UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

Optimizing Large Databases: A Study on Index Structures

Proyecto de Investigación

.

Ricardo Andres Leon Ruiz

Ingeniería en Sistemas

Trabajo de titulación presentado como requisito
para la obtención del título de
Ingeniero en Sistemas

Quito, 22 de Diciembre de 2017

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

COLEGIO DE CIENCIAS E INGENIERÍAS

HOJA DE CALIFICACIÓN
DE TRABAJO DE TITULACIÓN

Optimizing Large Databases: A Study on Index Structures

Ricardo Andres Leon Ruiz

Calificación: _____

Nombre del profesor, Título académico: Aldo Cassola, Ph.D.

Firma del profesor: _____

Quito, Diciembre de 2017

Derechos de Autor

Por medio del presente documento certifico que he leído todas las Políticas y Manuales de la Universidad San Francisco de Quito USFQ, incluyendo la Política de Propiedad Intelectual USFQ, y estoy de acuerdo con su contenido, por lo que los derechos de propiedad intelectual del presente trabajo quedan sujetos a lo dispuesto en esas Políticas.

Asimismo, autorizo a la USFQ para que realice la digitalización y publicación de este trabajo en el repositorio virtual, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

Firma del estudiante: _____

Nombres y Apellidos: Ricardo Andres Leon Ruiz

Código de estudiante: 110346

C. I.: 1714286265

Lugar, Fecha Quito, 22 de Diciembre de 2017

RESUMEN

La siguiente investigación trata sobre comparar estructuras de índice para grandes bases de datos, tanto analíticamente, como experimentalmente. El estudio se encuentra dividido en dos partes principales. La primera parte se centra en índices de hash y B-trees. Ambas estructuras son estudiadas en el contexto del modelo de acceso de disco tradicional. La segunda parte presenta al modelo cache-oblivious, incluyendo sus implicaciones en el diseño de algoritmos para niveles de memoria arbitrarios. Dentro de este modelo, se estudian dos estructuras: el cache-oblivious B-tree, y el lookahead array. Usando bases de datos que contienen varios millones de filas, los resultados experimentales confirman al B-tree como la opción preferida entre las estructuras convencionales. Sin embargo, comparaciones contra el lookahead array señalan a esta nueva estructura como una alternativa eficiente para actualizar y almacenar cantidades masivas de datos

PALABRAS CLAVE

I/O: Ingresar datos desde disco hacia la memoria principal, enviar datos desde la memoria principal al disco.
Bloque: Unidad de información que es escrita desde o hacia disco. Los bloques contienen registros. Equivalentemente, podemos referirnos a los bloques como páginas.
Índice: Archivo que permite la organización de registros de datos con el fin de optimizar operaciones de búsqueda de registros. Un índice brinda la posibilidad de obtener registros de manera eficiente, dada la clave de búsqueda de dicho índice. Una colección de registros de datos puede tener múltiples índices, cada uno con una clave de búsqueda distinta. Esto permite acelerar operaciones de búsqueda sin considerar la organización de archivos que almacenen a los registros en sí.
Elemento del Índice: A los registros que se hallan en un archivo de índice se los conoce como elementos del índice.
Tiempo de Acceso: Tiempo necesario para buscar un registro, o conjunto de ellos desde una estructura de índice.
Tiempo de Inserción: Tiempo necesario para insertar un registro en una estructura de índice.
Costo de Espacio: Espacio requerido para almacenar una estructura de índice.

ABSTRACT

The following research is about comparing index structures for large databases, both analytically and experimentally. The study is divided into two main parts. The first part is centered around hash-based indexing and B-trees. Both of which are set in the context of the widely known external memory model. The second part presents the cache-oblivious model, describing its implications on the design of algorithms for any arbitrary pair of memory levels. Within this model, two structures are studied: the cache-oblivious B-tree, and the lookahead array. Experimental results for databases containing several million rows confirm the B-tree as the preferred choice among conventional structures. However, comparisons against the more modern lookahead array, point to the newer structure as more fit to efficiently update and store large amounts of data.

KEY TERMS

I/O: Input data from disk to main memory. Output data from main memory to disk.
Block: Unit of information read from or written to disk. Blocks contain records. Equivalently, we can refer to blocks as *pages*.
Index: File that enables organization of data records on disk in order to optimize retrieval operations. It allows to efficiently retrieve records that satisfy a specific search condition, based on the *search key* field of the index. A given collection of data records can have multiple indexes, each with a different search key field. This enables to speed up search operations, regardless of the file organization used to store records.
Index Entry: Records stored in an index file are called index entries.
Access Time: Time required to retrieve a particular record, or set of records from an index structure.
Insertion Time: Time required to insert a record into an index structure.
Space Overhead: Space required to store an index structure.

TABLE OF CONTENTS

INDEX OF FIGURES

INTRODUCTION

From a historical standpoint, the ever-increasing demand for computing performance has often forced researchers to develop systems and techniques that take advantage of the available hardware. Particularly, the recent advances in networking and the development of cheap, high-throughput hardware systems has led to the prominence of massive amounts of data being processed around the world. Such conditions, have massively outgrown the expectations of experts and scientists from decades past. We can expect this trend to continue for the foreseeable future, making the research and study of optimal data-management techniques relevant to this day. In this study, we shall review data structures and algorithms developed in order to design efficient memory systems, built to handle the massive amounts of data being generated every instant.

We begin by identifying the main issues and motivations for developing efficient memory systems. Specifically, those dealing with the management of large OLTP databases. On the same note, it becomes necessary to capture such issues and abstract them into a coherent representation. The first of such, the disk access model, is a classical approach to model interactions between memory systems. Referring to this model allows us to define quality metrics that guide subsequent analysis. To follow, we shall define the core concept of this study, which consists in employing external-memory index structures as means to minimize secondary storage latency and improve overall database throughput. Owing to that idea, follows a detailed description for a set of index data structures, along with their respective asymptotic performance analysis. The aforementioned structures include conventional dictionaries such as hash-based indices and B-trees, the most prominent and widely used index structure to this day. Whereas these structures are well known and have been

optimized throughout decades of use, research points out that newer techniques may offer improved performance and adaptability.

Having understood the limitations and obstacles associated with implementing structures within the disk access model, we shall review a relatively modern scheme: the cache-oblivious model. The cache-oblivious model provides a vastly different approach for analysis and implementation of database engines. Consequently, the next stage involves studying structures within this model such as the cache-oblivious B-tree and the cache-oblivious lookahead array, with the latter being a strong contender to B-trees' dominance. To conclude the study, we perform an experimental comparison of the performance of B-trees against hash-based indexing and the cache-oblivious lookahead array. The B-Tree is chosen as the basis for contrast due to its pervasiveness in database management systems. The actual comparison parameters involve testing both search and update throughput for each structure, along with the disk space costs associated with large scale databases. The implementations of such experiments consist on executing performance benchmarks on real database systems, so as to put the analytic results into a real life context. Thus, providing verifiable knowledge to database users who seek to improve their systems' performance.

PART 1: CACHE-AWARE STRUCTURES AND ALGORITHMS

In order to compare different strategies and techniques to develop optimal memory systems, we must define a context for both theoretical and experimental analysis. We begin by exploring the disk access model, also known as the I/O model or external-memory model. This model was developed many decades ago but it is still relevant to this day.

1. Disk Access Model

The disk access model, also known as the external memory model and the I/O model, was introduced by Alok Aggarwal and Jeff Vitter in 1988. The *DAM* simplifies the memory hierarchy to just two levels. The CPU interacts with a fast memory layer called “cache”. Cache is then connected to a secondary storage called “disk” by a low bandwidth channel. On one hand, cache is finite and bounded in terms its size M. On the other hand, disk is infinitely large. The way both cache and disk exchange information is by transferring sets of records, called blocks. Both cache and disk, are divided into blocks of size B. Therefore, cache contains M/B blocks. Cache can perform memory operations on its internal blocks infinitely fast (0 cost). Nonetheless, transferring 1 block of data between memory and disk costs 1 time unit, equivalent to performing one I/O operation. A single I/O operation can only transfer a single contiguous block at a time, relative to the disk cursor's current position (Demaine, 2012).

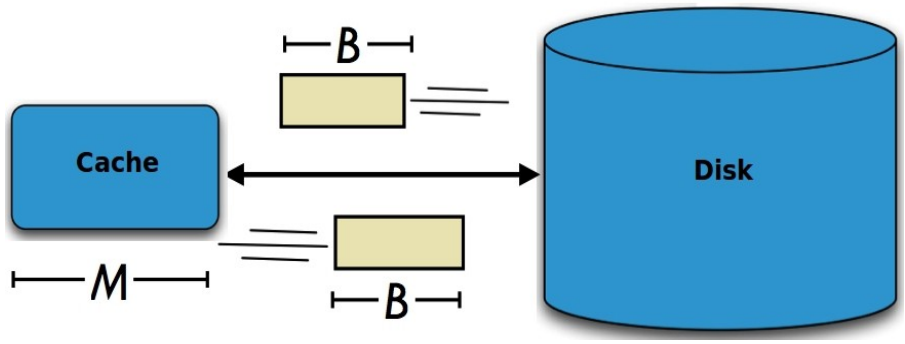

Figure 1: Disk Access Model (Bender & Kuszmaul, 2012)

In order to analyze problems within this model, let us define the appropriate notation:

Quantity	Meaning
B	Number of records per block
M	Number of records stored in cache
N	Number of records stored in disk

Figure 2: Notation for Analysis

For the sake of notational convenience, upper-case letters denote a determined quantity of records, while lower-case letters denote the corresponding number of blocks containing said records. To illustrate, $m = M/B$ is the number of blocks that fit into cache and $n = N/B$ is the number of blocks stored in disk.

2. Index Data Structures

Database management systems have the challenge to organize blocks of data residing in disk efficiently, so as to minimize the number of I/O operations required to perform a determined operation. Particularly, database engines must implement methods that enable to minimize both time and space complexity for data retrieval, insertion and deletion. In order to address this issue, researchers have devised several ways to build index files and their respective index entries, by analytically modeling them with data structures. We shall

refer to these data structures as index structures. Index structures, and the subsequent analysis of their asymptotic performance, are the most important resource researchers have in order to abstract and break-down I/O problems into comparable, measurable terms (Ramakrishnan & Gehrke, 2003).

To illustrate the impact of index structures, let's analyze a scenario where index structures haven't been implemented. Consider the use case where we want to perform a query on a specific data field, with a particular search-key value. The only way available to identify the first such entity that satisfies the query is by doing a binary search on the complete file containing the records, yielding $O(\log_2(FileSize))$ I/Os. This bound would be optimal for main-memory. However, recall we are attempting to improve on the cache-disk interface. In addition, remember that the database engine would have to maintain a sorted file to store records, which in turn presents additional I/O costs and possibly a greater space overhead to handle insert, and delete operations.

As evidenced, simply keeping a sorted file to manage efficient I/O operations is not good enough. In practice, severe performance limitations arouse when taking into account applications that require operating on large amounts of data. Overall, the importance of index structures for optimization of data retrieval operations cannot be understated. It follows the necessity to seek improvements by looking at some of the most relevant structures on this field (Graefe & Larson, 2001).

3. Hash-based Indexing

Introduction.

The first structure we shall review is based on encoding search-key values using hashing methods. Generally speaking, hashing is a technique used to perform mappings from a set of

keys to a set of values. In this context, we define the term bucket, which denotes a unit of storage that can store one or more data records. In practical terms, a bucket plays a role similar to a block or page of records. Hash-based indexing relies on mapping the set of all possible search keys-values with a set of bucket addresses by means of a hash function. In modern times, hash-based indexing is not used extensively because it is outperformed by tree-based structures in update operations and range searches, which will be discussed later. However, hashing leads to very efficient equality searches, making it particularly useful in the implementation of certain relational operations such as joins (Ramakrishnan & Gehrke, 2003).

Hashing can be used to serve two purposes. A hash file organization, which returns the address of the disk block containing a desired record by computing the hash function on the search-key value of said record. The other common application is to generate a hash index structure, which organizes search-keys into a hash file structure. Specifically, a hash index is built by applying a hash function on a search-key value, which returns a an identifier that enables to quickly retrieve a bucket. At the same time, the bucket itself stores the appropriate search-key and its associated pointers to the data records it contains (Silberchatz et. al., 2010). We shall focus only on hash index organizations in our discussion.

We shall review two hashing schemes. The first one static hashing, which suffers from scalability issues. Followed by the extendible hashing dictionary, a structure that supports efficient insertion and deletion with the use of an additional directory.

Analysis.

Formally, let K denote the set of all search-key values. Similarly, let A denote the set of all bucket addresses stored in disk. The hash function, denoted as h is a function from

K to A. The worst possible hash function would map K entirely into the same bucket. This would result in having to scan the whole record collection on any given lookup query. Therefore, ideally it is desired h assigns search-key values to buckets in such a way that:

1. The distribution is uniform. h Assigns each bucket the same number of search-key values from the set of all possible search-key values (Silberchatz et. al., 2010).
2. The distribution is random. The hashed value will not be correlated to any visible ordering on the search-key values. Thus, resulting that in average, each bucket would have the same number of assigned values (Silberchatz et. al., 2010).

In terms of performance, the goal of external-memory hashing techniques is to achieve $O(1)$ I/Os per lookup, which is possible even in the worst case when loading a complete bucket into cache (Moshkovitz & Tidor, 2012). In an ideal scenario, it would take $O(1)$ I/O's per insert and delete operation and a $O(N)$ space overhead amortized. The worst case would be $O(B)$ for updates, resulting from having to completely traverse a bucket to locate the desired record. The biggest challenge in designing such a hash-based index is to enable it to handle widely varying values of N (Vitter, 2008). However, we must take into account that the structure can only return individual data records at a time. The inability to return complete blocks is because records within each bucket are not sorted. Thereafter, such bounds apply only for single record operations. Consider performing a range search, then we would produce $O(n)$ I/O's, with an additional overhead for sorting the returned values for most practical scenarios. This indicates that hash-based algorithms are best suited

for equality searches and individual record updates (Ramakrishnan & Gehrke, 2003).

Static Hashing.

The basic method to construct a hash-based index is to use static hashing. This structure provides simple algorithms to perform search, insertion and deletion operations. However, issues arise when bucket overflows occur. The general search algorithm on a search-key value K_i , consists in computing $h(K_i)$, which returns the address of the bucket to be traversed. Once we locate the appropriate bucket, we scan sequentially every record within the bucket that matches the given search-key value. In order to insert a new index entry with search-key value K_i , we compute $h(K_i)$ and use its output as the address of the bucket to store the index entry. For now, assume there is sufficient space in the bucket to perform the operation. Similarly, deletion of an index entry simply consists in locating the appropriate bucket and consequently removing the item that matches K_i from the bucket (Silberchatz et. al., 2010).

Up to this point, we have only considered that during the insertion of a new record, the respective destination bucket has enough available space to perform the operation. Nonetheless, this situation does not occur all the time, leading to a bucket overflow. The reasons for bucket overflows occurring are insufficient buckets (Number of buckets < Number of records / Records per bucket), and skewing. Skewing refers to the situation where a bucket overflows due to being assigned more records than other buckets. Skewing is often a result of too many records having the same search-key value, or the use of a poor hashing function (Silberchatz et. al., 2010).

Static hashing uses overflow buckets as a policy to handle overflowing. If a new entry is to be inserted on a bucket b and b is already full, the algorithm creates an additional bucket,

in which the new entry is stored. If the overflow bucket becomes full yet again, an additional overflow bucket is created. Overflow buckets associated to b are chained together as part of a linked list. This concept is referred to as overflow chaining. The search algorithm of a search-key value remains the same, with the additional scanning of the overflow chain associated to a bucket. Other policies to deal with bucket overflowing include bucket probing and computing additional hashing functions (Silberchatz et. al., 2010).

The biggest drawback of implementing a static hash index as an efficient external-memory data structure is having to define the hash function a priori. Furthermore, it cannot be changed easily thereafter if the associated indexed record file grows or shrinks. This implies that, insert and delete operations are very likely to suffer delays associated with the overhead of keeping unbalanced pointers to overflow chains. Considering h maps search-key values to A, space is wasted if A is designed to be large enough to handle the record collection's growth. Alternatively, if A is not sufficiently large, bucket overflowing becomes more frequent, affecting performance as overflow chains grow larger (Silberchatz et. al., 2010).

Extendible Hashing.

A different approach to the simple static hashing structure is a more dynamic scheme called extendible hashing. A key feature of this structure is having an additional directory in the shape of an array. Each directory entry contains a pointer to a bucket where records are stored. The advantage of extendible hashing over directory-less methods is being able to perform efficient operations for multiple values of N, while minimizing waste in terms of space usage due to bucket overflowing (Fagin et. al., 1979). However, considering that for large databases, the directory is stored in disk, every operation takes an additional I/O to access the directory itself (Vitter, 2008).

The directory consist of a table of 2^d pointers, for a given $d \geq 0$. Each record address r_i is assigned to the table location address that matches the d least significant bits of $h(r_i)$. As such, we define d as the global depth of the directory. The global depth value is invariantly set to be the smallest value for which every bucket associated to its respective table entry holds at most B records. Nonetheless, it is possible that several table locations have fewer than B assigned records. In order to minimize space overhead, such table locations may share the same bucket for storing their items. A table location is set to share its corresponding bucket with all the other locations that share the same k least significant bits in their respective addresses. A local depth k , is associated to each table location in the directory. The value of k is selected to be the smallest possible value, as long as that the grouped records fit into a single bucket (Vitter, 2008).

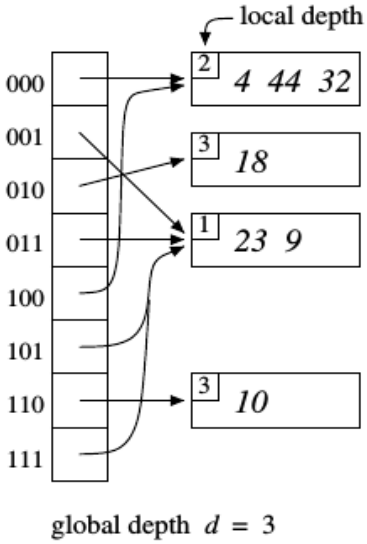

Figure 3: Extendible Hashing Structure with d=3 and k=2 (Vitter, 2008)

The insertion algorithm is identical to static hashing, with the benefit of not having to deal with overflow buckets, but the added overhead related to traversing the additional directory. When a bucket overflows, the global and local depths are recomputed so that the invariants on k and d hold once again. The scenario where a bucket has overflowed,

but k is smaller than the global depth, the bucket is split into two, with each having a separate pointer and location in the directory table and a local depth value of $k+1$. If the value of k becomes greater than d during an insertion, the global depth must be increased by 1 and consequently, the directory is doubled in size, in addition to the aforementioned splitting process. The worst case for insertion occurs when the directory's size must be increased, which yields $O(n)$ I/Os. The deletion algorithm simply follows a reversed insertion algorithm, keeping the invariants on the global and local depth (Vitter, 2008).

Summary of I/O Complexity Bounds.

Measure	Average	Worst
Access Time	$O(1)$	$O(1)$
Insertion Time	$O(1)$	$O(n)$
Deletion Time	$O(1)$	$O(n)$
Space Overhead	$O(N)$	$O(N)$

Figure 4: Time and Space Complexity for Extendible Hashing Operations

4. B-tree

Introduction.

The B-tree is a dynamic, flexible data structure that improves radically on most fronts when compared to hash-based indexing. The B-tree is the most widely used data structure among those that support efficient insertion and deletion. Being the most popular index structure, it is implemented by most database engines, including: InnoDB, SQLite, PostgreSQL, WiredTiger, among others. It is also part of operating systems such as: Mac OS HFS/HFS+, Windows NTFS, Linux ext4, among others. Considering the B-tree was invented several decades ago (1972), modern implementations have been heavily optimized and usually

differ from its basic definition. This is due to some of its implicit limitations, which will be discussed ahead. The main characteristics of a B-tree include:

- Insert and delete operations keep the tree balanced and ordered invariantly (Schwarz, 2016).
- Searching for a block requires a traversal from the root to the appropriate leaf. The time to perform this operation is constant regardless of the selected entry since a B-tree is balanced as its height (Schwarz, 2016).

A B-tree is a balanced tree, in which every path from the root of the tree to each of the leaves has the same length. In other words, all the leaves are always at the same depth. The internal nodes of the tree contain a set of pivot values that direct the search of a given value, while the leaf nodes contain the actual blocks of data. Every B-tree has a branching factor, or degree, which determines the shape and layout of the tree. Particularly, the following rules apply for every non-leaf node bar the root node:

- $\frac{B}{2} \leq C \leq B$. Where C stands for number of children nodes.
- $\frac{B}{2} - 1 \leq K \leq B - 1$. Where K stands for number of pivots within a node. In other words, the number of pivots per node is always one less than the number of children.

For the root node, The same inequality for the upper bounds of number of children is enforced. However for the lower bound for both children and keys is greater than 1 (Ramakrishnan & Gehrke, 2003).

Analysis.

The B-tree is the most popular index dictionary available for the DAM. The most common implementations are designed so that the data associated with a single node in the tree fits into a single block. This means that every leaf in the B-tree stores between B/2 and B data records. In terms of space requirements, B-trees use a linear space of $O(n)$.

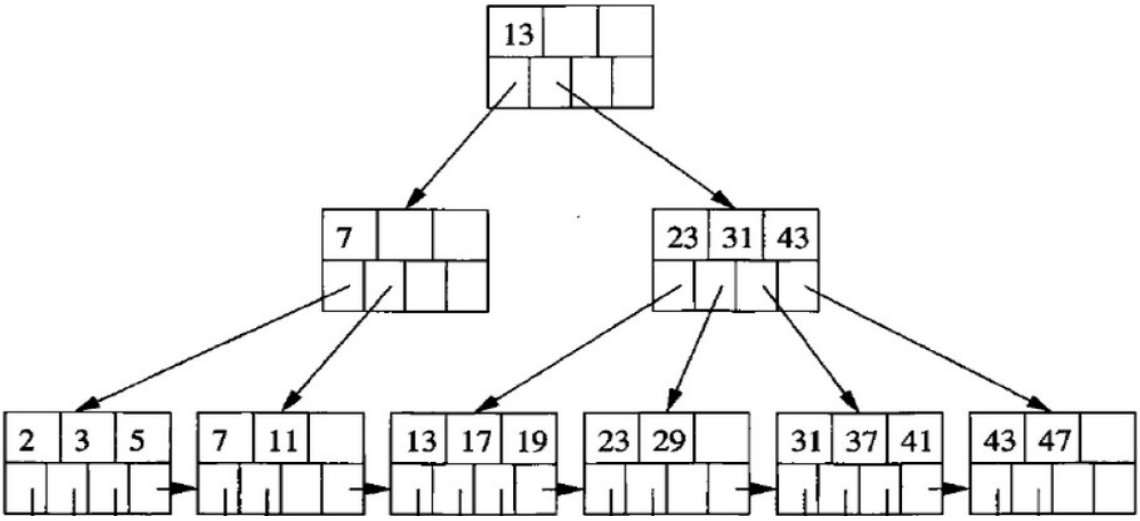

Figure 5: B- Tree with degree 4

The search algorithm for a search-key *x* from the tree consists in loading the block of keys at the root, locating the pair of keys that encapsulate *x*'s value and then searching the corresponding sub-tree recursively. In order to take advantage of the B-tree's properties, we may set the branching factor to be equal to B , more precisely $\Theta(B)$. This implies that by searching each level of the tree, the range of candidate positions for *x* drops by a factor of $\Theta(B)$, which is at least B/2, except possibly for the root level. Thus, the number of required I/Os for searching is $O(\log_B(N))$. This makes B-trees excellent for range selection queries since they simply require finding a single index key that matches the given range and then returning the corresponding sub-tree (Demaine et. al., 2015).

For insertion of a new record *x*, we would have to locate the appropriate leaf l to store its value. If l has available room, store *x* and return. Else, split l into two new leaves. Each

new leaf stores half of the elements of l . This split triggers the insertion of a new key node at the parent of l , which in turn is handled the same way as leaf nodes, propagating up to the root node if necessary. During propagation, the tree grows in height, but keeps itself balanced. If possible, propagation can be limited by having incomplete nodes borrowing elements from full sibling nodes. The total time to complete the insertion operation is $O(\log_B(N))$. The deletion algorithm is exactly opposite to insertion. First remove the target leaf. Then, instead of splitting full leaves, the algorithm would combine incomplete leaves and propagate if necessary. This results in deletion requiring the same number of I/O operations (Erickson, 2002).

In order use the B-tree within the context of external memory, we have considered an online setting. In other words, we are performing operations on single data records one at a time. The issue is that, in order to insert or delete a new record, the standard algorithm spends one I/O loading the root's index node, then finding position of a single element recursively. This results in sub optimal performance of the B-tree for batched problems like computational geometry and graph related problems. However, in a different, in-memory context, online structures often adapt well to these scenarios. This brings up the need to perform batch operations on external memory in order to take advantage of its size and inherent parallelism (Vitter, 2008).

Buffer Tree.

The solution for the batched-update problem proposed by Lars Arge, named *Buffer tree,* consists in adapting the B-tree into a modified tree with a branching factor of $\Theta(m)$, instead of the usual $\Theta(B)$ (Arge, 2003). The buffer tree's key distinguishing feature is that each index node has an associated buffer that can store $\Theta(m)$ blocks of items. Records

kept within a node's buffer are pushed down to the children once its filled, or a flush operation is called. Doing a full buffer flush operation on a node requires $\Theta(m)$ I/Os, reducing the cost of distributing possible M items on, at most m children . This method allows each item to be flushed down with an amortized cost of $O(m/M)=O(1/B)$ per level (Erickson, 2002). Thus resulting in $O(\frac{\log_m(N)}{B})$ amortized I/Os for deletion and insertion. In sum, the buffer tree provides means for optimal offline updates, which is useful whenever we intend to perform delayed queries, sorting, and implementing priority queues (Vitter, 2008).

Summary of I/O Complexity Bounds.

Measure	Average	Worst
Access Time	$O(\frac{\log(N)}{\log(B)})$	$O(\frac{\log(N)}{\log(B)})$
Insertion Time	$O(\frac{\log(N)}{\log(B)})$	$O(\frac{\log(N)}{\log(B)})$
Deletion Time	$O(\frac{\log(N)}{\log(B)})$	$O(\frac{\log(N)}{\log(B)})$
Space Overhead	$O(N)$	$O(N)$
Batched Update	$O(\frac{\log_m(N)}{B})$	$O(\log_B(N))$

Figure 6: Time and Space Complexity for B-Tree Operations

5. Experimental Results: Comparing the B-Tree vs. Hash-based indexing

In order to test the throughput of the analyzed data structures, let us first define what SQL operations are more relevant to the context of large-databases performance. As we reviewed both B-trees and hash-based indices, it becomes apparent that insertion and deletion procedures are equivalent in terms of I/O cost. In similar fashion, update operations

are redundant since they involve a search-write-delete pattern. Therefore, the primary focus are SELECT and INSERT operations. Moreover, the experiments have been designed to perform the aforementioned operations with randomly varying search-key values, so as to emulate a real working environment where data records possess diverse column values. In addition to the standard database operations, another concern regarding database's scalability is database disk usage. Thus, the experiments shall include a comparison for their space overhead.

SELECT statements may involve either equality searches or range searches. Nonetheless, it is only relevant to measure equality searching SELECT statements, since hash-based indices do not support range searches. Additionally, range searches would require large-sized test samples to be analyzed since they are heavily affected by the number of returned records. This fact, in combination with up to two random bound values that limit range searching, make this operation yield high entropy results. Thus, discarding range search testing becomes a sensible assumption for the scope of this study. It is imperative to mention that the techniques studied are based on a OLTP environment, and therefore, we define transaction throughput as our primary measure for performance.

The tests consist in OLTP benchmarks on the PostgreSQL 9.5 engine. The reasoning behind this decision is because PostgreSQL is one of the few remaining engines that still support hash-based indexing, while implementing modern, state of the art B-tree indexes.

The machine used to run the benchmarks was a Ubuntu 16.04 LTS virtual machine with a 4 Core 3.5GHz Intel processor, 4 GB DDR3 SDRAM and 100GB storage on a 5400 RPM SATA drive. The actual tests were executed using pgbench, an open source benchmark tool included in PostgreSQL.

Assumptions.

In this scenario, the focal point is to weight the transaction throughput in terms of the number of rows present in the database at the beginning of the operation. In this way, it is possible to obtain a clear picture on the database's performance for a determined number of records stored on external-memory. For the tests, we attempt to perform comparisons based on database system locality. In other words, we claim that by fixing the database engine's settings, except for the index of choice, the tests results will reflect the impact of said choice in terms of relative transaction throughput.

It is also worth noting that the analysis section presents results in terms of I/O operations count instead of SQL transaction throughput. These measures are clearly correlated, yet not equal. Therefore, we do not expect to generate data that reflects the actual asymptotic bounds, but rather a description of the throughput vs. database size trend. Thus, we may also assume that the SQL schema used to perform the tests, has little impact on the gathering and analysis of results, as long as they permit creating the desired indexes on them and the database contains enough data rows so that they cannot be fit into main memory.

For this specific comparison, we use a 8 column table, featuring 3 secondary indices, in addition to the primary key index. The benchmarks consisted on a set of 30, 90-second tests continuously running on the operation that was tested, be it SELECT and INSERT, for each of the tested indices.

Results.

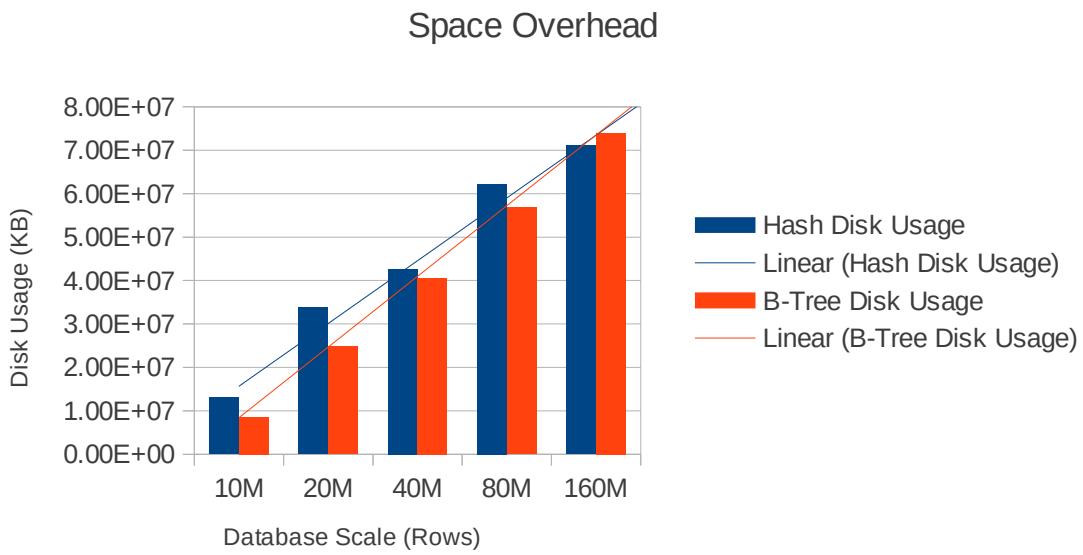

Figure 7: B-tree vs. hash index disk usage

The experiments for this comparison consist on doing benchmarks on five increasing database row count scales. Thus, the row count scales chosen were: 10, 20, 40, 80 and 160 million rows, with the latter filling the machine's secondary storage by 86.16%. Figure 7 displays a chart for the space requirements for each index type. However, this behavior is not consistent for different row counts, making it unappropriated to claim hash-based index require less disk blocks. Although we can recognize a linear behavior for the B-tree's ($O(N)$) space overhead, the trend line for the hash-based index is not as good a fit and may not correlate strongly to its analytic $O(N)$ bound. Thus, we may not point to a preferred index alternative for this parameter.

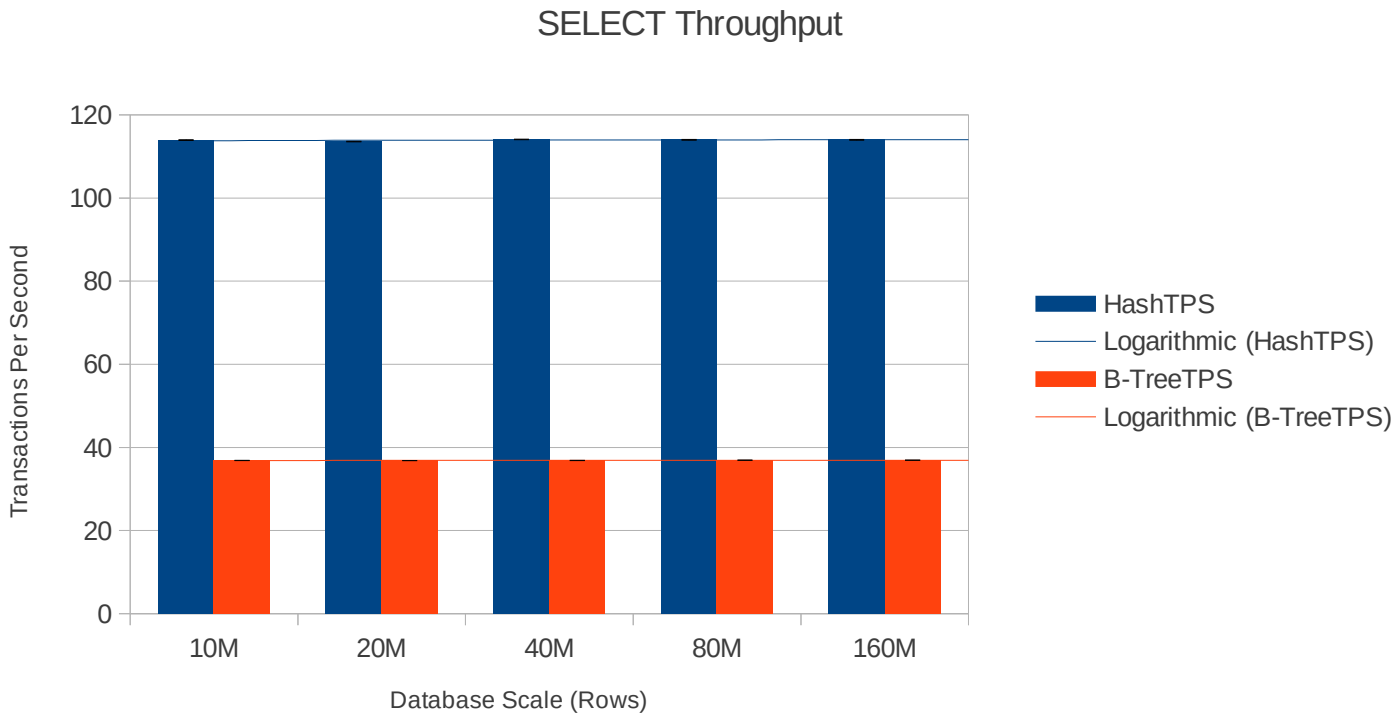

Figure 8: Transactions-per-second averages for SELECT

Figure 8 shows the transaction throughput for SELECT queries performed on the B-tree index, alongside the throughput the same operation on the hash-based index on different database size scales. It becomes apparent the trend points to hash-based index outperforming B-Tree indices considerably. For the largest value of $N = 160 * 10^6$ rows, the hash-based index was 3.086 times faster. The average the improvement offered by the hash index in terms of database size was a factor of 3.088. This fits nicely alongside the previous analysis, where it was expected that hash indices ($O(1)$) would outperform B-trees ($O(\log_B(N))$) for searches in terms of total I/O operations. Nonetheless, the logarithmic tendency lines suggest that in both cases, row count doesn't impact SELECT operations considerably, resembling more of a constant behavior $O(1)$. The tabulated values for the average and standard deviation for each of the database scales is presented in Figure 9:

Structure	Scale (Rows)	Average TPS	Standard Dev.

B-Tree	10M	36.86	0.76
B-Tree	20M	36.83	0.73
B-Tree	40M	36.86	0.45
B-Tree	80M	36.92	0.64
B-Tree	160M	36.93	0.54
Hash Index	10M	113.91	0.13
Hash Index	20M	113.58	0.15
Hash Index	40M	114.11	0.16
Hash Index	80M	114.02	0.12
Hash Index	160M	114	0.11

Figure 9: Average and Standard Deviation values for SELECT

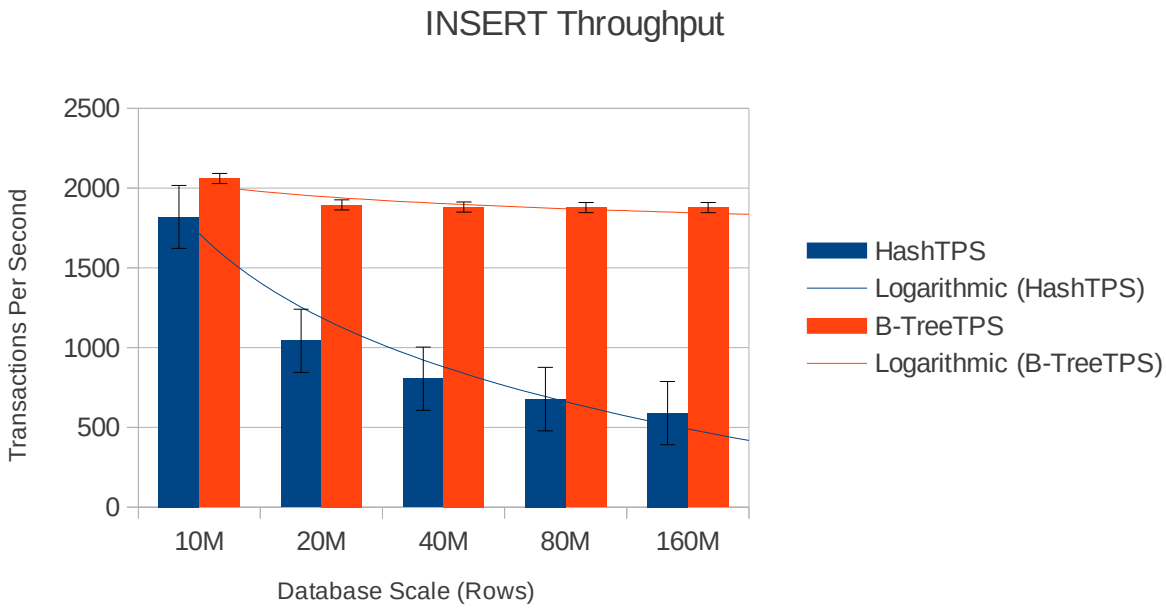

Figure 10: Transactions-per-second averages INSERT

Figure 10 shows the transaction throughput for INSERT queries performed on the B-tree index, alongside the throughput the same operation on the hash-based index on different database size scales. The logarithmic tendency line added in the figure suggests that as the row count grows, the hash-based index's performance consistently receives a greater impact. Contrary to the results observed for SELECT, it is clear that B-trees offer greater optimization for writing records into disk. For the largest row count of $N = 160 * 10^6$, the

B-tree index was 3.18 times faster. For the smallest row count of $N = 10 * 10^6$, the improvement scales down to a factor of 1.132. In average, the B-tree performed 2.2477 times more transactions than the hash-based index during the same interval of time and row count. This result is expected as we recall that in asymptotic analysis, the worst case for hash indices inserts is $O(N)$, which is massive in comparison to the $O(\log_B(N))$ I/Os for B-trees as the number of records in the index grows. Analysis could explain that the worst-case for hash-based indices insertion occurs more frequently during insert-heavy scenarios. Considering that as more data is inserted into the extendible hashing index's directory, the need to double the index's directory would become more frequent. The tabulated values for the average and standard deviation for each of the databases scale is presented in Figure 11:

Structure	Scale (Rows)	Average TPS	Standard Dev.
B-Tree	10M	2060.14	77.06
B-Tree	20M	1894.3	49.05
B-Tree	40M	1880.96	54.83
B-Tree	80M	1878.72	58.32
B-Tree	160M	1877.46	57.05
Hash Index	10M	1819.47	256.18
Hash Index	20M	1042.86	168.36
Hash Index	40M	804.88	54.67
Hash Index	80M	677.33	35.63
Hash Index	160M	590.47	25.38

Figure 11: Average and Standard Deviation values for INSERT

PART 2: CACHE-OBLIVIOUS DATA STRUCTURES

6. Cache-oblivious Model

So far, we have only analyzed structures and algorithms within the context of the traditional DAM model. We may call all previously analyzed structures cache-aware because the DAM model assumes that the cache layout is known. This incurs into difficulties when designing and implementing actual database engines due to the wide array of possible values for memory size and block length in different memory hierarchy levels and systems altogether. For instance, in order to use standard B-Trees to efficiently manipulate data across a multi-leveled hierarchy, it would require tuning the structure so that it becomes optimal for every level interface, for every machine architecture. In some cases it is even impossible in heterogeneous computing environments such as across a network. However, it is not entirely necessary to assume we know the memory layout. The cache-oblivious memory model is a solution that provides more flexibility for implementing high performance database engines, while staying close to the lower bounds of required memory transfers for cache-aware algorithms (Prokop, 1999).

The key difference from the DAM based analysis, is that cache-oblivious algorithms don't know the values of B and M. This does not mean that the actual machine that runs the algorithm is agnostic about B or M, it would still have a fixed memory size and transfer blocks of B data records during a memory transfer. Nonetheless, algorithms cannot make decisions based on block length, including reading or writing a particular block. We also assume that the strategy for block replacement is optimal, which can be approximated by operating systems using a least-recently used replacement policy, or even a first-in-first-out

policy. This assumptions make cache-oblivious algorithms resemble main memory algorithms in terms of their implementation. A good data structure for the cache-oblivious model is very valuable, since it becomes simultaneously good for every possible cache layout. Specifically, if an algorithm performs optimally on the cache-oblivious model, the algorithm performs asymptotically optimally on any unknown multilevel memory hierarchy (Demaine, 2002).

7. Cache-oblivious B Tree

Introduction.

The first cache-oblivious problem we shall study is a cache-oblivious version of the well known B- tree. However, the solution consists in the combination of two concepts: a cache-oblivious static binary search tree and a packed memory structure (Kasheff, 2004). This structure was developed around 2002, but real life implementations and usage has not reflected the great theoretical advantages it offers. Nonetheless, understanding how the most popular data structure for external-memory can be adapted to a cache-oblivious context could lead to the improvement of heterogeneous memory systems in the future.

Static Cache-Oblivious Binary Search Tree.

Similar to the usual B-tree's implementation, the binary search tree (BST) is laid out as an array of contiguous blocks of data following a breadth-first search order. We also assume that the tree is full and balanced. However, the BST follows a cache-oblivious layout, known as Van Emde Boas layout (Kasheff, 2004). Assume a BST with N items and height *h*, h being a power of 2 for simplicity. Consider splitting the tree so that each sub-tree has height h/2. This leaves a top half of the tree, sharing the same root as the original tree, holding $\sqrt{(N)}$

items. The division leaves a bottom half that contains $\sqrt{(N)}$ sub-trees, each containing approximately $\sqrt{(N)}$ nodes. The Van Emde Boas order consists in laying out the top half of the tree recursively, followed by recursively laying out each of the $\sqrt{(N)}$ bottom-half sub-trees (Moshkovits & Tidor, 2012).

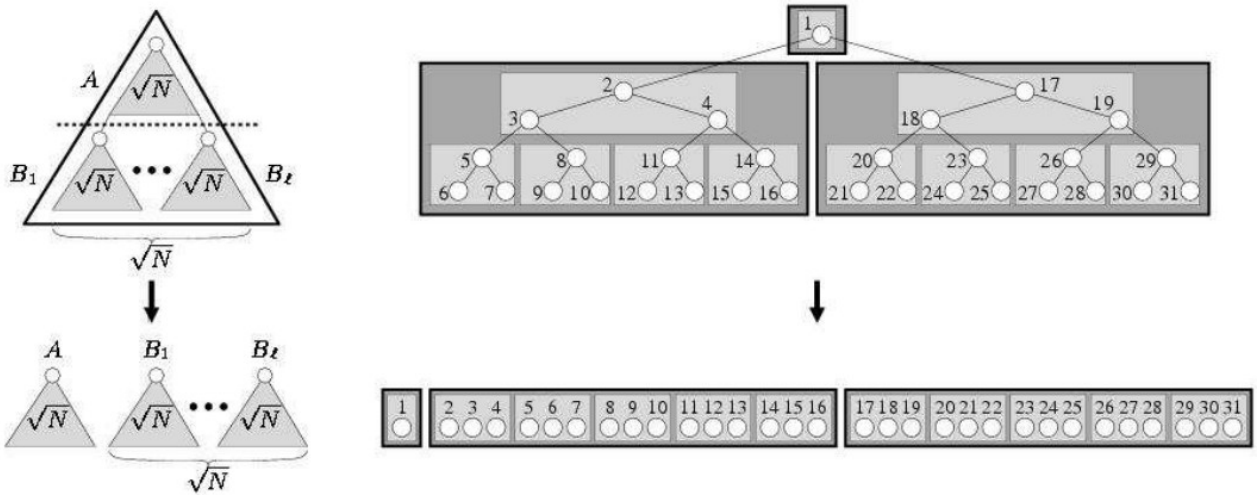

Figure 12: Van Emde Boas layout for a BST tree of height 5 (Kasheff, 2004)

In the scenario displayed in the left of Figure 12, the entire A sub-tree would be recursively laid out first, followed by each of the B_i sub-trees consecutively. To handle the case where *h* is not a power of 2, such as the one in Figure 12, the bottom-half of the tree is chosen such that its height becomes a power of 2. The Van Emde Boas order allows the BST tree to be laid out in disk in a recursive manner, this property is what makes it work as a cache-oblivious structure, since it is possible to adjust the level of recursion to match cache's block size B. In practice, the idea is to continue dividing each sub-tree until each one has between $\sqrt{(B)}$ and B elements and height of $\frac{\log_2(B)}{2}$ and $\log_2(B)$ respectively. This implies that each sub-tree requires less or equal than 2 disk blocks to be stored and the number of levels searched per memory transfer is at least $O(\log_2(B))$. Thus, by calculating the number of sub-trees that are traversed in root-to-node order, we can now

the lower bound for searching an element. In this way we would require at most

$O(2\frac{\log_2(N)}{1/2*\log_2(B)})=O(\log_B(N))$ memory transfers, which is asymptotically optimal

(Bender, et. al., 2005).

A massive issue with this structure is that it is static because it doesn't support updates in a trivial fashion. The challenge is to maintain a dynamic Van Emde Boas layout of a BST. On one hand, the solution would have to achieve locality of reference by packing index entries densely into disk. At the same time, it should support future insertions by having enough extra space between entries. The solution that follows consists in using a packed memory structure (Demaine, 2012).

Packed memory structure.

This structure consists in an array containing N elements. Inserting elements into arrays is a well studied problem which can be approached similarly to the ordered file maintenance problem. In order to insert a new element into this array, it is necessary to move elements in the array and handle split and merge operations. Therefore, a good strategy is to have a loosely packed structure, with empty slots among elements of the array. The number of

memory transfers it takes to perform an update into this structure is $O(\frac{\log_2^2(N)}{B})$

amortized (Demaine, 2012).

If we combine the static binary search tree alongside the packed memory structure, it is possible to get a cache-oblivious data structure that is efficient on both searches and updates. The combined data structure is made up of the BST, with its corresponding leaves stored as the packed memory structure which keeps the Van Emde Boas layout. The

searching algorithm is the same as the BST, keeping the same bounds. In order to insert a new element, it is required to first locate the place where it should be stored ($O(\log_B(N))$). Then, we update the packed memory array in $O(\frac{\log_2^2(N)}{B})$. The next step is to propagate the changes into the BST tree in order to preserve the Van Emde Boas layout. This procedure is equivalent to performing a post-order traversal of the tree, which changes the values of the BST nodes, but does not affect the tree's layout. The result is a total $O(\frac{\log_2^2(N)}{B}+\log_B(N))$ memory transfers for updating the tree (Bender, et. al., 2005).

The insertion bound for the cache-oblivious B-tree differs from the cache-aware version by a factor of $O(\log_2^2(N))$. The way we improve on this bound was by adding an additional level of indirection to the packed memory structure. By splitting the packed memory array into linked sub-arrays of size $\Theta(\log_2(N))$, which work similarly to disk blocks in cache-aware B-trees, it is possible to optimize update operations. Introducing the additional level of indirection affects searches by having to scan a memory block in $\Theta(\frac{\log_2(N)}{B})$ memory transfers. However, this modification helps improve insertions and deletions by a factor of $\log_2(N)$ since the BST tree does not have to keep pointers to each of the packed array elements, but rather to each block. The resulting bounds for searching and insertion, with the added level of indirection are, respectively: $O(\log_B(N)+\frac{\log_2(N)}{B})\approx O(\log_B(N))$ and

$$O\left(\frac{\frac{\log_2^2(N)}{B}}{\log_2(N)}+\frac{\log_2(N)}{B}+\log_B(N)\right)\approx O(\frac{\log_2(N)}{B}+\log_B(N))$$ (Bender, et. al., 2005).

Summary of I/O Complexity Bounds.

Measure	Average	Worst
Access Time	$O(\log_B(N))$	$O(\log_B(N)+\frac{\log_2(N)}{B})$
Insertion Time	$O(\frac{\log_2(N)}{B}+\log_B(N))$	$O(\frac{\log_2(N)}{B}+\log_B(N))$
Deletion Time	$O(\frac{\log_2(N)}{B}+\log_B(N))$	$O(\frac{\log_2(N)}{B}+\log_B(N))$
Space Overhead	$O(N)$	$O(N)$

Figure 13: Time and Space Complexity for Cache-oblivious B Tree Operations

8. Cache-oblivious lookahead array

Introduction.

The following section will describe one of the modern cache-oblivious structures, the cache-oblivious lookahead array (COLA). This structure was developed by researchers from MIT and Stony Brook university (2007), and is intended to optimize insertions, while keeping searching bounds that compete with B-trees.

Basic COLA.

The basic COLA structure consists of $\log_2(N)$ arrays, or levels. The k^{th} array is of size 2^k and is always either completely full or completely empty. The way the COLA is stored is by contiguously laying out each array into memory (Bender et. al., 2007). The COLA follows the following invariants:

1. The k^{th} array contains items if and only if the k^{th} least significant bit of N is equal to 1 (Bender et. al., 2007).

2. Each array contains its items in ascending order by key (Bender et. al., 2007).

In this basic instance of the COLA, searches are very costly, requiring to perform a binary search on each array. Yielding up to $O(\log_2^2(N))$ memory transfers. The insertion algorithm requires having an additional COLA structure, with the same number of arrays as temporary storage (Bender et. al., 2007).

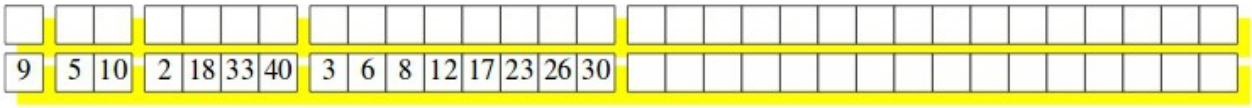

Figure 14: COLA of 5 levels (Kuszmaul, 2010)

In order to keep the invariants defined previously, the new element to be stored must first be put into the first level of the temporary structure. We proceed by merging the elements of the 1st level of each structure together and keep them into the temporary COLA.

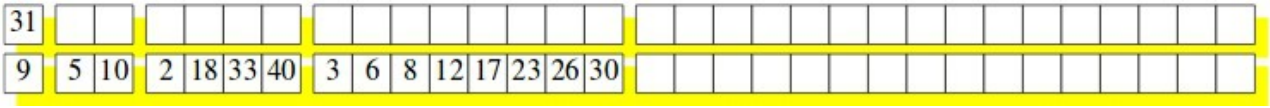

Figure 15: Inserting 31 into the COLA (Kuszmaul, 2010)

The process is repeated until every level is merged together, storing the last merged array into the original COLA.

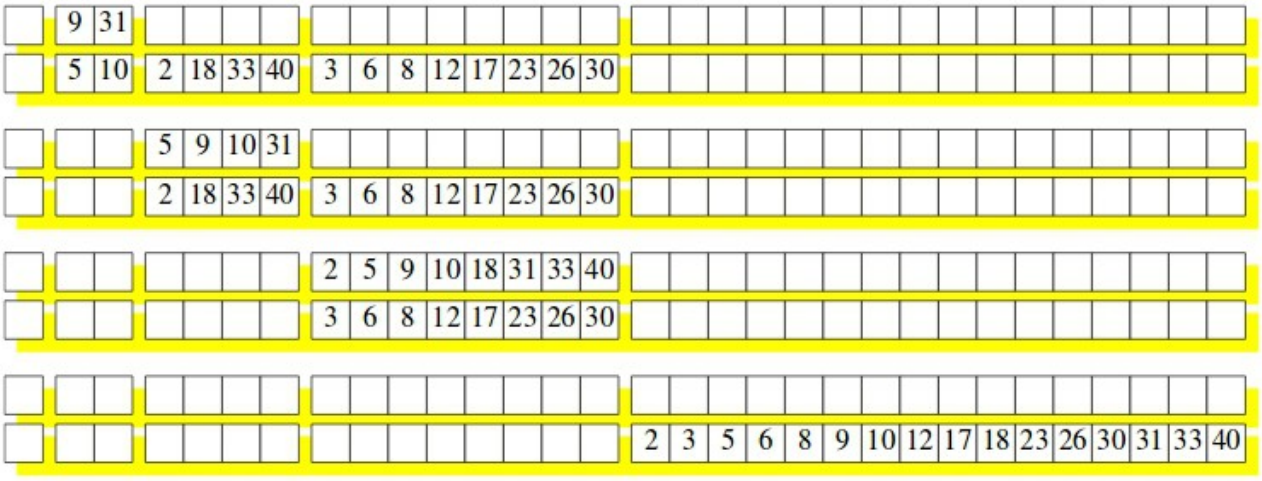

Figure 16: Merging the levels (Kuszmaul, 2010)

Since there are $\log_2(N)$ levels, once an item has been inserted, it has been involved in at most $\log_2(N)$ merges. Consider the worst case where the new element ends up stored in the last level. The first $\log_2(B)$ merged levels require only one memory transfer and can be performed in cache. It follows that the next levels being merged have at least *k* elements, *k* being greater than B. Each level would require $O(k/B)$ memory transfers, equivalent to scanning its respective array. This means that each merging operation requires $O(1/B)$

per item, resulting in an amortized $O(\frac{\log_2(N)}{B})$ number of memory transfers . However, is is possible to have up to $O(N/B)$ merge operations in the worst case, where every element, in every array must be merged. To improve this, we first assume that $M=\Theta(\log_2(N))$. We then perform deamortization by using the cache blocks to store additional information related to each level in the array, containing details about its elements and their likely future destination level. This allows the algorithm to avoid performing unnecessary merges, thus deamortizing the worst case into $O(\log_2(N))$ (Bender et. al., 2007).

Optimizing Searches in the COLA.

Searching in the basic COLA structure is expensive. In order to improve on the $O(\log_2^2(N))$ bound, we use a technique called “fractional cascading”. The general idea is to skip doing a binary search on a determined level by using information from the binary search on the previous level. This is achieved by setting pointers among elements from consecutive levels, called lookahead pointers. We setup this technique beginning in the third level of the structure, of size 8. For each level *i* onward, every eight element in the level is

duplicated into the *i*-1 level alongside a pointer to the original element, the duplicated entries are called “lookahead pointers”. Additionally, in each level, every fourth element is reserved for an additional entry called “duplicate lookahead pointer”, which holds pointers to the nearest lookahead pointers both to its left and right. Thus, incurring into each level holding half of its space for real elements and the remaining half for lookahead pointers (Nelson, 2013).

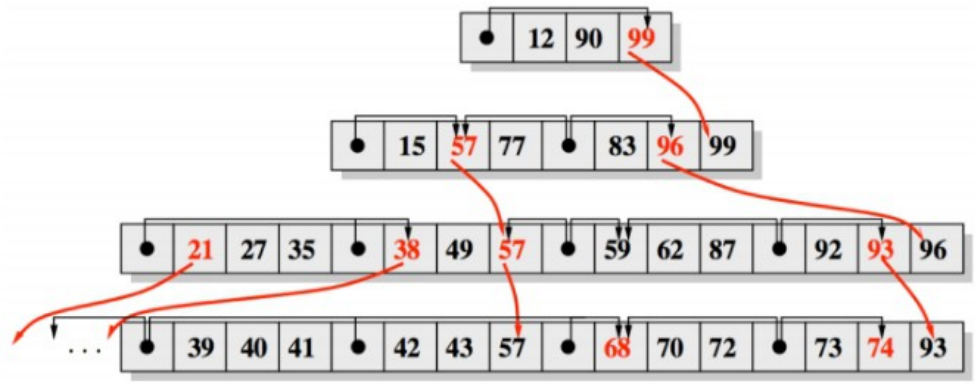

Figure 17: COLA with lookahead pointers (Nelson, 2013)

In order to measure the improvement on searches produced by adding lookahead pointers, we inductively prove its properties for each pair of levels. If we can prove that each level requires $\Theta(1)$ comparisons per level, then it is intuitive that the overall cost for searches is $O(\log_2(N))$. We first claim that a search for a key *r* looks at most eight contiguous items in each level and at the same time, *r* is greater than or equal to the smallest of such items and smaller than or equal to the larger of such items. The claim is inherently true for the first three levels since each has size of at most eight. The claim is also true at the k^{th} level, where searching in this level involves looking at contiguous items with keys $r_1<r_2<\ldots<r_8$ and that $r_1 \leqslant r \leqslant r_8$. Let *l* be such that $r_l<r<r_{l+1}$. The first possibility is that $r_l=r$, then we have successfully found the desired element. For this case, the induction is trivial for all remaining levels. Otherwise, it must be true that $r_l<r$. In this scenario, the algorithm

restricts searching on the next level to the keys that fall between the elements pointed by the two closest lookahead pointers, which hold key values that are the maximum below *r* and minimum above *r*. Thus searching with the lookahead pointers incurs in $O(\log_2(N))$ memory transfers (Bender et. al., 2007).

Cache-aware analysis.

So far we have found bounds that are optimal within the cache-oblivious model. However, it is possible to improve on those bounds by adapting the COLA to cache-aware context.

Specifically, it is possible to achieve $O(\log_B(N))$ for searches and $O(\frac{\log_B(N)}{B})$ for updates (Bender et. al., 2007).

Take the previously discussed COLA, but instead of doubling the array size at level *k* from the size of the array at level *k*-1, we can have it grow by some arbitrary multiplicative factor *g*, called growing factor. The cache-oblivious COLA would have a *g* value of two. In order to reach the desired bounds, we set $g=\Theta(B)$. This means that now there are $O(\log_B(N))$ levels in the structure (Bender et. al., 2007).

For searches, instead of having a lookahead pointer every eight element per level, the cache-aware structure would create a lookahead pointer every $\Theta(B)^{th}$ element. The algorithm would look through $\Theta(B)$ elements per level, instead of $\Theta(1)$. Nonetheless, we tune $\Theta(B)$ so that $\Theta(B)$ elements fit in at most 2 blocks, implying a constant number of I/Os per level. This settings enable us to keep the proof for the regular COLA searching bound, resulting in $O(\log_B(N))$ memory transfers because of the reduced number of levels (Bender et. al., 2007).

For updates, the level of the original structure that is being merged may not be empty, thus

having to merge its elements with the ones stored in previous levels. It is known that the sum of the sizes of the first *k*-1 levels is at least an $\Omega(1/B)$ fraction of the size of the k^{th} level, since each level grows by a factor of $\Theta(B)$. Therefore, a level can handle at most B merges before its items are merged into a higher level in the future. Combining this assertion, with there being $O(\log_B(N))$ levels, it turns out that insertions have an

$O(\frac{\log_B(N)}{B})$ amortized worst case bound (Bender et. al., 2007).

Summary of I/O Complexity Bounds.

Measure	Average	Worst	Cache-aware Average	Cache-aware Worst
Access Time	$O(\log_2(N))$	$O(\log_2(N))$	$O(\log_B(N))$	$O(\log_B(N))$
Insertion Time	$O(\frac{\log_2(N)}{B})$	$O(\log_2(N))$ If $M=\Theta(\log_2(N))$ else $O(N/B)$	$O(\frac{\log_B(N)}{B})$	$O(\log_B(N))$
Deletion Time	$O(\frac{\log_2(N)}{B})$	$O(\log_2(N))$ If $M=\Theta(\log_2(N))$ else $O(N/B)$	$O(\frac{\log_B(N)}{B})$	$O(\log_B(N))$
Space Overhead	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Figure 18: Time and Space Complexity for COLA Operations

9. Experimental Results: Comparing B-Tree vs COLA

On a similar note to the set of experiments on cache-aware structures, testing the throughput of the analyzed data structures, requires to first define what SQL operations are more relevant to the context of large-databases performance. Just like in the B-tree vs. hash experiments, insertion and deletion procedures are equivalent in terms of I/O cost. Consequently, update operations are still omitted from the experiments. Therefore, the primary focus are SELECT and INSERT operations, followed by testing the corresponding

space overhead. Lastly, we maintain the randomness policy for search-key values used in experiments, so as to emulate a real working environment where data records possess diverse column values.

For the following set of experiments, we continue to ignore range searching SELECT statements. In this case, the aforementioned argument stands since range searches would still require excessive test samples in order to visualize consistent patterns.

The tests executed as part of the following experiments consist in OLTP benchmarks on Percona Server 5.7, an open source distribution of the MySQL database management system. Specifically, tests are intended to isolate the InnoDB and TokuDB engines. This choice is based on the fact that TokuDB is the only database engine that supports lookahead arrays (albeit with the commercial name of Fractal Tree Index®), while being integrated into a major database management system. Percona Server is also a great platform for comparison, when taking into account that it also supports InnoDB, which is actually the most popular and one of the most heavily optimized open source implementation of B-tree indexes available.

The machine used to run the benchmarks was the same as the one used for the previous set of experiments. The actual tests were executed using sysbench 0.4.12, a multi-use open source benchmark tool.

Assumptions.

For this comparison, most of the assumptions made for the hash vs B-tree experiments remain. The main focus remains measuring transaction throughput in terms of the database's row count.

For this specific set of experiments, we use a 4 column table, featuring 1 secondary index

and the primary key index. The benchmarks consisted on a set of 60, 90-second tests continuously running on the operation that was tested, be it SELECT and INSERT, on each of the tested database engines.

Results.

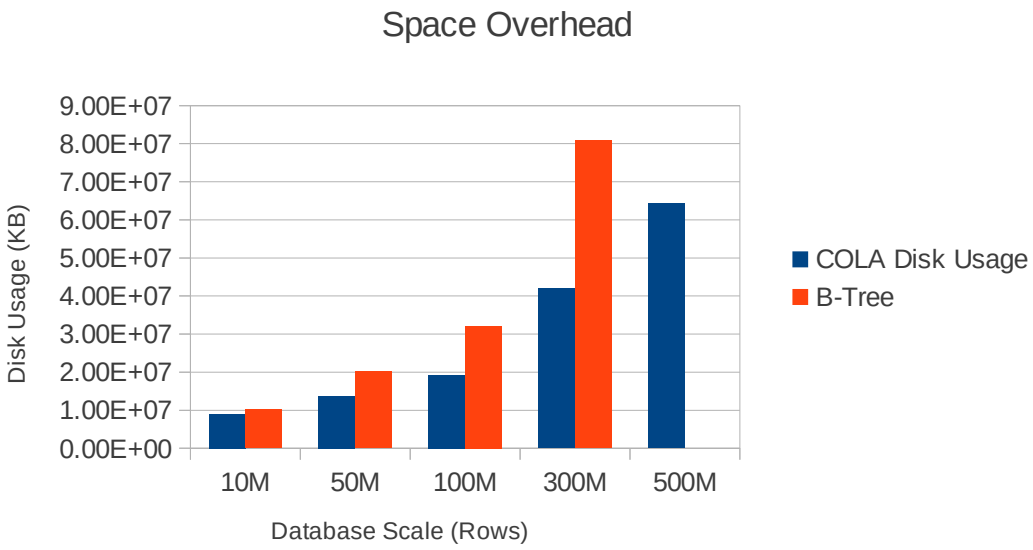

Figure 19: B-tree vs. COLA disk usage

The experiments were executed on five database row count scales. Specifically on: 10, 50, 100, 300 and 500 million rows. However, during experimentation, it turned out that the virtual machine's disk could not fit 500 million records using InnoDB's engine. In actual fact, the 300 million-row sample nearly saturated the virtual machine's secondary storage (94.23% usage). On the other hand, TokuDB indices showed massively greater scalability in terms of disk space usage, managing to store 500 million records with even less space requirements than the InnoDB 300-million-row database. For the highest comparable database size of 300 million records, B-tree indices were outclassed by fractal-tree indices by a factor of 0.519. In average, fractal-tree indices required 0.6439 less disk space than their B-tree counterpart. It is clear that the results would be more complete with more database scales for comparison. One may suggest that a smaller row count value would have solved

this issue, yet this would have hindered other tests' results, which shall be discussed on the INSERT tests' results that follow.

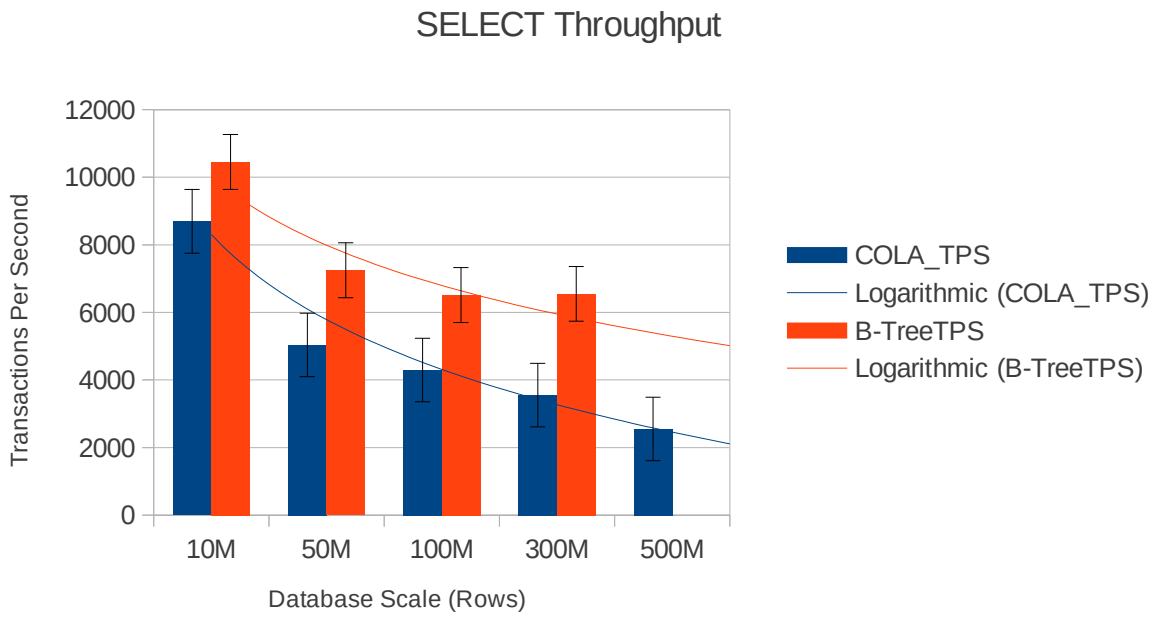

Figure 20: Transaction-per-second averages for SELECT

Figure 20 is a visualization of the average transaction throughput for SELECT queries performed on the fractal-tree index, alongside the throughput of the same operation on the B-tree index on increasing database size scales. Figure 20 suggests that there is a consistent throughput advantage for B-trees, edging over COLA indices for every row count tested. Nonetheless, both indices display a clear trend to reduce transaction throughput as database sizes increase. For the largest value of $N = 300 * 10^6$ rows, B-trees were 1.256 times faster. The average the improvement offered by B-trees in terms of database size was a factor of 1.384. Even though we visualize a clear trend, there is no clear evidence as to why this occurs. From an analytical standpoint, it could be argued that TokuDB implements a cache-oblivious lookahead array ($O(\log_2(N))$) rather than its more efficient cache-aware implementation, which should manage to compete with cache-aware B-trees ($O(\log_B(N))$). However, results obtained on the INSERT test disprove this argument. In

addition, TokuDB developers have explained that there is an inevitable tradeoff between search queries and insert queries for this structure (Bender, n.d.). The tabulated values for the average and standard deviation for each of the database scales are presented in Figure 21:

Structure	Scale (Rows)	Average TPS	Standard Dev.
B-Tree	10M	10454.78	904.7
B-Tree	50M	7248.98	492.15
B-Tree	100M	6512	358.91
B-Tree	300M	6548.09	438.15
COLA	10M	8697.68	276.71
COLA	50M	5041.26	183.97
COLA	100M	4297.09	259.13
COLA	300M	3553.65	54.78
COLA	500M	2550.46	67.77

Figure 21: Average and Standard Deviation values for SELECT

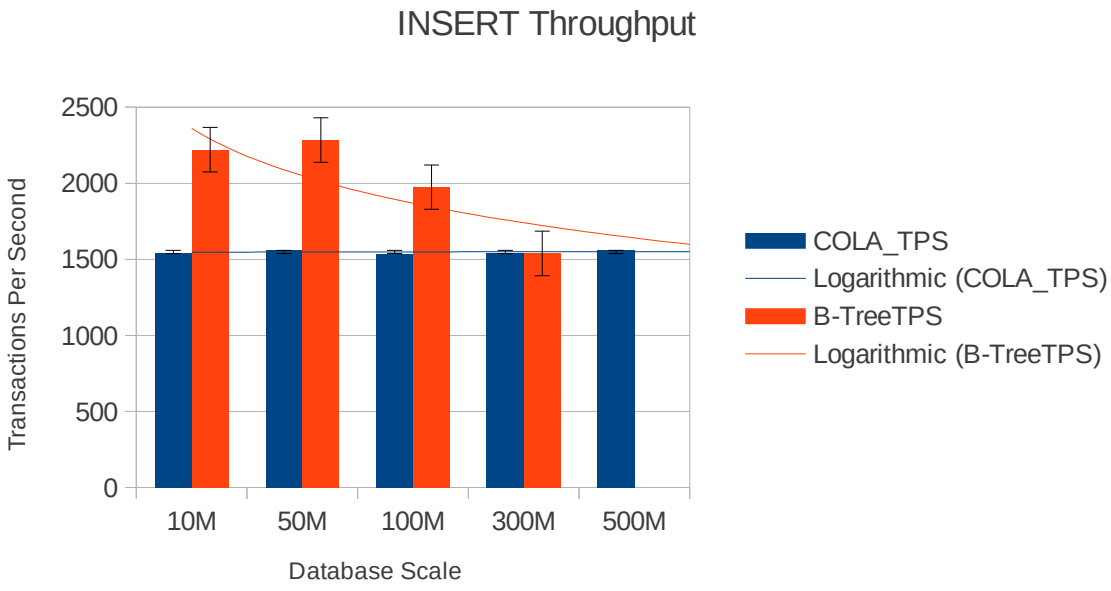

Figure 22: Transaction-per-second averages for INSERT

The final experiment for the COLA vs. B-tree comparison. Figure 22 shows the average INSERT throughput for the sixty samples. From Figure 18, it becomes clear that for relatively

small row count values, B-trees tend to outperform COLA indices considerably. For the three smallest database scales, B-trees showed a 1.395 times larger throughput than COLA. However, as database size increases, the difference between the two indices seems to be reduced. From the figure, it may appear that the critical point for this phenomenon is around the 300 million-row mark. The trend lines show that although, B-trees may seem superior for smaller databases, their throughput seems to decrease more rapidly as databases become larger. On the other hand, COLAs appear to be slow for relatively smaller databases, yet display a trend that resembles near constant time $O(1)$. This result is very insightful to show the COLA's strongest suit, and provides justification for not selecting a smaller database scale in order to fit more samples into the tests' results. Analysis doesn't seem to provide a clear explanation as to the lookahead array indices seemingly constant insert time. Nonetheless, it was expected that a cache-aware implementation for the lookahead array ($O(\frac{\log_B(N)}{B})$) could outperform B-trees ($O(\log_B(N))$), which is exactly what TokuDB's brand offers as a commercial product. The tabulated values for the average and standard deviation for each of the database scales are shown in Figure 23:

Structure	Scale	Average TPS	Standard Dev.
B-Tree	10M	2220.57	75.24
B-Tree	50M	2283.81	135.38
B-Tree	100M	1974.36	108.35
B-Tree	300M	1539.22	146.79
COLA	10M	1543.56	42
COLA	50M	1562.04	38.33
COLA	100M	1534.5	32.09
COLA	300M	1542.46	26.13
COLA	500M	1560.05	37.84

Figure 23: Average and Standard Deviation values for INSERT

CONCLUSIONS

- Observation of modern computers' memory hierarchies has revealed that minimizing I/O operations is key to the design of optimal memory systems.
- The disk access model is a good abstraction for the disk-memory interaction that facilitates the analysis of algorithms and data structures. However, its algorithms require careful implementation and tuning.
- Analytically, B-trees present optimal bounds for range searches and are often more effective at inserts than other cache-aware structures.
- Analytically, hash-based indices are optimal for equality searches. Although, their applications on real life databases are limited due to their lack of support of range searches.
- Both analytically and empirically, B-trees and hash-based indices tend to show similar bounds in terms of disk space overhead.
- Benchmarks performed on the PostgreSQL engine suggest that, although hash-based indices display near constant throughput over growing database sizes, B-trees' throughput scales similarly well.
- Benchmarks performed on the PostgreSQL engine strongly suggest B-trees as a preferable solution within the disk access model for use cases that require high-insertion throughput.
- The cache-oblivious model provides a wide platform to design optimal algorithms and structures that could theoretically be implemented for any pair of levels within the memory hierarchy.

- The cache-oblivious B-tree has been analytically proven to be competitive with its cache-aware counterpart in terms of worst-case bounds required for search and insert.
- Analytically, it has been proven the cache-oblivious lookahead array could outperform B-tree's worst-case insertion bound depending on block size.
- Cache-aware optimizations on the cache-oblivious lookahead array have proven to make the lookahead array's worst-case search bound equal to B-trees and its worst-case insertion bound considerably superior.
- Benchmarks comparing InnoDB's B-tree implementation against TokuDB's COLA implementation suggest that COLA's tend to offer massive space overhead improvements.
- Benchmarks comparing InnoDB's B-tree implementation against TokuDB's COLA implementation suggest that B-trees tend to outperform COLA indices on equality searches.
- Experimental results point to TokuDB's COLA implementation as a highly scalable solution for large databases due to the observation of a nearly constant trend for insert operations' throughput.

REFERENCES

Arge, L. (2003). “The Buffer Tree: A Technique for Designing Batched External Data Structures” (pp. 1-9). Retrieved from: https://www.cs.cmu.edu/~guyb/realworld/slidesF10/buffertree.pdf

Bender, M., Kuszmaul, B. (2012). “Data Structures and Algorithms for Big Databases”(pp. 8-72). Retrieved from: http://people.csail.mit.edu/bradley/BenderKuszmaul-tutorial-xldb12.pdf

Bender, M., Farach-Colton, M., Fineman, J., Fogel, Y., Kuszmaul, B., Nelson, J. (2007). “Cache-Oblivious Streaming B-trees” (pp. 1, 8-11). Retrieved from: http://supertech.csail.mit.edu/papers/sbtree.pdf

Bender, M., Demaine, E., Farach-Colton, M. (2005). “Cache-Oblivious B-Trees” (pp. 1-6, 10). Retrieved from: http://erikdemaine.org/papers/FOCS2000b/paper.pdf

Bender, M. (n. d.) “Performance of Fractal-Tree Databases” (pp. 6-18). Retrieved from: https://www.bnl.gov/csc/seminars/abstracts/Bender_Presentation.pdf

Demaine, E., Devadas, S., Lynch N. (2015). Recitation 2: 2-3 Trees and B-Trees on 6.046J/18.410J : Design and Analysis of Algorithms (pp. 6-7). Massachusetts Institute of Technology, Cambridge, Massachusetts. Retrieved from: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/recitation-notes/MIT6_046JS15_Recitation2.pdf

Demaine, E. (2012). Lecture 7 on 6.851: Advanced Data Structures (pp. 1-8). Massachusetts Institute of Technology, Cambridge, Massachusetts. Retrieved from: https://courses.csail.mit.edu/6.851/spring12/scribe/lec7.pdf

Demaine, E. (2002). “Cache-Oblivious Algorithms and Data Structures” (pp. 3-6, 15-16, 19-23). http://erikdemaine.org/papers/BRICS2002/paper.pdf

Erickson, J. (2002). Lecture 1: Introduction on CS473: Topics in Analysis of Algorithms (pp. 1-4). University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois. Retrieved from: http://jeffe.cs.illinois.edu/teaching/473/01-search+sort.pdf

Erickson, J. (2002). Lecture 2: Buffer Trees on CS473: Topics in Analysis of Algorithms (pp. 1-4). University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois. Retrieved from: http://jeffe.cs.illinois.edu/teaching/473/02-buffertrees.pdf

Fagin, R., Nievergelt, J., Pippenger, N., Strong, H. (1979). “Extendible Hashing-A Fast Access Method for Dynamic Files” (pp. 315-318). Retrieved from: http://researcher.watson.ibm.com/researcher/files/us-fagin/tods79.pdf

Graefe, G., Larson, P. (2001). “B-tree Indexes and CPU Caches” (pp. 10). Retrieved from:

http://www.cse.unt.edu/~huangyan/6330/Papers/B-tree-Cache-ICDE2001.pdf

Kasheff, Z. (2004). “Cache-Oblivious Dynamic Search Trees” (pp. 1, 15-22). Retrieved from: http://people.csail.mit.edu/bradley/papers/Kasheff04.pdf

Kuszmaul, B. (2010). Lecture 19: Cache-oblivious B-Tree (TokuDB) on 6.172 Performance Engineering (pp. 4-32). Massachusetts Institute of Technology, Cambridge, Massachusetts. Retrieved from: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2010/video-lectures/lecture-19-how-tokudb-fractal-tree-indexes-work/MIT6_172F10_lec19.pdf

Moshkovitz, D., Tidor, B. (2012). Lecture 10: Hashing and Amortization on 6.046J/18.410J: Design and Analysis of Algorithms (pp. 1-2, 6-7). Massachusetts Institute of Technology, Cambridge, Massachusetts. Retrieved from: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec10.pdf

Moshkovitz, D., Tidor, B. (2012). Lecture 15: Van Emde Boas Data Structure on 6.046J/18.410J: Design and Analysis of Algorithms (pp. 1-3). Massachusetts Institute of Technology, Cambridge, Massachusetts. Retrieved from: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec15.pdf

Nelson, J. (2013). Lecture 23 on CS229r: Algorithms for Big Data (pp. 1-3). Harvard University, Cambridge, Massachusetts. Retrieved from: https://pdfs.semanticscholar.org/abcc/8e337925ef5d8f8e348c5056256bce9b16bc.pdf

Prokop, H. (1999). “Cache-Oblivious Algorithms” (pp. 55-58). Retrieved from: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-895-theory-of-parallel-systems-sma-5509-fall-2003/readings/cach_oblvs_thsis.pdf

Ramakrishnan, R., Gehrke, J. (2003) “Database Management Systems” (pp. 273-384). Third Edition. McGraw-Hill.

Silberchatz, A., Korth, H., Sudarshan, S. (2010) “Database system concepts” (pp. 476-523), Sixth edition. McGraw-Hill

Schwarz, K. (2016). Lecture 5: Balanced Trees on CS166: Data Structures (pp. 18-41). Stanford University, Stanford, California. Retrieved from: https://web.stanford.edu/class/cs166/lectures/05/Small05.pdf

Vitter, J. (2008). “Algorithms and Data Structures for External Memory” (pp. 83-95). Retrieved from: https://www.ittc.ku.edu/~jsv/Papers/Vit.IO_book.pdf