a few random accesses to locate any value, which is sublinear to the size of the data (e.g., logarithmic for trees, constant for hash indexes).

1.1 Implicit Clustering

Big data analysis - often performed in a real-time manner - is becoming increasingly more popular and crucial to business operation. New datasets including data from scientists (e.g., simulations, large-scale experiments and measurements, sensor data), social data (e.g., social status updates, tweets) and, monitoring, archival and historical data (managed by data warehousing systems), all have a time dimension, leading to implicit, time-based clusters of data, often resulting in storing data based on the creation timestamp. The property of *implicit clustering* [30] characterizes data warehousing datasets, which are often naturally partitioned for the attributes that are correlated with time. For example, in a typical data warehousing benchmark (TPCH [44]) for every purchase we have three dates (ship date, commit date and receipt date).

Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 14

While these three dates do not have the same order for different products, the variations are small and the three values are typically close. Figure 1(a) shows the dates of the first 10000 tuples of the lineitem table of the TPCH benchmark when data is ordered using the creation time. A second example is smart home dataset (SHD) taken from electricity monitoring data[1] which keeps timestamped information about the current energy consumption, the aggregate energy consumption, and other sensor measurements like temperature. Figure 1(b) shows the first 100000 entries containing the timestamp and the aggregate energy of several clients. The timestamps are in increasing order and the aggregate consumed energy has also implicit clustering[2].

Today, real-time applications like monitoring sensors [23] and Facebook [9] have a constant ingest of timestamped data. In addition to the real-time nature of such applications, immutable files with historical time-generated data are stored and the goal is to offer cheap yet efficient storage and searching. For example, Facebook has announced projects to offer cold storage [43] using flash or shingled disks [22]. When cold data are stored on low-end flash chips as immutable files, they can be ordered or partitioned anticipating future access patterns, offering explicit clustering.

Datasets with either implicit or explicit clustering are *ordered* or *partitioned*, typically, on a time dimension. In this paper, we design an index that is able to exploit such data organization to offer competitive search performance with smaller index size.

1.2 The capacity-performance trade-off

Solid-state disks (SSD) use technologies like flash and Phase Change Memory (PCM) [16] that do not suffer from mechanical limitations like *rotational delay* and *seek time*. They have virtually the same random and sequential read throughput and several orders of magnitudes smaller read latency when compared to hard disks [10, 41]. The capacity of SSD, however, is a scarce resource compared with the capacity of HDD. Typically, SSD capacity is one order of magnitude more expensive than HDD capacity. The contrast between capacity and performance creates a *storage trade-off*. Figure 2 shows how several SSD and HDD devices (as of end 2013) are characterized in the trade-off according to their capacity (GB per $) on the x-axis and the advertised random read performance (IOPS) on the y-axis. We show two enterprise-level and two consumer-level HDD (E- and C-HDD respectively) and, four enterprise-level and two consumer-level SSD (E- and C-SSD respectively). The two technologies create two distinct clusters. HDD are on the lower right part of the figure offering cheap capacity (and in all cases cheaper than SSD) and inferior performance - in terms of random read I/O per second (IOPS) - varying from one to four orders of magnitude. Hence, instead of designing indexes for storage with cheap capacity and expensive random accesses (for HDD), today we need to design indexes for storage with fast random accesses with expensive capacity (for SSD).

1.3 Indexing for modern storage

Systems and applications requirements are heavily impacted by the emergence of SSD and there has been a plethora of research aiming at integrating and exploiting such devices with existing DBMS. These efforts include flash-only DBMS [41, 45], flash-HDD hybrid DBMS [25], using flash in a specialized way [5, 13] and optimizing internal structures of the DBMS for flash (e.g., flash-aware B-Trees and write-ahead-logging [1, 13, 18, 26, 38]). Additionally, the trends of increasing capacity and performance of SSD lead to higher adoption of hybrid or flash-only storage sub-systems [9]. Thus, more and more data resides on SSD and we need to access them in an efficient way.

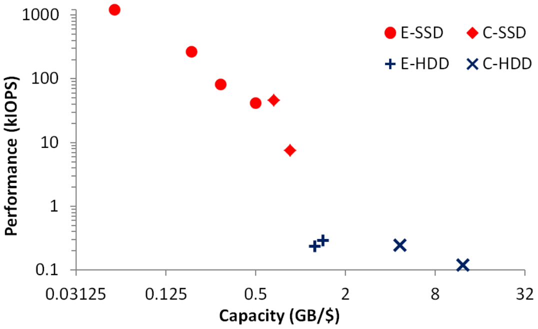


Figure 2: The capacity/performance storage trade-off.

SSD-aware indexes are not enough. Prior approaches for flash-aware B$^+$-Tree, however, focus on addressing the slow writes on flash and the limited device lifetime (see Section 2). These techniques do not address the aforementioned *storage trade-off* - between storage capacity and performance - since they follow the same principles that B$^+$-Trees are built with: minimize the number of slow random accesses by having a wide (and potentially large) tree structure. Instead of decreasing the index size at the expense of more random reads, traditional tree indexing uses larger size in order to reduce random reads.

1.4 Approximate Tree Indexing

We propose a novel form of sparse indexing, using an approximate indexing technique which leverages efficient random reads to offer performance competitive with traditional tree indexes, reducing drastically the index size. The smaller size enables fast rebuilds if needed. We achieve this by indexing in a lossy manner and exploiting natural partitioning of the data in a Bloom filter tree (BF-Tree). In the context of BF-Trees, Bloom filters are used to store the information whether a key exists in a specific range of pages. BF-Trees can be used to index attributes that are ordered or naturally partitioned within the data file. Similarly to a B$^+$-Tree, a BF-Tree has two types of nodes: internal and leaf nodes. The internal nodes resemble the ones of a B$^+$-Tree but the leaf nodes are radically different. A leaf node of a BF-tree (BF-leaf) consists of one or more Bloom filters which store the information whether a key for the indexed attribute exists in a particular range of data pages. The choice of Bloom filters as the building block of a BF-Tree allows accuracy parametrization based on (i) the indexing granularity (data pages per Bloom filter) and (ii) the indexing accuracy (false positive probability of the Bloom filters). The former is useful when the data is not strictly ordered and the latter can be used to vary the overall size of the tree.

Contributions. This paper makes the following contributions:

- We identify the *capacity-performance storage trade-off*.
- We introduce approximate tree indexing using probabilistic data structures, which can be parametrized to favor either accuracy or capacity. We present such an index, the BF-Tree, which is designed for workloads with implicit clustering tailored for emerging storage technologies.
- We model the behavior of BF-Trees and present an analytical study of their performance, comparing them with B$^+$-Trees.
- We show in our experimental analysis that BF-Trees offer competitive performance with 2.22x to 48x smaller index size when compared with B$^+$-Trees and hash indexes.

[1] The dataset was made available through the EU project BigFoot: http://www.bigfootproject.eu.

[2] The aggregate energy of every client is increasing throughout every the billing cycle, but not always with the same pace.

Paper Organization. In Section 2 we discuss background about SSD-aware indexing and Bloom filters. In Section 3 we present a key insight which makes BF-Trees viable, in Section 4 we present the internals of a BF-Tree. Section 5 presents an analytical model predicting BF-Trees behavior and Section 6 presents the evaluation of BF-Trees. In Section 7 we further discuss BF-tree as a general index, and in Section 8 we discuss possible optimizations. In Section 9 we discuss related work, and in Section 10 we conclude.

2. BACKGROUND

SSD-aware indexing. We find that approximate indexing is suitable for modern storage devices (e.g., flash or PCM-based) because of their principal difference compared to traditional disks: random accesses perform virtually the same as sequential accesses[3]. Since the rise of flash as an important competitor of disks for non-volatile storage [21] there have been several efforts in creating a flash-friendly indexing structure (often a flash-aware version of a B$^+$-Tree). LA-Tree [1] uses lazy updates, adaptive buffering and memory optimizations to minimize the overhead of updating flash. FD-Tree [26] addresses the performance asymmetry between random read and writes in SSD using the logarithmic method and fractional cascading techniques. In μ-Tree [24] the nodes along the path from the root to the leaf are stored in a single flash memory page in order to minimize the number of flash write operations during the update of a leaf node. IPLB$^+$-tree [32] avoids costly erase operations - often caused by small random write requests common in database applications - in order to improve the overall write performance. SILT [27] is a flash-based memory-efficient key-value store based on cuckoo hashing and tries, which offers fast search performance using minimal amount of main memory.

What SSD-aware indexing does not do. Related work focuses, mostly, on optimizing for specific flash characteristics (read/write asymmetry, lifetime) maintaining the same high-level indexing structure and does not address the shifting trade-off in terms of capacity. BF-trees propose, orthogonally to flash optimizations, trading off capacity for indexing accuracy.

Bloom Filters' applications in data management. A Bloom filter (BF) [8] is a space-efficient probabilistic data structure supporting membership tests with non-zero probability for false positives (and zero probability for false negatives). Typically in a BF one can only add new elements and never remove elements. A deletable BF has been discussed [39] but it is not generally adopted. BFs have been extensively used as auxiliary data structures in database systems [3, 12, 14, 15, 31]. Modern database systems utilize BFs while implementing several algorithms, like semijoins [31], where BFs help in implementing faster the join algorithm. Google's Bigtable [12] uses BFs to reduce the number of accesses to internal storage components and a study [3] shows that Oracle systems use BFs for tuple pruning, to reduce data communication between slave processes in parallel joins and to support result caches.

BFs have been used for changing workloads and different storage technologies. Scalable Bloom Filters [2] study how a BF can adapt dynamically to the numbers of elements stored while assuring maximum false positive probability. Buffered Bloom filters [11] use flash, as well, as a cheaper alternative to main memory. The forest-structured Bloom filter [29] aims at designing an efficient flash-based BF by using the memory to capture the frequent updates. Bender et al. [7] propose three variations of BFs: the quotient filter, the buffered quotient filter and the cascade filter. The first uses more space than traditional BF but shows better insert/lookup performance and supports deletes. The last two variations are designed on top of quotient filter supporting larger workloads, serving, as well, as alternatives of BF for SSD.

We extend the usage of BFs in DBMS by proposing, BF-Tree, an indexing tree structure which uses BFs to trade off capacity for indexing accuracy by applying approximate indexing. BF-Tree is orthogonal to the optimizations described above. In fact, a BF-Tree can take advantage of such optimizations in order to fine-tune its performance.

3. SPLITTING BLOOM FILTERS

Bloom proposed [8] BF as a space-efficient probabilistic data structure which supports membership tests, with a false positive probability. On the other hand, the small size of a BF allows for its re-computation when needed. Thus, in BF-Trees we do not employ a BF for the entire relation. We use BF to perform a membership test for a specific range, keeping the range of values for a single BF small in order to be feasible to recompute it.

A BF is comprised of m bits, and it stores membership information for n elements with false positive probability p. An empty BF is an array of m bits all set to 0. We need, as well, k different hash functions to be used to map an element to k different bits during the process of adding an element or checking for membership. When an element is added, the k hash functions are used in order to compute which k out of m bits have to be set to 1. If a bit is already 1 it maintains this value. To test an element for membership the same k bits are read and, if any of the k bits is equal to 0, then the element is not in the set. If all k bits are equal to 1 then the element belongs to the set with probability $1-p$. Assuming optimal number of k hash functions the connection between the BF parameters is approximated by the formula[4] [42]:

$$n = -\frac{m \cdot ln^2(2)}{ln(p)} \tag{1}$$

From this formula we can derive two useful properties:

1. If a BF with size M bits can store the membership information of N elements with false positive p, then S BFs with size $\frac{M}{S}$ bits each can store the membership information of $\frac{N}{S}$ elements each with the same p.
2. Decreasing the probability of false positives has a logarithmic effect on the number of elements we can index using a given space budget (i.e., number of bits).

Property (1) allows to divide the index into smaller BFs that incorporate location information. This process is done hierarchically and is presented in Section 4. We present a tree structure with a large BF per leaf, which contains membership information, and internal nodes which help navigating to the desired range before we do the membership test. Each leaf node corresponds to a number of data pages and upon positive membership test we have to search these data pages for the desired values.

4. BLOOM FILTER TREE (BF-TREE)

In this section we describe in detail the structure of a BF-tree, highlighting its differences from a typical B$^+$-Tree.

4.1 BF-Tree architecture

A BF-tree consists of nodes of two different types. The root and the internal nodes have the same morphology as a typical B$^+$-Tree

[3] Early flash devices had one order of magnitude faster random reads than random writes [10, 41]; later devices are more balanced.

[4] Hereafter, when modeling the behavior of a BF we use Equation 1.

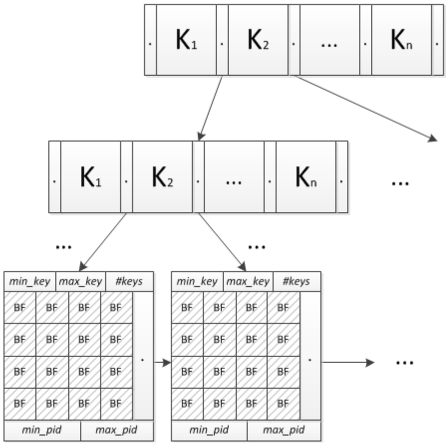


Figure 3: BF-tree

node: they contain a list of keys with pointers to other nodes between each pair of keys. If the referenced node is internal then it has exactly the same structure. The leaf nodes (BF-leaves), however, are different. They contain membership information of indexed keys in ranges of pages.

BF-leaf. Each leaf node corresponds to a page range (*min_pid* - *max_pid*) and to a key range (*min_key* - *max_key*), and it consists of a number, S, of *BFs*, each of which stores the key membership for each page of the range (or a group of consecutive pages). A pointer to the next leaf is created during bulk loading and is maintained throughout the lifetime of the index to facilitate range scans. Finally, each leaf contains the number of indexes keys (*#keys*) in order to guarantee the desired false positive probability. The connection between the page range and the key range does not imply sorted data, which is only one way to sustain it. In addition, if the dataset is partitioned using the index key the same connection is still valid. In cases of composite indexing keys, or for attributes that have values correlated with the order of the data, such an assumption can hold for more than one index.

For simplicity and compatibility with the existing framework, the root, the internal nodes and the leaf nodes have the same size (typically either 4KB or 8KB). The number of BFs in a BF-leaf can vary between 1 (where a single BF stores the membership information of the entire range) up to the number of pages comprising the range in question, which gives the best results because an index probe will be directed only to the pages containing the key in question. As shown in Section 3, using property (1), we can guarantee that creating a BF per page of the range will not alter the false positive probability, because dividing a BF with *#keys* elements into S BFs for $\frac{\#keys}{S}$ elements each, results in the same false positive probability. This property guarantees stable false positive probability for every BF, as long as the distribution of keys is not highly skewed. The search time of a BF-tree depends on three parameters: (i) the height of the tree, (ii) the *false positive probability* (fpp), and, (iii) the number of data pages that each BF corresponds.

4.2 BF-Tree algorithms

Searching for a tuple. As shown in Algorithm 1, once we have retrieved the desired BF-leaf, we perform a membership test for every BF. This test decides whether the key we search for exists in each BF, with probability for false positive answer fpp. The average search cost includes the overhead of false positives, which is negligible when fpp is low as we show in Sections 5 and 6. During a BF-Tree index probe, the system reads the BF-leaf which corresponds to the search key and then probes all BFs (one for each data page). When a BF matches the searched key then its corresponding page contains the key with false positive probability fpp. The page id is calculating by adding the the index within the leaf of the matching BF to the min_pid. All such pages are sequentially retrieved from the disk and searched for the search key. In case of a primary key search, as soon as the tuple is found the search ends and the tuple is presented to the user. If the indexed attribute is not unique then each page is read entirely.

Search key k using BF-Tree

```
Binary search of root node; read the appropriate child node.
Recursive search the values of the internal node, read the appropriate child node until a leaf is reached.
Read $min\_key$ and $max\_key$ from the leaf node.
if key $k \in [min\_key, max\_key]$ then
    Probe all BFs with key $k$.
    for $\forall$ BF within current leaf with index $bid$ that matches do
        Load page $min\_pid + bid$ (false positive with $fpp$).
    end for
else
    Key $k$ does not exist.
end if
```

Algorithm 1: Search using BF-Tree

Creating and Updating a BF-Tree. For creating and updating a BF-Tree, the high-level B$^+$-Tree algorithms are still relevant. One important difference, however, is that, apart from maintaining the desired node occupancy, we have to respect the desired values for the BFs accuracy.

Split a BF-Tree node N to N_1, N_2

```
Create new nodes $N_1$ and $N_2$.
Node split may need to propagate.
$N_1$ $keys \in \left[N.min\_key, \frac{N.min\_key+N.max\_key}{2}\right)$
$N_1$ $pids \in [N.min\_pid, N.min\_pid]$
$N_2$ $keys \in \left(\frac{N.min\_key+N.max\_key}{2}, N.max\_key\right]$
$N_2$ $pids \in [N.max\_pid, N.max\_pid]$
for $k = min\_key$ to $max\_key$ do
    if key $k$ exists in $N$ then
        if $k \in N_1$ $keys$ then
            Update $N_1.max\_pid$ with $max(pid : k \in pid)$
            Update $N_1.\#keys$
        else
            Update $N_2.min\_pid$ with $min(pid : k \in pid)$
            Update $N_2.\#keys$
        end if
    end if
end for
Allocate $(N_1.max\_pid - N_1.min\_pid)$ BFs for $N_1$
Allocate $(N_2.max\_pid - N_2.min\_pid)$ BFs for $N_2$
for $\forall k \in N_1$ $keys$ do
    if key $k$ exists in $N$ in $pids$ then
        Insert $k$ in BFs corresponding to all $pids$
    end if
end for
for $\forall k \in N_2$ $keys$ do
    if key $k$ exists in $N$ in $pids$ then
        Insert $k$ in BFs corresponding to all $pids$
    end if
end for
```

Algorithm 2: Split a BF-Tree node

Let us assume that we have a relation R which is empty and we start inserting values and the corresponding index entries on index key k. The initial node of the BF-Tree is a BF node, as discussed. For each new entry we need to update i) the BF, ii) *#keys*, iii) possibly *min_key* or *max_key* and in some cases iv) the page range. When the indexed elements exceed the maximum number of elements that maintains the desired false positive probability fpp we have to perform a node split. Bulk load of an entire index can minimize creation overhead since we can precompute the values of BF-Tree's parameters and allocate the appropriate number of nodes more efficiently. If the tree is in update-intensive mode, each node can maintain a list of inserted/deleted/updated keys (along with their page information) in order to accumulate enough number of such operations to amortize the cost of updating the BF.

The lossy way to keep indexing information for a BF-Tree increases the cost of splitting a BF-leaf. Algorithm 2 shows that in order to split a BF- leaf we need to probe the initial node for all indexed values. Thus, splitting a leaf node is computationally expensive, but it can be accelerated because it is heavily parallelizable. During a node split several threads can probe the BFs of the old node in order to create the BFs of the new node.

Algorithm 3 shows how to perform an insert in a BF-Tree. After navigating towards the BF-leaf in question, we check the BF-leaf size. If this leaf has already indexed the maximum number of values then we perform a node split as described above. After this step, the values of the BF-leaf variables are updated. First, we increase the number of keys (#*keys*). Second, we may need to update the *min_key* or *max_key* accordingly. Third, the new key value is added to the corresponding BF.

Insert key k (stored on page p)

```
if $\#keys + 1 \leq max\_node\_size$ then
    if $k \notin [min\_key, max\_key]$ then
        Extend range (update $min\_key$ or $max\_key$).
    end if
    Increase $\#keys$.
    Insert $k$ into BF with index $p - min\_pid$ within current leaf.
else
    Split Node.
    Run insert routine for the newly created node.
end if
```

Algorithm 3: Insert a key in a BF-Tree

Partitioning. A BF-Tree works under the assumption that data is organized (ordered or partitioned) based on the indexing key. We can take advantage of the order of the data if it follows the indexing key, or an implicit order. For example, data like orders of a shop, social status updates or other historical data is usually ordered by date. Thus, any index on the date can use this information. Note, that we do not apply a specific order, we rather simply use the nature of the data for more efficient indexing.

Bulk loading BF-Trees. Similarly to other tree indexes, the build time of a BF-Tree can be aggressively minimized using bulk loading. In order to bulk load a BF-Tree the system creates packed BF-leaves with BFs and builds the remaining of the tree on top of the leaves level during a new scan of the leaves. Hence, bulk loading requires one pass over the data and one pass over the leaves of the BF-Tree.

5. MODELING BF-TREES

In this section we present an analytical model to capture the behavior of BF-Trees and compare them with B$^+$-Trees, and the flash-aware indexing techniques FD-Tree [26] and SILT [27], regarding size and performance. Since data is ordered or partitioned an alternative option for searching is to use interpolation search [36] or binary search. Interpolation search can be very effective for canonical datasets achieving $log(log(N))$ search time, in the specific case that the values are sorted and evenly distributed.[5] B$^+$-Trees' performance serves as a more general upper bound since binary search average response time is $log_2(N)$ and B$^+$-Trees average response time is $log_k(N)$, where N is the size of the dataset and the k is the number of *¡key, pointer¿* pairs a B$^+$-Tree page can hold. In addition, B$^+$-Trees serve as a baseline for comparing the size of an index structure used to enhance search performance.

Table 1 presents the key parameters of the model. Most of the parameters are sufficiently explained in the table, however, a number of parameters are further discussed. For a BF-Tree the average occurrence of a value of the indexed attribute (*avgcard*) plays an important role since no new information is stored in the index (effectively reducing its size). Moreover, the desired false positive probability (fpp) allows us to design BF-Trees with variable size and accuracy for exactly the same dataset. Two more parameters characterize BF-Trees: indexed values per leaf ($BFkeysperpage$) which is a function of the fpp and data pages per leaf ($BFpagesleaf$) which is a function of the first and it is related to performance since it is the maximum amount of I/O needed when we want to retrieve a tuple indexed in a BF-leaf. Finally, for the I/O cost there are three parameters, traversing the index (randomly), *idxIO*, random accesses to the data (*dataIO*) and sequential access to the data (*seqDtIO*). That way, we can alter the assumptions of the storage used for the index and the data: either keep them in the same medium (e.g., both on SSD) or store the index on the SSD and the data on HDD.

Table 1: Parameters of the model

Parametername	*Description*
pagesize	pagesize for both data and index
$tuplesize$	(fixed) size of a tuple
notuples	size of the relation in tuples
avgcard	avg cardinality of each indexed value
keysize	size of the indexed value (bytes)
ptrsize	size of the pointers (bytes)
$fanout$	fanout of the internal tree nodes
BPleaves	number of leaves for the B$^+$-Trees
BPh	height of the B$^+$-Trees
BPsize	size of the B$^+$-Trees
fpp	false positive probability for BF-Trees
$BFkeysperpage$	indexed keys per BF leaf
$BFpagesleaf$	data pages per leaf
$BFleaves$	number of leaves for the BF-Tree
BFh	height of the BF-Tree
BFsize	size of the BF-Tree
mP	number of matching pages per key
BPcost	cost of probing a B$^+$-Tree
BFcost	cost of probing a BF-Tree
idxIO	cost of a random traversal of the index
dataIO	cost to access randomly data
seqDtIO	cost to access sequentially data

Equation 2 is used to calculate the $fanout$ of the internal nodes of both BF-Trees and B$^+$-Trees. In Equation 3 we calculate the number of leaves of a B$^+$-Tree needed based on the data and the indexing details, while the height of the B$^+$-Tree is calculated with the Equation 4.

$$fanout = \frac{pagesize}{ptrsize + keysize} \tag{2}$$

$$BPleaves = \frac{notuples \cdot (\frac{keysize}{avgcard} + ptrsize)}{pagesize} \tag{3}$$

$$BPh = \lceil log_{fanout}(BPleaves) \rceil + 1 \tag{4}$$

In order to calculate the leaves of a BF-Tree we first need to calculate the different keys that a BF of a BF-leaf can index (solving in Equation 5, Equation 1 assuming the bits available in a page) and then plug in this number in Equation 6 where we make sure that we correctly calculate the size of the BF-Tree by discarding multiple entries of the same index key. Equation 7 calculates the height of the BF-Tree, and Equation 8 uses the number of different indexed values per BF-leaf to calculate the the number of data pages per BF-leaf.

$$BFkeysperpage = -pagesize \cdot 8 \cdot \frac{ln^2(2)}{ln(fpp)} \tag{5}$$

[5] A more widely applicable version of interpolation search has also been discussed [19].

$$BFleaves = \frac{notuples}{avgcard \cdot BFkeysperpage} \quad (6)$$

$$BFh = \lceil log_{fanout}(BFleaves) \rceil + 1 \quad (7)$$

$$BFpagesleaf = \frac{BFkeysperpage \cdot avgcard \cdot tuplesize}{pagesize} \quad (8)$$

Next, Equations 9 and 10 estimate the sizes of the trees.

$$BPsize = pagesize \cdot (BPleaves + \frac{BPleaves}{fanout}) \quad (9)$$

$$BFsize = pagesize \cdot (BFleaves + \frac{BFleaves}{fanout}) \quad (10)$$

Equation 11 calculates the average number of pages to be retrieved after a probe with a match (*mP*). Equation 12 calculates the cost of probing a B$^+$-Tree and reading the tuple from its original location. Note that for small *avgcard* the matching pages are equal to 1. If there is no match, *mP* is equal to 0.

$$mP = \lceil \frac{avgcard \cdot tuplsize}{pagesize} \rceil \quad (11)$$

$$BPcost = BPh \cdot idxIO + mP \cdot dataIO \quad (12)$$

Finally, Equation 13 calculates the cost of searching with the enhanced BF-Tree. In this equation we need to reintroduce the term *mP*, which is the number of matching pages when there a positive search on the index. We calculate the cost of a false positive as the cost to retrieve sequentially the false positively attributes pages since all these pages are calculated in search time and will be given to the disk controller as a list of sorted disk accesses.

$$\begin{aligned} BFcost =& BFh \cdot idxIO + mP \cdot dataIO + \\ &+ fpp \cdot BFpagesleaf \cdot seqDtIO \Rightarrow \\ BFcost =& BFh \cdot idxIO + mP \cdot dataIO + \\ &+ \frac{8 \cdot avgcard \cdot ln^2(2) \cdot fpp \cdot seqDtIO}{tuplesize \cdot ln(fpp)} \quad (13) \end{aligned}$$

Figure 4 presents key comparisons between BF-Tree, B$^+$-Tree, compressed B$^+$-Tree and two representative approaches for indexing over flash memory: FD-Tree [26] and SILT [27]. We assume 4KB pages of 256 bytes long tuples with the indexed attribute of size 32 bytes and pointers of size 8 bytes. The relation in this case has 1GB size, the index is stored on SSD and the main data on HDD. The I/O cost is depicted by using the appropriate values of *idxIO*, *dataIO*, and *seqDtIO*. In particular we use $idxIO = 1$, $dataIO = 50$, and $seqDtIO = 5$, modeling an SSD which has random accesses fifty times faster than random accesses on HDD and five times faster than sequential accesses on HDD. For the compressed B$^+$-Tree we calculate the size assuming that key-prefix compression [6, 20] is used, and for FD-Tree and SILT we use the modeling tools provided in their respective analyses [26, 27] to estimate size and performance for point queries.

On the x axes of Figures 4(a), (b) we vary the desired false positive probability, hence every line other than the one BF-Tree is a straight line in order to show where are - if any - the crossover points between BF-Tree and the other approaches. Figure 4(a) plots the response time of BF-Tree, SILT, and FD-Tree normalized with B$^+$-Tree. We see that BF-Tree can offer better search time for $fpp \leq 0.001$. Moreover, SILT can be 5% faster than B$^+$-Tree if the search cost of the trie is negligible (i.e., the trie is entirely cached). If the trie has to be loaded the response time is 32% higher, while on average the response time will be between the two values. SILT, however, is designed only for point queries for key-value stores and it does not support other access patterns (e.g., range scans) and systems (traditional DBMS). FD-Tree has very similar performance with the BF-Tree if the optimal value for *k* is chosen. Figure 4(b) shows the index size of the BF-Tree, the compressed B$^+$-Tree, SILT, and FD-Tree, normalized with the size of B$^+$-Tree. FD-Tree has the same size as vanilla B$^+$-Tree, while SILT has significant capacity savings, being 28% as large as the B$^+$-Tree. The compressed B$^+$-Tree has size about 10% of the B$^+$-Tree, while BF-Tree, has the same size as the compressed B$^+$-Tree for $fpp = 10^{-8}$. Hence, for the described workload if we maintain the $fpp \in [10^{-8}, 10^{-3}]$, BF-Tree offers the smallest size and performance within 5% of the fastest configuration.

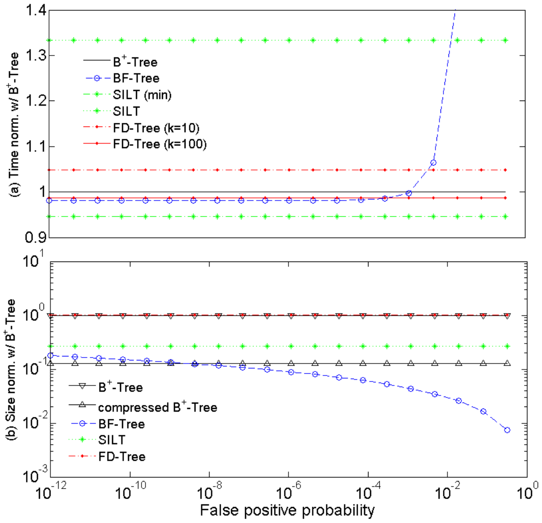


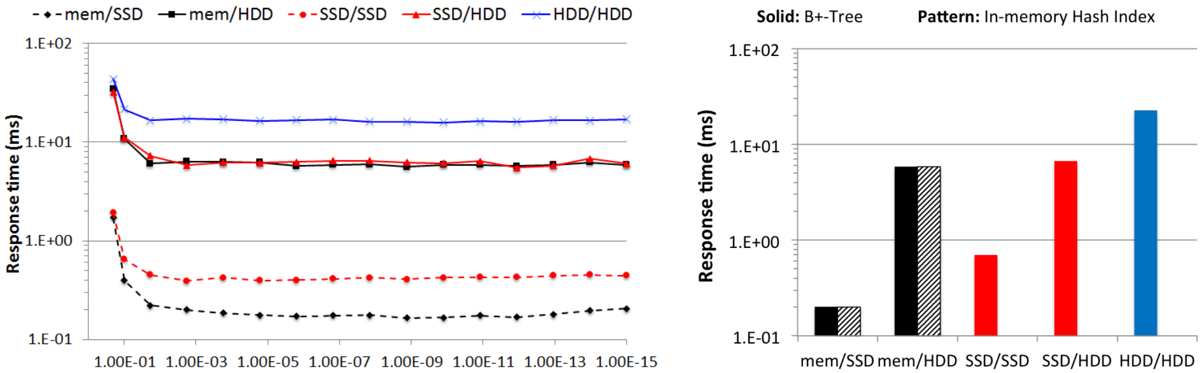
Figure 4: Analytical comparison of BF-Tree vs. B$^+$-Tree.

6. EXPERIMENTAL EVALUATION

We implement a prototype BF-Tree and we compare against a traditional B$^+$-Tree and an in-memory hash index. The BF-Trees are parametrized according to the false positive probability for each BF, which affects the number of leaf nodes needed for indexing an entire relation and, consequently, the height of the tree. The BF-Tree can be built and maintained entirely in main memory or on secondary storage. The size of a BF-Tree is typically one or more orders of magnitude smaller than the size of a B$^+$-Tree, so we examine cases where the BF-Tree is entirely in main memory and cases where the tree is read from secondary storage. We use stand-alone prototype implementations for both B$^+$-Trees and hash indexes. The code-base of the B$^+$-Tree with minor modifications serves as the part of the BF-Tree above the leaves. BF-Trees can be implemented in every DBMS with minimal overhead since they require to add support for BF-leaves, and build their methods as extensions of the typical B$^+$-Tree methods.

6.1 Experimental Methodology

In our experiments we use a server running Red-Hat Linux with 2.6.32 64-bit kernel. The server is equipped with 2 6-Core 2.67GHz Intel Xeon CPU X5650 and 48GB of main memory. Secondary storage for the data and indexes is either a Seagate 10KRPM HDD - offering 106MB/s maximum sequential throughput for 4KB pages



(a) BF-Tree performance varying fpp

(b) B^+-Tree/Hash Index performance

Figure 5: BF-Tree and B^+-Tree performance for the PK index for five storage configurations for storing the index and the main data.

- or a OCZ Deneva 2C Series SATA 3.0 SSD - with advertised performance 550MB/s (offering as much as 80kIO/s of random reads). Data on secondary storage (either HDD or SSD) are accessed with the flags O_DIRECT and O_SYNC enabled, hence *without using the file system cache*.

We experiment with a synthetic workload comprised of a single relation *R* and with TPCH data. For the synthetic workload, the size of each tuple is 256 bytes, the primary (PK) key is 8 bytes and the second attribute we index (ATT1) has size 8 bytes as well, having, however, each value repeated 11 times on average. Both attributes *are ordered* because they are correlated with the creation time. For the TPCH data, we use the three date columns of the lineitem table with scale factor 1. The tuple size is 200 bytes and the indexed attribute is shipdate on which the tuples are ordered. Each date of the shipdate is repeated 2400 times on average. Every experiment is the average of a thousand index searches with a random key. The same set of search keys is used in each different configuration. In the experiment we vary the (i) the false positive probability (fpp) to understand how the BF-Tree is affected by different values and (ii) the indexed attribute in order to show how BF-Tree indexing behaves for a PK and for a sorted attribute. The BFs are created using 3 hash functions, typically enough to have hashing close to ideal. Throughout the experiments the page size is fixed to 4KB. Hence, in order to vary the fpp, the maximum number of keys per BF-leaf is limited accordingly using Equation 1.

When the B^+-Tree or the hash index is used during an index probe, the corresponding page is read and, consequently, the tuple in question is retrieved using the tuple id. For the BF-Tree probes, the index is used as described in Section 4.2.

6.2 BF-Tree for primary key

We first experiment with indexing the primary key (PK) of *R* having size 1GB. Each index key exists only once and the data pages are ordered based on it. All index probes in the experiment match a tuple, hence every probe retrieves data from the main file.

Build Time and size. The build time of the BF-Tree is one order of magnitude smaller than the build time of the corresponding B^+-Tree following roughly the difference in size between the two trees shown in Table 2. The bulk creation of the BF-Tree first creates the leaves of the tree in an efficient sequential manner and then builds on top the remainder of the tree (which is 2-3 orders of magnitude smaller) to navigate towards the desired leaf. The principal goal of the BF-Tree, however, is to minimize the required space. Varying the fpp from 0.2 to 10^{-15} the size of a BF-Tree is 48x to 2.25x time smaller than the corresponding B^+-Tree. Both grow linearly with the number of data pages indexed in terms of the total number of nodes. The capacity gain as a percentage of the size of the corresponding B^+-Tree remains the same for any file size.

Response time. Figure 5(a) shows on the y-axis the average response time for probing the BF-Tree index of a 1GB relation using the PK index. Every probe reads a page from the main file and returns the corresponding tuple. On the x-axis we vary the fpp from 0.2 to 10^{-15}. As the fpp decreases (from the left-to-right on the graph) the BF-Tree is getting larger but it offers more accurate indexing. The five lines correspond to different storage configurations. The lines with the solid color represent experiments where data are stored on the HDD and the index either in memory (black), on the SSD (red) or on the HDD (blue). The dotted lines show the experiments where data are stored on SSD and the index either in memory (black) or on SSD (red). In order to compare the performance of BF-Tree with that of a B^+-Tree and a hash index we show in Figure 5(b) the response time of a B^+-Tree using the same storage configurations and the hash index when index resides in memory. Note that in this experiment the B^+-Tree and every BF-Tree has height equal to 3.

Table 2: B^+-Tree & BF-Tree size (pages) for 1GB relation

Variation	fpp	Size for PK	Size for ATT1
B^+-Tree		19296	1748
BF-Tree	0.2	406	38
BF-Tree	0.1	578	54
BF-Tree	$1.5 \cdot 10^{-7}$	3928	358
BF-Tree	10^{-15}	8565	786

Table 3: False reads/search for the experiments with 1GB data

fpp	False reads for PK	False reads for ATT1
0.2	13.58	701.15
0.1	1.23	80.93
$1.9 * 10^{-2}$	0.11	4.75
$1.8 * 10^{-3}$	0	0.36
$1.72 * 10^{-4}$	0.01	0.04

Data on SSD. When the index resides in memory and data reside on SSD BF-Tree manages to match the B^+-Tree performance for $fpp \leq 1.8 \cdot 10^{-3}$, leading to capacity gain 12x. This is connected with the number of falsely read pages per search (see Table 3), which are virtually zero for $fpp \leq 1.8 \cdot 10^{-3}$. We observe, as well, a very slow degradation of performance as fpp is getting close to 10^{-15}, which is attributed the larger index size (see Table 2); for every positive search we have to read more leaves. We compare the in-memory search time with a hash index which performs similarly to the memory-resident B^+-Tree and hence the optimal

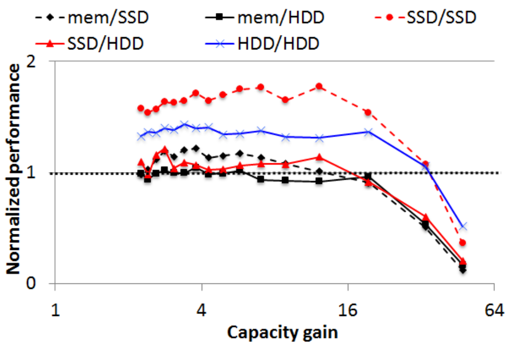


Figure 6: The break-even points when indexing PK

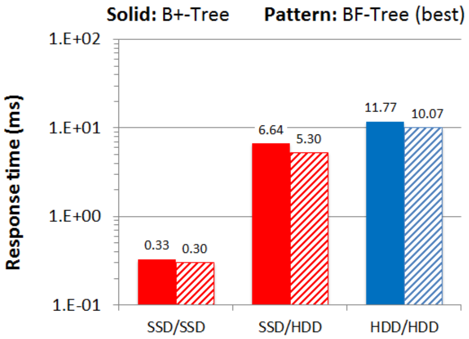


Figure 7: BF-Tree and B$^+$-Tree with warm caches

BF-Tree. When index is on SSD as well, increased fpp can be tolerated leading to capacity gain 33x with a low performance gain (1.08x), while we can still observe significant capacity gain (12x) with 1.77x lower response time, for $fpp = 0.002$. In the latter case we observe that a higher number of falsely read pages is tolerated because their I/O cost is faster amortized by of the I/O cost to retrieve the index.

Data on HDD. If the index is stored in memory and data on HDD, the PK BF-Tree can still provide significant capacity gains. The BF-Tree matches the B$^+$-Tree for $fpp \leq 1.56 \cdot 10^{-6}$, offering similar indexing performance while requiring 6x less space, but it is already competitive for $fpp = 0.02$ providing capacity gain 19x. The in-memory hash index search shows similar performance to the memory-resident B$^+$-Tree and hence the optimal BF-Tree. If the index is stored on SSD, the BF-Tree outperforms B$^+$-Tree for $fpp \leq 10^{-3}$ offering 12x smaller index size. Finally, if the index is stored on HDD as well then the height of trees dominate performance. Having to read from disk the pages of the two trees (starting from the root) leads to a BF-Tree which can outperform the corresponding B$^+$-Tree even for $fpp = 0.1$ (leading to more than 33x smaller indexing tree). This case however is not likely to be realistic because the nodes of the higher levels of a B$^+$-Tree reside always in memory.

Break-even points. A common observation for all storage configurations is that there is a *break-even point* in the size of a BF-Tree for which it performs as fast as a B$^+$-Tree. Figure 6 shows on the y-axis the normalized performance of BF-Trees compared with B$^+$-Trees. The x-axis shows the capacity gain: the ratio between the size of the B$^+$-Tree and the size of the BF-Tree for a given fpp. For normalized performance higher than 1, the BF-Tree outperforms the B$^+$-Tree (it has lower response time), for lower than 1 the other way around, and for values equal to 1, the BF-Tree has the same response time as the B$^+$-Tree. The cross-sections between the line with normalized performance equal to 1 and the five lines for the various storage configuration give us the break-even points. As the I/O cost increases (going from memory to SSD or from SSD to HDD) the break-even point shifts towards larger capacity gains, since less accuracy can be tolerated as long as it requires more CPU time (for example probing more BFs, or scanning more tuples in memory) instead of more random reads on the storage medium.

Warm caches. In order to see the efficacy of BF-Trees with warm caches, in Figure 7 we summarize the response time of B$^+$-Trees and the best response time of the fastest BF-Tree for the available storage configurations. Figure 7 has only three sets of bars because, trivially, the first two sets of bars correspond to the data presented in Figures 5(a) and (b). In the experiments with warm caches, only accessing the leaf node would cause an I/O operation, hence, the height of the tree is not a crucial factor for the response time any more. Moreover, since, typically, the B$^+$-Tree is higher, its performance improvement with warm caches is higher compared to the BF-Tree performance improvement. Comparing the first set of bars of Figure 7 with the third bar of Figure 5(b) and the third line of Figure 5(a) – both corresponding to the storage configuration when both index and data reside on SSD – we observe that having warm caches results in 2x improvement for B$^+$-Tree and only 25% improvement for BF-Tree, however, still leading to a 10% faster BF-Tree. For the SSD/HDD configuration the improvement is small for both indexes because the bottleneck is now the cost to retrieve the main data. Last but not least, when both index and data reside on HDD, the cost of traversing the index is significantly reduced by almost 2x for B$^+$-Tree and about 33% for BF-Tree, resulting in a 17% faster BF-Tree.

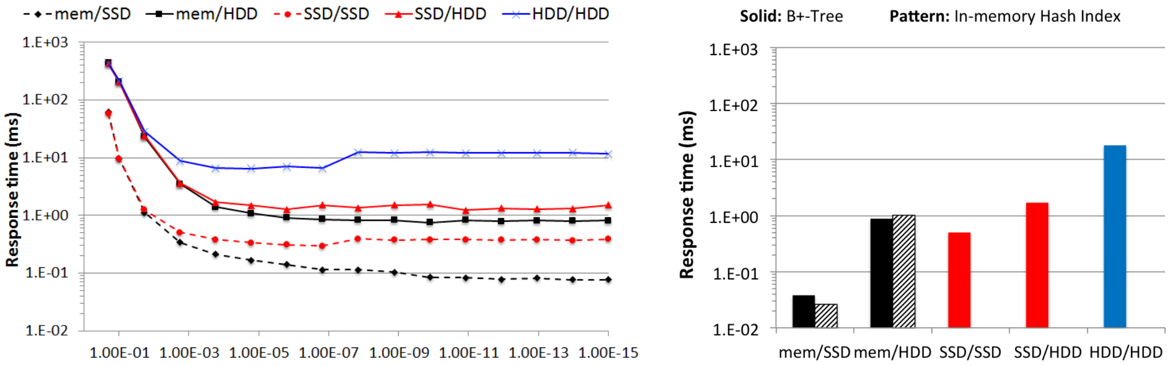
By having warm caches, the fundamental difference is that the height of three plays a smaller part in the response time. Nevertheless, the response time of BF-Tree is lower than the one of B$^+$-Tree in every storage configuration because of the lightweight indexing. The gain in response time depends on the behavior of the storage for the index and for the data. When index and data are stored on the same medium (i.e., both on SSD, or both on HDD) there is small room for improvement, however, when data reside in a slower medium than index, having a lightweight index makes a larger difference.

6.3 BF-Tree for non-unique attributes

In the next experiment we index a different attribute which does not have unique values. In particular, in the synthetic relation R the attribute ATT1 is a timestamp attribute. In this experiment 14% of the index probes, on average, have a match.

Build time and Size. The build time of a BF-Tree is one order of magnitude or more lower than the one of a B$^+$-Tree. This is attributed to the difference in size which for attribute ATT1 varies between 46x and 2.22x, for fpp from 0.2 to 10^{-15}, offering similar gains with the PK index.

Response time. Figure 8(a) shows on the y-axis the average response time for probing the BF-Tree using the index on ATT1, as a function of the fpp. For the B$^+$-Tree every probe with a positive match will read all the consecutive tuples that have the same value as the search key. When the BF-Tree is used for the positive matches (regardless whether they are false positives or actual positive matches) the corresponding page is fetched and every tuple of that page has to be read and checked whether it matches the search key (as long as the key of the current tuple is smaller than the search key). The fpp varies from 0.2 to 10^{-15}. Similarly to the previous experiment, the five lines correspond to different storage configurations. As in Section 6.2, the lines with the solid color represent experiments where data are stored on the HDD and the index either



(a) BF-Tree performance varying fpp

(b) B^+-Tree/Hash Index performance

Figure 8: BF-Tree and B^+-Tree performance for ATT1 index for five storage configurations for storing the index and the main data.

in memory (black), on the SSD (red) or on the HDD (blue). The dotted lines show the experiments where data are stored on SSD and the index either in memory (black) or on SSD (red). We compare the performance of BF-Tree with that of a B^+-Tree and a hash index (Figure 8(b)) using the same storage configurations (the hash index always reside in memory). For this experiment the B^+-Tree has 3 levels while, the BF-Trees have 2 levels for $fpp > 1.41 \cdot 10^{-8}$ and 3 levels for $fpp \leq 1.41 \cdot 10^{-8}$ Because of the difference in height amongst the BF-Trees we observe a new trend. For all configurations that the index probe is a big part of the overall response time (i.e., the SSD/SSD and the HDD/HDD) we observe a clear increase in response time when the height of the tree is increased (Figure 8(a)). In other configurations this is not the case because the index I/O cost is amortized by the data I/O cost. This behavior exemplifies the trade-off between fpp and performance regarding the height of the tree and is later depicted in Figure 9 when the best case for BF-Tree gives 2.8x performance gain for HDD and 1.7x performance gain for SSD while the capacity gain is 5x-7x.

Data on SSD. When the data reside on the SSD and the fpp is high (0.3 to 10^{-3}) there is very small difference between storing the index on SSD or in memory. The reason behind that is that the overhead of reading and discarding false positively retrieved pages dominates the response time. Thus, in order to match the response time of the B^+-Tree when index is on SSD the fpp has to be $2 \cdot 10^{-3}$, giving capacity gain 12x, while the in memory BF-Tree never matches the in-memory B^+-Tree nor the hash index.

Data on HDD. The increased number of false positives per index probe (Table 3) has increased impact when data are stored on HDD. Every false positively retrieved page incurs an additional randomly located disk I/O. As a result, in order to see any performance benefits the false positively read pages have to be minimal. When the index resides in memory, the BF-Tree outperforms the B^+-Tree and the hash index for $fpp \leq 2 \cdot 10^{-6}$. The break-even point when the index resides on the SSD is shifted for higher fpp ($2 \cdot 10^{-3}$) because a small number of unnecessary page reads can be tolerated as they are being amortized by the cost of accessing the index pages on the SSD. This phenomenon has bigger impact when the index is stored on the HDD. The break-even point is now further shifted and for capacity gain 12x there is already a performance gain of 2x.

Break-even points. Similarly to the PK index, the ATT1 BF-Tree indexes have break-even points, shown in Figure 9. Qualitatively, the behavior of the BF-Tree for the five different storage configurations is similar but the break-even points are now shifted towards smaller capacity gains. The reason being mainly the increased number of false positively read pages. In order for the BF-Tree to be competitive it needs to minimize the false positives. On the other hand, the impact of increasing the false positive (and thus, reducing the tree height) is shown by the sudden increase in performance gain particularly for the HDD/HDD (blue line) and the SSD/SSD (dotted red line).

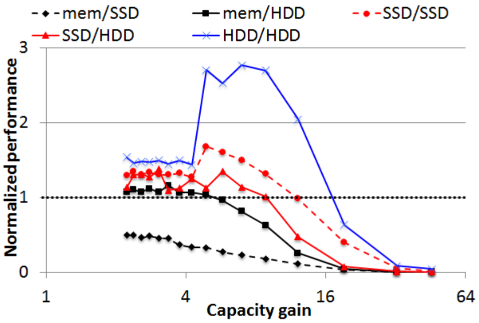


Figure 9: The break-even points when indexing ATT1

Benefits for HDD. Our experimentation shows that using BF-Trees when data reside on HDD can lead to higher capacity gains before performance starts decreasing. The symmetric way to see this, is that with the same capacity requirements using BF-Trees on top of HDD gives higher performance gains (in terms of normalized performance). This is largely a result of the enhancement algorithm of BF-Trees which minimizes the occurrence of random reads, the main weak point of HDD.

Warm caches. Similarly to the PK experiments, we repeat the ATT1 experiments and we present the results with warm caches in Figure 10. As in the PK index case, the improvement for the B^+-Tree is higher than the improvement for BF-Tree. In addition, for the storage configuration that both index and data reside on SSD, B^+-Tree is, in fact, faster than BF-Tree by 30% because the overhead of the false positives overthrow the marginal benefits of the lightweight indexing. For the other two configurations, however, BF-Tree is faster (2.5x for SSD/HDD and 1.5x for HDD/HDD) because any additional work is hidden by the cost of retrieving the main data, now residing on HDD.

6.4 BF-Tree for TPCH

Figure 11 shows the response time of the optimal BF-Tree normalized with the response time of the B^+-Tree for the TPCH table lineitem for index probes on the shipdate attribute (on which it is partitioned). Similarly to the five configurations previously described, Figure 11 has five lines. The y-axis shows normalized response time in a logarithmic scale. BF-Tree performance for different fpp has very low variation, because of the fact that the high

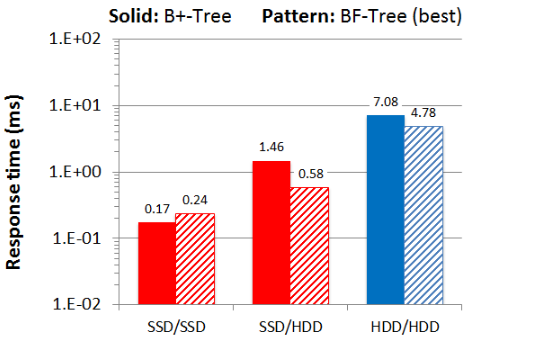


Figure 10: BF-Tree and B$^+$-Tree for ATT1 with warm caches

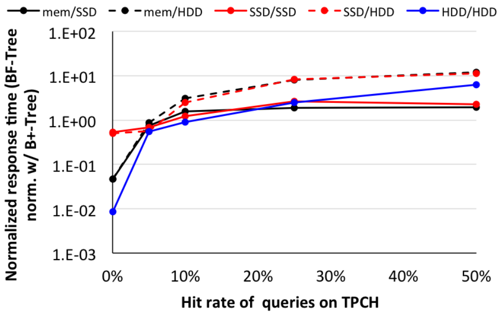
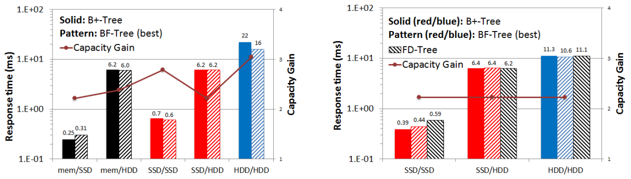


Figure 11: BF-Tree for point queries on TPCH varying hit rate

cardinality of each date results in short trees. We observe, however, large differences in the behavior of BF-Trees vs. the one of B$^+$-Trees for different hit rates. When all index probes end up requesting data that do not exist (hit rate 0%), BF-Tree is 20x faster than B$^+$-Tree when the index is in memory. When the index is stored on SSD, BF-in secondary storage a similar behavior is observed. This effect is pronounced for HDD where each additional level of the tree is adding a high overhead. As we increase the hit rate to 5% BF-Tree is still faster but the benefit is much smaller (13% - 40%) because the time to retrieve data starts dominating the response time. For hit rate 10% or more B$^+$-Tree in general has faster response time than BF-Tree. If both data and index are stored in the same medium (i.e., both on SSD or both on HDD) the performance penalty is smaller. Hence, for hit rate 10%, using the configuration SSD/SSD the best BF-Tree is 20% slower than the B$^+$-Tree and using the configuration SSD/SSD the best BF-Tree is 10% faster than the B$^+$-Tree (instead of 1.6 to 3x slower which is the case for the other three configurations). The explanation is that for these two configurations the cost of traversing the B$^+$-Tree dominates the overall response time and BF-Tree is competitive because of its shorter height. In the above experiments the BF-Trees were 1.6x-4x smaller than the B$^+$-Trees.

6.5 BF-Tree and FD-Tree for SHD

Figures 12(a) and (b) show the response time of the optimal BF-Tree compared with B$^+$-Tree and FD-Tree when indexing and probing data from the smart home dataset. In these experiments the index is built on the timestamp of the SHD, which has average cardinality 52 keys for every timestamp (cardinality varies from 21 to 8295, with 99.7% of the timestamps having cardinality less or equal to 126). The SHD dataset, which serves as motivation for the synthetic datasets as well, presents the additional challenge of having variable cardinality. The executed workload consists of index probes on randomly generated timestamps with 100% hit rate, which is the hardest case for BF-Trees as discussed in the case for the TPCH data in Section 6.4. Figure 12(a) shows the average response time of the optimal BF-Tree and B$^+$-Tree with cold caches for the five storage configurations, along with the corresponding capacity gain. We observe that BF-Tree matches the performance of B$^+$-Tree offering capacity gain between 2x and 3x following the trend that when index and data are stored on storage devices with similar capabilities BF-Tree has better response time and higher capacity gain than the B$^+$-Tree. Figure 12(b) shows the average response time of the optimal BF-Tree, B$^+$-Tree and FD-Tree with warm caches for the three storage configurations, along with the corresponding capacity gain of BF-Tree vs. B$^+$-Tree. Since we are using the original FD-Tree code, we compare against both BF-Tree and B$^+$-Tree with warm caches (following the original code approach). Further, we allow the FD-Tree to choose the optimal parameters based on the prediction algorithm it uses and we make sure that FD-Tree fetches the entire tuple for the index probe (similarly to what an index probe with BF-Tree and B$^+$-Tree does). Corroborating the results from the analysis in Section 5, FD-Tree has very similar performance than both BF-Tree and B$^+$-Tree when data resides on hard disk and is about 33% slower than BF-Tree and 50% slower than B$^+$-Tree when both index and data reside on SSD.



(a) Cold caches

(b) Warm caches (also FD-Tree)

Figure 12: BF-Tree and B$^+$-Tree for smart house workload.

6.6 Summary

In this section we compare BF-Trees with B$^+$-Trees, hash indexes and FD-Tree in order to understand whether BF-Trees can be both space and performance competitive. We show that depending on the indexed data and the storage configuration there is a BF-Tree design that can be competitive with B$^+$-Trees both in terms of size and response time. Moreover, for the in-memory index configurations, BF-Trees are competitive against hash indexes as well. The number of false positively read pages has an impact in each and every one of the five storage configurations. When data reside on SSD the number of falsely read pages affect the performance only when it dominates the response time. Random reads do not hurt SSD performance, hence, the BF-Tree performance drops gradually with the number of false positives. When data are stored on HDD the performance gains are high for no false positives but drop drastically as soon as unnecessary reads are introduced. Finally, corroborating the analysis FD-Tree offers very similar performance with BF-Tree when data reside HDD and is slightly slower when data reside on SSD.

7. BF-TREE AS A GENERAL INDEX

BF-Tree vs. interpolation search. The datasets used to experimentally evaluate BF-Tree are either ordered or partitioned. For the case that data are ordered a good alternative candidate would be to use binary search or interpolation search [36]. Interpolation search can be very effective for canonical datasets achieving $log(log(N))$ search time, in the specific case that the values are sorted and evenly distributed.[6] B$^+$-Trees performance serves as a more general upper bound since binary search average response time is $log_2(N)$ and B$^+$-Trees average response time is $log_k(N)$, where N is the size of

[6] A more widely applicable version of interpolation search has also been discussed [19].

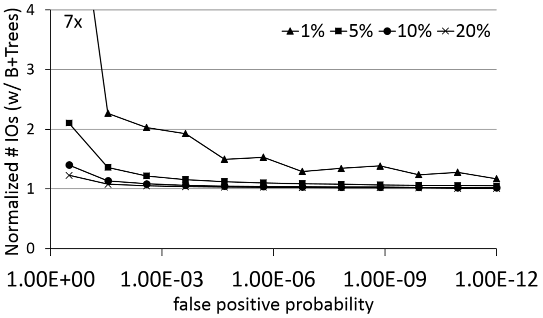


Figure 13: IOs for range scans using BF-Trees vs. B$^+$-Trees.

the dataset and the k is the number of <*key, pointer*> pairs a B$^+$-Tree page can hold. In addition, B$^+$-Trees serve as a baseline for comparing the size of an index structure used to enhance search performance. BF-Trees can be used as a general index which is further discussed in this section. Data do not need to be entirely ordered (being partitioned is enough), hence, BF-Tree is a more general access method than interpolation search.

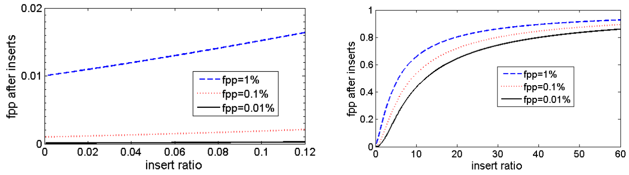
Range scans. In addition to the evaluation with point queries here we discuss how BF-Tree support range scans, showing that they have competitive performance for range scans. The BF-leaf corresponds to one partition of the main data. When a range scan spanning multiple BF-Tree partitions is evaluated, the partitions are either entirely part of the range, being *middle partitions*, or part of the boundaries of the range, being *boundary partitions*. The boundary partitions, typically, are not part of the range in their entirety. In this case, the range scan shows a read overhead. An optimization is to enumerate the values corresponding to the boundary partitions and probe the BFs in order to read only the useful pages. Note that within the partition the pages do not need to be ordered on the key. Such an optimization, however, is not practical when the values have a theoretically indefinite domain, or even a domain with very high cardinality. Figure 13 shows the number of I/O operations on the main data when executing a range scan using a BF-Tree, normalized with the number of I/O operations needed when a B$^+$-Tree is used. In the x-axis the fpp is varied from 0.3 to 10^{-12}. The four lines correspond to different ranges, varying from 1% to 20%. We use the synthetic dataset described in Section 6, and the indexed attribute is the primary key. A key observation is that as the fpp decreases the partitions hold less values, hence, the overhead of reading the boundary partitions decreases. Hence, we observe that for $fpp \leq 10^{-4}$ for ranges larger than 5% there is negligible overhead. In the case of 1% range scan with $fpp \leq 10^{-6}$, the overhead is less than 20%, and for $fpp \leq 10^{-12}$ the overhead is negligible for every size of the range scan.

Impact of inserts and deletes. Both inserts and deletes affect the fpp of a BF. In particular, if every BF is allowed to store additional entries for the values falling into their range the fpp is going to gradually increase. If we assume M bits for the BF, and a given initial fpp, then using Equation 1, we calculate that such a BF indexes up to $N = -M \cdot ln^2(2)/ln(fpp)$ elements. If we allow to index more elements, i.e., increase N, the effective fpp will in turn increase following a near linear trend for small changes of N (note that the new_N is the number of indexed elements after a number of inserts, hence $new\_N = N + inserts$):

$$new\_fpp = e^{\frac{-M \cdot ln^2(2)}{new\_N}}, \quad \text{using Equation 1 we have,}$$

$$new\_fpp = e^{\frac{N}{new\_N} ln(fpp)} = fpp^{\frac{N}{new\_N}} = fpp^{\frac{N}{N+inserts}} \Rightarrow$$

$$new\_fpp = fpp^{\frac{1}{1+\frac{inserts}{N}}} = fpp^{\frac{1}{1+insert\_ratio}} \qquad (14)$$



(a) Inserting up to 12% more keys

(b) Inserting up to 10x more keys

Figure 14: fpp in the presence of inserts.

The new_fpp depends on the number of inserts. Equation 14 does not depend on the BF size, nor on the number of elements. It depends on the initial fpp and on the relative increase of indexed elements ($insert\_ratio = \frac{inserts}{N}$).

The behavior of a BF in the presence of inserts is depicted in Figures 14(a) and (b). The x-axis is the insert ratio, i.e., the number of inserts as a percentage of the initially indexed elements and the y-axis is the resulting fpp. There are three lines corresponding to initial fpp equal to 0.01% (black line), 0.1% (red dotted line), and 1% (blue dashed line) respectively. In the x-axis of Figure 14(a) the insert ratio varies between 0% and 12%. The new_fpp for every initial fpp increases linearly for this range of insert ratio. For higher insert ratio, new_fpp eventually converges to 100%. We observe that all three lines have a liner trend, and more importantly, the line corresponding to 0.01% has a small increase even for 12% increase of the indexed elements. For example, starting from $fpp = 0.01\%$, for 1% more elements, $new\_fpp \approx 0.011\%$, and for 10% more elements, $new\_fpp \approx 0.23\%$. In Figure 14(b) the x-axis varies between 0% and 600% and we observe how the fpp is affected in the long run. Figures 14(a) and (b) show that BF-Tree can sustain a number insert without any updates as long as they represent a fraction of up to 15%, while when more inserts or updates are applied the index should be accordingly updated.

Similarly, for deletes the fpp increases because we artificially introduce more false positives. In fact, the number of deletes affects directly the fpp. If we remove 10% of the entries, $new\_fpp = fpp + 10\%$. The above analysis assumes no space overhead. On the other hand, we can maintain the desired fpp by splitting the nodes when the maximum tolerable fpp is reached. That way inserts will not affect querying accuracy (as described in Section 4). We can maintain a list of deleted keys in order to avoid increasing the fpp, which are used to recalculate the BF from the beginning when such a list has reached the maximum size. A different approach is to exploit variations of BFs that support deletes [7, 39] after considering their space and performance characteristics.

8. OPTIMIZATIONS

Exploiting available parallelism. With the current approach the BFs of a leaf node are probed sequentially. A BF-leaf may index several hundreds data pages leading to an equal number of BFs to probe. These probes can be parallelized if there are enough CPU resources available. In the conducted experiments we do not see a bottleneck in BF probing, however, for different applications, BF-Trees may exhibit such a behavior.

Complex index operations with BF-Trees. In this paper, we cover how the index building and the index probing are performed and we briefly discuss how updates and deletes are handled. Traditional indexes, however, offer additional functionality. BF-Trees support *index scans* similarly to a range scan. Since data are partitioned, with one index probe we find the starting point of the scan, and then a sequential scan is performed. *Index intersections* are also possible. In fact, the false positive probability for any key after the intersection of two indexes will be the product of the probability for each index, and hence, typically much smaller than both.

9. RELATED WORK

Specialized optimizations for tree indexing. SB-Tree [34] is designed to support high-performance sequential disk access for long range retrievals. It assumes an array of disks from which it retrieves data or intermediate nodes should they not be in memory, by employing multi-page reads during sequential access to any node level below the root. SB-Tree and B$^+$-Tree have similar capacity requirements. On the contrary, BF-Tree aims at indexing partitioned data, at decreasing the size of the index, and at taking advantage of the characteristics of SSD. Litwin and Lomet [28] introduce a generic framework for index methods called the bounded disorder access method. The method, similarly to BF-Trees, aims at increasing the ratio of file size to index size (i.e., to decrease the size of the index) by using hashing for distributing the keys in a multi-bucket node. The bounded disorder access method does not decrease the index size aggressively for good reason, since by doing that it would cause more random accesses on the storage medium which is assumed to be traditional hard disks. On the contrary, BF-Trees, propose aggressive reduction of the index size by utilizing a probabilistic membership-testing data structure as a form of compression. The performance of such an approach is competitive because the SSD can sustain several concurrent read requests with zero or small penalty [4, 38] and the penalty of (randomly) reading pages due to a false positive is lower on SSD compared with HDD.

Indexing for data warehousing. Efficiently indexing of ordered data having small space requirements is recognized as a desired feature by commercial systems. Typically data warehousing systems use some form of sparse indexes to fulfil this requirement. The Netezza data warehousing appliance uses ZoneMap acceleration [17], a lightweight method to maintain a coarse-index, to avoid scanning rows that are irrelevant to the analytic workload, by exploiting the natural ordering of rows in a data warehouse. Netezza's ZoneMap indexing structure is based on the Small Materialized Aggregates (SMA) earlier proposed by Moerkotte [30]. SMA is a generalized version of a Projection Index [35]. A Projection Index is an auxiliary data structure containing an entire column - similar to how column-store systems (e.g., MonetDB, Vertica) today physically store data. Instead of saving the entire column, SMA store an aggregate per group of contiguous rows or pages, thus, enhancing queries calculating aggregates than can be inferred by the stored aggregate. In addition to primary indexes or indexes on the order attribute, secondary, hardware-conscious data warehouse indexes have been designed to limit access to slow IO devices [40].

10. CONCLUSIONS

In this paper, we make a case for approximate tree indexing as a viable competitor of traditional tree indexing. We propose the Bloom filter tree (BF-Tree) which is able to index a relation with sorted or partitioned attributes. BF-Trees parametrize the accuracy of the indexing as a function of the size of the tree. Thus, B$^+$-Trees are the extreme where accuracy and size are maximum. BF-Trees allow to decrease the accuracy and, hence, the size of the index structure by introducing (i) a small number of unnecessary reads and (ii) extra work to locate the desired tuple in a data page. We show, however, through both an analytical model and experimentation with a prototype implementation, that the introduced overhead can be amortized, hidden, or even superseded by reducing the I/O accesses when a desired tuple is retrieved. Secondary storage is moving from HDD - with slow random accesses and cheap capacity - to the diametrically opposite SSD. BF-Trees achieve competitive performance and minimize index size for SSD, and even when data reside on HDD, BF-Trees index it efficiently as well.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments, feedback, and suggestions to improve the paper. This work was supported by the FP7 project BIGFOOT (grant n. 317858).

11. REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2(1):361–372, 2009.

[2] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable Bloom Filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.

[3] C. Antognini. Bloom Filters in Oracle DBMS. *White Paper*, 2008.

[4] M. Athanassoulis, B. Bhattacharjee, M. Canim, and K. A. Ross. Path processing using Solid State Storage. *ADMS*, 2012.

[5] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: efficient online updates in data warehouses. *SIGMOD*, pages 865–876, 2011.

[6] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.

[7] M. A. Bender et al. Don't Thrash: How to Cache Your Hash on Flash. *PVLDB*, 5(11):1627–1637, 2012.

[8] B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Com. ACM*, 13(7):422–426, 1970.

[9] D. Borthakur. Petabyte scale databases and storage systems at Facebook. *SIGMOD*, pages 1267–1268, 2013.

[10] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. *CIDR*, 2009.

[11] M. Canim, C. A. Lang, G. A. Mihaila, and K. A. Ross. Buffered Bloom filters on solid state storage. *ADMS*, 2010.

[12] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *OSDI*, pages 205–218, 2006.

[13] S. Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. *SIGMOD*, pages 73–86, 2009.

[14] T. Condie et al. Online Aggregation and Continuous Query support in MapReduce. *SIGMOD*, pages 1115–1118, 2010.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, pages 10–10, 2004.

[16] E. Doller. PCM and its impacts on memory hierarchy. 2009. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[17] P. Francisco. The Netezza Data Appliance: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks*, 2011.

[18] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. PCMLogging: reducing transaction logging overhead with PCM. *CIKM*, pages 2401–2404, 2011.

[19] G. Graefe. B-tree indexes, interpolation search, and skew. *DaMoN*, 2006.

[20] G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), 2006.

[21] J. Gray. Tape is dead, disk is tape, flash is disk. *CIDR*, 2007.

[22] J. Hughes. Revolutions in Storage. *MASSIVE*, 2013.

[23] IBM. Managing big data for smart grids and smart meters. *White Paper*, 2012.

[24] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim. mu-tree: an ordered index structure for NAND flash memory. *EMSOFT*, pages 144–153, 2007.

[25] I. Koltsidas and S. Viglas. Flashing up the storage layer. *PVLDB*, 1(1):514–525, 2008.

[26] Y. Li, B. He, J. Yang, Q. Luo, and K. Yi. Tree Indexing on Solid State Drives. *PVLDB*, 3(1):1195–1206, 2010.

[27] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. *SOSP*, pages 1–13, 2011.

[28] W. Litwin and D. B. Lomet. The Bounded Disorder Access Method. *ICDE*, pages 38–48, 1986.

[29] G. Lu, B. Debnath, and D. Du. A Forest-structured Bloom Filter with flash memory. *MSST*, pages 1–6, 2011.

[30] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. *VLDB*, pages 476–487, 1998.

[31] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Software Engineering*, 16(5):558–560, 1990.

[32] G.-J. Na, B. Moon, and S.-W. Lee. IPLB$^+$-tree for Flash Memory Database Systems. *J. Inf. Sci. Eng.*, pages 111–127, 2011.

[33] P. E. O'Neil. Model 204 Architecture and Performance. *HPTS*, pages 40–59, 1987.

[34] P. E. O'Neil. The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. *Acta Inf.*, 29(3):241–265, 1992.

[35] P. E. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. *SIGMOD*, pages 38–49, 1997.

[36] Y. Perl, A. Itai, and H. Avni. Interpolation search – a Log LogN search. *Commun. ACM*, 21(7):550–553, 1978.

[37] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2002.

[38] H. Roh, S. Park, S. Kim, et al. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. 5(4):286–297, 2011.

[39] C. Rothenberg, C. Macapuna, F. Verdi, and M. Magalhaes. The deletable Bloom filter: a new member of the Bloom family. *IEEE Comm. Letters*, 14(6):557–559, 2010.

[40] L. Sidirourgos and M. L. Kersten. Column imprints: a secondary index structure. *SIGMOD*, pages 893–904, 2013.

[41] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. *DaMoN*, 2009.

[42] S. Tarkoma, C. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Comm. Surv. & Tut's*, 14(1):131–155, 2012.

[43] J. Taylor. Flash at Facebook: The Current State and Future Potential. *Flash Mem. Sum.*, 2013.

[44] TPC. TPCH benchmark. http://www.tpc.org/tpch/.

[45] D. Tsirogiannis, S. Harizopoulos, M. Shah, J. Wiener, and G. Graefe. Query processing techniques for solid state drives. *SIGMOD*, pages 59–72, 2009.