Optimizing Trees for Static Searchable Encryption

Mohammad Etemad   Mohammad Mahmoody   David Evans
etemad@virginia.edu   mohammad@virginia.edu   evans@virginia.edu
Department of Computer Science, University of Virginia

Abstract

Searchable symmetric encryption (SSE) enables data owners to conduct searches over encrypted data stored by an untrusted server, retrieving only those encrypted files that match the search queries. Several recent schemes employ a server-side encrypted index in the form of a search tree where each node stores a bit vector denoting for each keyword whether any file in its subtree contains that keyword. Our work is motivated by the observation that the way data is distributed in such a search tree has a big impact on the cost of searches. For single-keyword queries, it impacts the number of different paths that must be followed to find all the matching files; for multi-keyword queries, the arrangement of the tree also impacts the number of nodes visited during the search on paths that do not lead to any satisfying data elements. We present three algorithms that improve the performance of SSE schemes based on tree indexes and prove that for cases where the search cost is high, the cost of our algorithms converges to the cost of the optimal tree. In our experiments, the resulting search trees outperform the arbitrary search trees used in previous works by a factor of up to two.

1 Introduction

Much research has been done during the last decade on securely outsourcing data storage to untrusted servers. Data confidentiality can be obtained by encrypting the data, but for most applications, data owners need to be able to conduct searches on the encrypted data. *Searchable encryption* enables the owner to search over encrypted data stored on a remote server, retrieving only the data items that match her queries.

In searchable symmetric encryption schemes [18, 8, 13, 9, 40, 22, 25, 21, 39, 35, 7, 4], the data owner builds an 'encrypted' index and uploads it together with the encrypted files to the server to be used for processing later search queries. Two types of indexes have been used in most existing SSE schemes: *inverted indexes* and *trees*.

SSE schemes based on inverted indexes (e.g., [22, 7, 39, 35]) relate to each keyword the set of identifiers of files containing the keyword. Given some information about the keyword under search according to the index structure, the server can find and return all related files. While being efficient, they leak more information about file insertion/deletion operations and Boolean search queries.

Tree-based SSE schemes (e.g., [18, 21, 36]) build a "*file vector*" (which could be a bit vector or a Bloom filter [1]) for each file that specifies which keywords appear in that file. These file vectors together with the corresponding file identifiers are then assigned to the leaves on which the tree is built. Inner tree nodes also have their own vectors that specify which keywords appear in the *union* of files in that subtree. Tree-based SSE schemes are particularly important because they support not only single-keyword search (e.g., [18, 21]), but also *arbitrary* multi-keyword (Boolean) search where the query is a Boolean formula (for example, this was an essential functionality for Blind Seer [36]). Combined with secure computation techniques, they can also provide *query privacy*.

The vector labels of nodes enable monotone (without negation) Boolean search queries to be processed starting from the root and traversing multiple paths down the tree, where only the children of satisfying nodes are visited, to finally reach the file identifies stored at the leaves satisfying the search query. Our optimization techniques directly apply to any other vector formats (such as designs based on Bloom filter) as long as the vectors encode the keywords of the union of the files in those subtrees. However, for simplicity, we focus on OR trees where the vectors of inner nodes are the bitwise OR of its children's vectors. As an example, a search tree (unencrypted for simplicity) for 8 files

$f_1, \ldots, f_8$ containing 6 keywords in total is given in Figure 1. For instance, f_1 contains only w_1 and w_6. It shows the nodes visited during processing a Boolean search query $q = w_3 \wedge w_4$ and outputs the only satisfying file, f_4.

The exact process at each node of a search tree depends on the data structures and encryption schemes used to store the node's data. However, the total *set* of visited nodes does not depend on whether we use OR vectors or Bloom filters. The search cost is exactly proportional to the number of visited nodes. Assuming the cost of processing a node for a search query q is c, and the number of visited nodes is n_q, the total cost of processing q is $n_q \cdot c$. Since c is fixed for each query on the same tree, n_q determines the total search cost. If several files satisfy a query, their relative locations in the tree affects n_q. Our goal is to minimize the number of nodes visited over a distribution of queries. Since the search operation is the most frequent SSE operation, its performance is the main factor in determining the efficiency of scheme in the long run.

Previous tree-based searchable encryption schemes [18, 21, 36] assign file vectors to the search tree leaves randomly or arbitrarily (but, as we will discuss later, their security proofs do not depend on this arrangement). With the goal of reducing the cost of query processing, our work explores strategies for arranging the search trees in ways that reduce the search cost.

However, the position of each file in the optimized search tree is determined relative to all other files. This directly affects the update since insertion/deletion of a file requires the whole tree to be re-arranged. Therefore, we aim at incorporating the query distribution into the process of generating the optimized search trees for *static* searchable encryption schemes that are suitable for outsourcing backup and census data, for example.

1.1 Contributions

Ours is the first work that takes the query distribution into account in constructing SSE search trees. This results in optimized search trees that are arranged to minimize costs over an expected query distribution. Our approach is driven by the observation that the average search cost of a tree (i.e., what we call the *global cost* of the search operation) can be characterized as the summation of some costs associated with each node (the *local cost*) for any distribution of monotone Boolean queries. Therefore, the global search cost of a tree can be optimized by minimizing its local costs. This allows us to design tree-optimizing algorithms that start with a given set of file vectors for leaves and build the search tree towards the root while minimizing the "local" search costs with respect to the query distribution. Our optimized trees outperform "typical" search trees used in previous work [18, 21, 36] by a factor of up to two. Our optimizations are general and can be applied in any scenario in which deterministic search trees are allowed and random tree arrangements are not necessary for security.

We propose three heuristic algorithms (Section 3) for constructing optimized search trees depending on how the cost minimization is done for internal nodes and whether the obtained tree is balanced or not. The optimized search trees generated by our heuristic algorithms minimize the expected *average* search cost for a given query distribution. Our first algorithm (the *best-first algorithm*) borrows ideas from the Huffman code trees and generates potentially unbalanced trees with minimal average search cost. On the down side, this algorithm constructs trees that are not balanced, and so the shape of the tree could leakage information about its content. To remedy the leakage through the shape of the search tree, our second algorithm (the *level-optimal algorithm*) generates optimized *balanced* binary trees

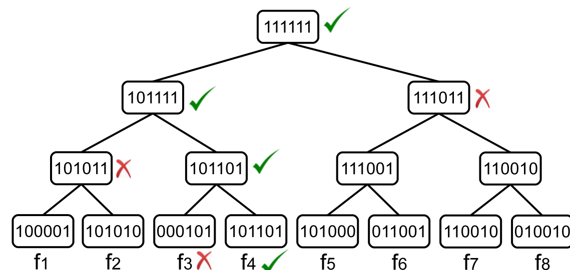

Figure 1: A search tree on 8 files having 6 keywords in total. The marked nodes are visited during processing $q = w_3 \wedge w_4$.

with a height that is logarithmic in the number of files. The bounded height of the balanced search trees also minimizes the *round complexity* (the interactive search processing) of the search operation, if secure computation techniques as in Blind Seer [36] are employed for achieving query privacy as well. This algorithm is expensive due to its use of the Blossom's algorithm of finding optimal matching in general graphs [15, 17], however, perhaps surprisingly, we find experimentally that the optimized search trees generated by our level-optimal algorithm perform very closely to those constructed by the best-first algorithm while guaranteeing the logarithmic depth *in the worst case*. Finally, we propose a third algorithm (the *hybrid algorithm*) that combines ideas from the two previous algorithms to get the best of both worlds: it produces optimized *balanced* trees and generates them much faster than the level-optimal algorithm.

By exhaustive search among all possible trees (for small number of real data files), we find that in most cases, our level-optimal algorithm generates trees that are very close to the optimal trees and that our best-first algorithm indeed produces the *optimal* trees. Further, our intensive experiments show that the level-optimal algorithm performs very closely to the best-first algorithm. The average search cost in trees generated by these algorithms is optimized by a factor of up to two. Similarly, our hybrid algorithm efficiently generates trees with up to 2X improved average search cost compared to the random trees on the same files and query distribution.

In addition to the experiments above, we also theoretically analyze the produced search trees and prove that our level optimal algorithm produces a tree whose search time is within logarithmic factor compared to the global optimal tree (among all possibly *unbalanced* trees), and at the same time its search time is within a *constant* factor compared to the optimal *balanced* tree, as long as the search cost is not "too low" and is above certain threshold. In other words, either the produced tree leads to small search time anyway, or in case it does not, we would know that we are not doing too bad either, as even the optimal balanced solution needs almost the same amount of time to answer search queries.

Leakage. SSE schemes reveal some information for efficiency. Recent results have shown that this leakage can sometimes be exploited to reveal the outsourced encrypted data [20, 29, 5, 34, 42]. The minimal leakage includes the *access pattern* leakage that is the set of encrypted files matching a query and *search pattern* leakage that states how many times a query is searched for [13, 9, 21, 39]. Tree-based schemes further leak the set of nodes visited during processing a query. For single-keyword and disjunctive multi-keyword queries, the visited nodes are determined by the result, so this leakage would carry no information beyond the access pattern. For conjunctive queries, though, there may be some extra nodes visited along paths that do not lead to any satisfying elements.

Our work does not address these leakage issues, which remain a fundamental challenge for searchable encryption [38, 7, 6, 36]. They are orthogonal to this work to mitigate leakage, and consistent with the prevailing research goal of offering well-understood leakage properties with sufficiently efficient and scalable operations. Our optimizations rearrange the search tree in a way that the number of such extra nodes is reduced on average. This reduces the apparent leakage since fewer nodes are revealed, but because the arrangement of the nodes is no longer random each node touched may reveal more information. We note that the proofs of security in Blind Seer [36] does not depend on this randomization and only rely on the defined leakage functions. Therefore, the random trees and our optimized trees need the same leakage function for the security proof, as they both satisfy the security definitions in common usage, but also leak some information that may be exploited. Section 3.5 discusses the leakage in more detail.

1.2 Prior Work

Our work is the first to optimize *any* data structure (not just trees) based on the the query distribution for answering *general monotone* Boolean queries. The idea of designing data structures based on the query distribution and minimizing the *expected access time* has been previously used for binary search trees (BST) [33, 30, 26, 28, 41, 24, 14]. However, BSTs are not designed for answering Boolean query searches, and the best BSTs can do in the context of Boolean query search is to answer single keyword search queries assuming that each keyword appears in at most one file! More recently, the idea of optimizing data structures based on the query distribution is also used for certain geometric problems [3, 11, 19]. When it comes to Boolean query searches, the special case of *intersection* (i.e. conjunctive) queries is studied [12, 23, 10, 37]. However, in addition to being limited in the type of accepted queries (1) their data structure is not tree based and thus not useful for our applications for SSE, and (2) they do not incorporate the query distribution into account.

Single-keyword queries. The existing single-keyword SSE schemes [13, 9, 22, 25, 39, 35, 4] employ an encrypted inverted index that maps to each keyword the set of identifiers of files containing the keyword. These schemes suit

single-keyword search well. Answering Boolean queries with these schemes requires revealing the query structure to the server. Tree-based indexes have also been used for single-keyword search [18, 21]. These two works differ in the data structure used at the nodes: Goh [18] uses a Bloom filter, while Kamara and Papamanthou [21] use hash tables.

Boolean queries. Some works targeted general Boolean search [32, 7, 36], where multiple keywords can be combined in a logical expression. We discuss only the tree-based one.

Blind Seer [36] employs a tree-based index and regards query privacy. The data structure employed to store nodes' values is an encrypted Bloom filter. It also combines different cryptographic techniques and building blocks to hide the structure of the query while enabling the server evaluate the query and find the answer.

The tree-based schemes in both groups do not consider the effect of an optimized tree on the search cost. Together with the fact that search is a frequent operation in an SSE scheme, optimizing the tree affects performance considerably. Our work aims at optimizing the search tree and is applicable on top all the exiting tree-based works, for both single-keyword and Boolean search schemes.

2 Preliminaries

The formal definition of SSE scheme and its security follows.

Notation. λ is the security parameter. $[a,b]$ denotes the set of integers from a to b, and we define $[n] = [1,n]$. A function $\nu : \mathbb{Z}^+ \rightarrow [0,1]$ is called *negligible* if $\forall$ positive polynomials $p(.)$ and $\forall\, n \in \mathbb{Z}^+,\ n > n_0 : \nu(n) < 1/p(n)$. For random variable $\mathcal{X}$, by $x \leftarrow \mathcal{X}$ we denote that x is an element sampled form the distribution of $\mathcal{X}$. Let $F = (f_1, f_2, ..., f_n)$ be a collection of n files. When it is clear from the context, we might simply use f_i to denote $id(f_i)$. We assume the file identifiers are not encrypted; they can simply be some random strings. The set of all unique keywords $(w_1, w_2, ..., w_m)$ appearing in all files in the collection is called the *dictionary*.

2.1 Symmetric Searchable Encryption

A searchable symmetric encryption scheme involves two parties: a *client* (data owner) and a *server*. The client encrypts a set of files and builds an encrypted index. These are transferred to the server, which is expected to provide responses to queries from the client but not trusted to see the data. The server stores the index and files and answers the client queries. We assume a single-client model.

The client sends search queries to the server to execute on the encrypted outsourced files. These requests include tokens generated using a secret key stored by the client. In a bandwidth-efficient scheme, the server's response only includes encrypted files that made the search query.

Now, we give a general definition of an SSE scheme following definitions from prior works [22, 9, 6]:

Definition 2.1 (SSE Scheme) *An SSE scheme consists of the following polynomial time algorithms:*

- `Gen`$(1^\lambda) \rightarrow K$*: is a probabilistic algorithm run by the client. Given the security parameter* λ*, it generates a key* K.
- `Build`$(K, F) \rightarrow (\mathcal{I}, C)$*: is a probabilistic algorithm run by the client. Given a key* K *and a file collection* F *as input,* `Build` *generates an encrypted index* $\mathcal{I}$ *and a collection of encrypted files* C *(corresponding to* F*). The client uploads both of these to the server, removing its local copy of* F.
- `Token`$(K, q) \rightarrow t_q$*: is the token generation algorithm run by the client. It takes as input the key* K *and a query* q*, and generates the corresponding search token* t_q.
- `Search`$(t_q, \mathcal{I}, C) \rightarrow C_q$*: is executed by the server to find and return the encrypted files* C_q *matching a query* q*. The client provides the search token* t_q *and the server supplies the encrypted index* $\mathcal{I}$ *and the collection of encrypted files* C.

Leakage. All known SSE schemes [8, 13, 9, 40, 22, 25, 21, 7, 39, 35, 36, 4] allow some information be leaked during each operation, and specify the amount of leakage in the form of leakage functions. The security definition of an SSE scheme is then based on showing that the scheme does not reveal any information beyond what is specified by the leakage functions. We use the same data structure (search trees) for our index as is used by Blind Seer [36] and Kamara *et al.* [22], and define the leakage function for our static scheme identically to these schemes as follows:

• $\mathcal{L}_{\texttt{Build}}$ denotes the information leaked to the server during initialization by Build: $\mathcal{L}_{\texttt{Build}}(F) = (n, m, (|f_1|, \ldots, |f_m|))$ where $(f_1, \ldots, f_m) = F$. This includes the number of files n, the dictionary size m, and the sizes of all files.

• $\mathcal{L}_{\texttt{Search}}$ denotes the information leakage during processing a query q: $\mathcal{L}_{\texttt{Search}}(q) = (T_q, F_q)$ where T_q is the set of nodes of the search tree that are traversed, and F_q is the set of identifiers of files satisfying q (i.e., the access pattern of q).

During the Build, the server learns the number and sizes of the files. A search operation reveals the set of files matching the query. These are the minimal leakages accepted in most of SSE schemes, and are independent of the SSE constructions. Tree-based indexes leak the set of nodes visited during search.

Threat model. We assume the client is trusted. The server is honest-but-curious and follows the scheme as defined, but it tries to learn more information than allowed through the leakage functions (e.g., by combining the leakage with some prior knowledge) at the end of protocol execution. We define security of our scheme as in Stefanov et al. [39], where a real-world experiment simulates the ideal-world functionality. Since we do not modify the scheme, just change the tree arrangement, we do not include the full details here. Previous proofs of security for tree-based searchable encryption schemes [21, 36] do not depend on the arrangement of tree nodes.

3 Optimizing Search Trees

Search is the most frequent operation and the dominant factor in determining the cost of a searchable encryption scheme. Once the search tree (index) is built and outsourced, the server uses it to respond to many search queries. In this section, we describe algorithms for constructing the search tree with the goal of minimizing the average search cost over an expected query distribution.

As opposed to trying to optimize the search time for each individual query, our approach is to minimize the total expected cost for a *distribution* over a set of expected queries. In other words, we optimize the "average-case" cost of the search operation as opposed to targeting "worst-case" [2].

OR **trees.** We now define the labeled binary tree data structure for performing the search operation in SSE that we optimize in this work. We denote the set of leaves and internal nodes of a binary tree T by $L(T)$ and $I(T)$, respectively. For any node $u \in T$ other than the root, $\pi(u)$ denotes the parent node of u. For an internal node $u \in I(T)$, $\Gamma(u) = \{x \neq y \mid \pi(x) = \pi(y) = u\}$ denotes the set of its two children. We always construct trees over a set of files starting from the leaves. Therefore, we will be given $L(T) = \{f_1, \ldots, f_n\}$ with vector labels $\{\bar{f}_1, \ldots, \bar{f}_n\}$. For a Boolean formula q over m binary variables, and a node $u \in T$, $q(\bar{u}) = 1$ if and only if the vector $\bar{u} \in \{0,1\}^m$ satisfies the Boolean formula q. For any binary vector $\bar{v} \in \{0,1\}^m$, we denote the jth bit of $\bar{v}$ by $\bar{v}[j]$. For any file f_i, we assign a binary vector $\bar{f}_i$ such that $\bar{f}_i[j] = 1$ if $w_j \in f_i$. For a Boolean query q over variables $(x_1, \ldots, x_m)$ and a vector $\bar{v} \in \{0,1\}^m$, we say that $\bar{v}$ satisfies q, and denote it by $q(\bar{v}) = 1$, if by setting $x_i = \bar{v}[i]$ for all $i \in [m]$, the formula q evaluates to 'true'. The set of identifiers of files satisfying a query q is denoted by $F_q = \{f_j \colon q(\bar{f}_j) = 1\}$. We will only work with full binary trees, and simply call them trees when it is clear from the context.

Definition 3.1 (OR **trees)** *Suppose T is a binary tree. We call T an* OR tree *if for every non-leaf node u, with children $\{x, y\}$ it holds that $\bar{u} = \mathsf{OR}(\bar{x}, \bar{y})$ where $\bar{u}[i] = \bar{x}[i] \vee \bar{y}[i]$ for all $i \in [m]$.*

Search cost. To compare different trees and select the most efficient one, we need to compare the cost of performing the search operation on them. The search cost is determined by the number of nodes visited during a search, which we denote as $|\text{Vis}_q(T)|$ which is the size of the set of nodes visited while searching the query q in T. The time to visit each node should be constant (otherwise it would leak information), so this quantity will be proportional to the actual search time. Since our objective is to optimize the search cost *on average* over a *distribution* of queries, we aim to minimize the expected size of $\text{Vis}_q(T)$ over the choice of q from its distribution.

3.1 Algorithms for Optimizing OR Trees

We describe three heuristic algorithms that aim at finding OR trees with minimum average search cost $\mathbb{E}_{q \leftarrow \mathcal{Q}}\left[|\text{Vis}_q(T)|\right]$ for a distribution of queries $\mathcal{Q}$. Our algorithms are designed to use OR trees, and thus we allow the set of queries Q

Algorithm 1: Best-first algorithm - general trees

Input: $\{f_1,\dots,f_n\}$ describing n files with vector labels $\{\bar{f}_1,\dots,\bar{f}_n\}$ in $\{0,1\}^m$, and a distribution $\mathfrak{Q}$ over the set of monotone formulas over m Boolean variables.
Output: An OR tree T with leaves $L(T)=\{f_1,\dots,f_n\}$.

1. Let $W=L(T)=\{f_1,\dots,f_n\}$.
2. For every $x,y\in W$, let z be a new (potential) node with label $\bar{z}=\mathsf{OR}(\bar{x},\bar{y})$, and let $C(x,y)=\mathsf{P}_{\mathfrak{Q}}(z)$.
3. **while** $|W|>1$ **do**
4.   Find the pair $x\neq y$ in W with minimum $C(x,y)$.
5.   Add a new node $z\in I(T)$ with label $\bar{z}=\mathsf{OR}(\bar{x},\bar{y})$.
6.   Set $\pi(x)=\pi(y)=z$ and $\Gamma(z)=\{x,y\}$.
7.   Remove x,y from W, and add z to W.
8.   Update $C(z,u)$ for all $u\in W$.
9. Return $W=\{r\}$ as the root of the constructed tree T.

include arbitrary *monotone* Boolean formulas. That is because if we allow negations in q, a search starting from the root of an OR tree with leaves $\{f_1,\dots,f_n\}$ (with labels $\{\bar{f}_1,\dots,\bar{f}_n\}$) might miss some of the nodes that should be part of the output set.

All our algorithms construct the tree inductively by starting from the given set of leaves and building the tree towards the root, assuming we can efficiently compute (or approximate) the probability that the label $\bar{u}$ of the node u satisfies $q\leftarrow\mathfrak{Q}$, which we denote as $\mathsf{P}_{\mathfrak{Q}}(u)=\Pr_{q\leftarrow\mathfrak{Q}}[q(\bar{u})=1]$ for any node u in the tree with a vector label $\bar{u}\in\{0,1\}^m$. In Section 3.3, we show how to compute this probability for natural query distributions over single-keyword as well as conjunctive and disjunctive multi-keyword queries. The quantity $\mathsf{P}_{\mathfrak{Q}}(u)$ plays a central role in our algorithms and will be used as the "cost" function that we aim to minimize (its summation over all nodes). By linearity of the expectation, the "global cost" $\mathbb{E}_{q\leftarrow\mathfrak{Q}}[|\mathrm{Vis}_q(T)|]$ of searching a query $q\leftarrow\mathfrak{Q}$ is *proportional* to the sum of the "local costs" $\mathsf{P}_{\mathfrak{Q}}(u)$ for all internal nodes[1] $u\in I(T)$. Hence, our algorithms aim at minimizing this summation $\sum_{u\in I(T)}\mathsf{P}_{\mathfrak{Q}}(u)$. Appendix B proves this property.

We introduce our algorithms assuming we have an efficient procedure to compute $\mathsf{P}_{\mathfrak{Q}}(\cdot)$. We analyze running time of each algorithm (to construct the tree) assuming it takes time $t_{\mathfrak{Q}}$ to compute $\mathsf{P}_{\mathfrak{Q}}(\cdot)$. Finally, we discuss the time complexity $t_{\mathfrak{Q}}$ of computing $\mathsf{P}_{\mathfrak{Q}}(\cdot)$ for various natural query distributions.

Best-first algorithm for general unbalanced trees (Algorithm 1)**.** Our first approach is inspired by the idea behind the Huffman codes and leads to potentially unbalanced trees. Starting with leaf nodes, $W=L(T)=\{\bar{f}_1,\dots,\bar{f}_n\}$, the best-first algorithm keeps matching pairs, $\{x,y\}\subseteq W$, as siblings into a (new) parent node z and replace $\{x,y\}$ with z in the set W until we are left only with the root node, $W=\{r\}$. At each step, we choose to match a pair that gives us a parent node z with minimum "local cost" $\mathsf{P}_{\mathfrak{Q}}(z)$ among all possible choices of z that are available at the moment.

This algorithm leads to solutions that (based on our experimental results in Section 4.2) are very close to optimal solutions over all OR trees. However, as the constructed trees might have different shapes (not necessarily balanced), which requires including the shape of the tree in the build leakage.

Level-optimal algorithm for balanced trees (Algorithm 2)**.** Our second algorithm produces *balanced* OR trees with depth $\log(n)$, eliminating any shape leakage. Another motivation behind the second algorithm is to guarantee *worst-case* $\log(n)$ search cost when we have access to parallel computing resources and still want to minimize the total CPU time across different parallel threads. As with the best-first algorithm, we start with the leaves $L=\{\bar{f}_1,\dots,\bar{f}_n\}$ and build the tree towards the root by finding pairing of nodes x,y and creating their portent node $z=\pi(x)=\pi(y)$. But, instead of choosing pairs that minimize the function $C(x,y)=\mathsf{P}_{\mathfrak{Q}}(z)$ "one-by-one", we choose the best possible way to do a "global pairing" of the nodes in the current level of the tree. In other words, we aim at finding the least costly *weighted matching* among the nodes of W based on the cost function defined by $C(x,y)=\mathsf{P}_{\mathfrak{Q}}(z)$. We use the Blossom algorithm [15, 17] for weighted matching in general graphs to find a this matching among the "current" nodes of the tree in W. This process goes on "level-by-level" until the complete tree is constructed. One drawback of this algorithm is its scalability, as its running time is lower bounded by the running time of the expensive Blossom algorithm.

[1] In fact, this intuition only works when the OR tree is "valid" for the queries $q\leftarrow\mathfrak{Q}$ that always happens when q is a monotone query. See Appendix B for more details.

Algorithm 2: Level-optimal algorithm - balanced trees

Input: $\{f_1,\dots,f_n\}$ describing $n=2^k$ files with vector labels $\{\bar{f}_1,\dots,\bar{f}_n\}$ in $\{0,1\}^m$, and a distribution $\mathcal{Q}$ over the set of monotone formulas over m Boolean variables.

Output: A balanced OR tree T over the leaves $\{f_1,\dots,f_n\}$.

1 Let $W = L(T) = \{f_1,\dots,f_n\}$.
2 **while** $|W| > 1$ **do**
3 Let G_W be a complete weighted graph over vertices W. For every $x \neq y \in W$, let z be an imaginary node with label $\bar{z} = \mathsf{OR}(\bar{x},\bar{y})$, and connect x to y with an edge of weight $C(x,y) = \mathsf{P}_{\mathcal{Q}}(z)$.
4 Run the Blossom algorithm to find the matching M in G_W with $|W|/2$ edges and minimum total weight $\sum_{(x,y)\in M} C(x,y)$.
5 **for** *all* $(x,y) \in M$ **do**
6 Let z be a new node in $I(T)$ with label $\bar{z} = \mathsf{OR}(\bar{x},\bar{y})$.
7 Set $\pi(x) = \pi(y) = z$ and $\Gamma(z) = \{x,y\}$.
8 Remove $\{x,y\}$ from W, and add z to W.
9 Return $W = \{r\}$ as the root of the constructed tree T.

Hybrid algorithm (Algorithm 3)**.** Our third algorithm is a hybrid variant of the two previous algorithms that inherits the scalability of first algorithm and the balanced trees of our second algorithm. It uses a heuristic matching algorithm based on our first algorithm, rather than running Blossom's algorithm. This algorithm is scalable and does not need to leak shape of tree, yet it improves upon random trees by a factor of up to 2.

Algorithm 3: Hybrid algorithm - balanced trees

Input: $\{f_1,\dots,f_n\}$ describing $n=2^k$ files with vector labels $\{\bar{f}_1,\dots,\bar{f}_n\}$ in $\{0,1\}^m$, and a distribution $\mathcal{Q}$ over the set of monotone formulas over m Boolean variables.

Output: A balanced OR tree T over the leaves $\{f_1,\dots,f_n\}$.

1 Let $U = L(T) = \{f_1,\dots,f_n\}$.
2 **while** $|U| > 1$ **do**
3 Let $W = U$.
4 Let $U = \emptyset$.
5 For every $x,y \in W$, let z be a new (potential) node with label $\bar{z} = \mathsf{OR}(\bar{x},\bar{y})$, and let $C(x,y) = \mathsf{P}_{\mathcal{Q}}(z)$.
6 **while** $|W| > 1$ **do**
7 Find the pair $x \neq y$ in W with minimum $C(x,y)$.
8 Add a new node $z \in I(T)$ with label $\bar{z} = \mathsf{OR}(\bar{x},\bar{y})$.
9 Set $\pi(x) = \pi(y) = z$ and $\Gamma(z) = \{x,y\}$.
10 Remove x,y from W, and add z to U.
11 Return $U = \{r\}$ as the root of the constructed tree T.

3.2 Construction Cost Analysis

In this section, we presume we can compute the "local cost" $\mathsf{P}_{\mathcal{Q}}(u)$ of an internal node $u \in I(T)$ in time $t_{\mathcal{Q}}$, and analyze the cost of our tree construction algorithms. The time $t_{\mathcal{Q}}$ to compute $\mathsf{P}_{\mathcal{Q}}(u)$ actually depends on m and the complexity of query distribution $\mathcal{Q}$. In Section 3.3 we give a detailed discussion on examples of $t_{\mathcal{Q}}$ for different natural distributions $\mathcal{Q}$.

Best-first algorithm. Our best-first algorithm could be implemented using a heap data structure that contains the $O(n^2)$ weights of the edges of the graph to find their minimum in each step efficiently. We first need to compute all the n^2 weights, which takes time $n^2 \cdot t_{\mathcal{Q}}$, and then construct the heap on them which takes time $O(n^2 \cdot \log n)$. Then, in each iteration, we will need to update $O(n)$ elements in the heap. The total update of the weights would be $O(n^2 \cdot t_{\mathcal{Q}})$ and the total time to relocate them in the heap would be $O(n^2 \cdot \log n)$. Thus, the running time will be $O(n^2 \cdot \log n + n^2 \cdot t_{\mathcal{Q}})$.

Level-optimal algorithm. In our level-optimal algorithm, the most expensive step is to run the Blossom algorithm to find the weighted matching. It takes $O(n^3)$ time [16, 27, 31], so the total time spent on finding matchings will be

$O(n^3) + O((n/2)^3) + O((n/4)^3) + \cdots = O(n^3)$. Processing the weights of the edges also takes time $n^2 t_{\mathcal{Q}} + (n/2)^2 t_{\mathcal{Q}} + \cdots = O(n^2 t_{\mathcal{Q}})$. Thus, the total running time will be $O(n^3 + n^2 \cdot t_{\mathcal{Q}})$.

Hybrid algorithm. The running time of our hybrid algorithm is closely related to the best-first algorithm. We can again use a heap to find the pair with minimum local cost in each step. But after finding the pair with minimum local cost, we do not add the parent of these nodes to the same level in the tree, and we keep that node for the next layer. Therefore, the running time of the algorithm for pairing the leaves will be the same $\alpha(n) = O(n^2 \cdot \log n + n^2 \cdot t_{\mathcal{Q}})$ as it was for the best-first algorithm, but now we have to repeat this for each layer. This leads to running time $\alpha(n) + \alpha(n/2) + \ldots$ which is still bounded by $2\alpha(n) = O(n^2 \cdot \log n + n^2 \cdot t_{\mathcal{Q}})$.

3.3 Computing Local Cost Functions

Our tree-construction algorithms require a way to efficiently compute (or approximate) $\mathsf{P}_{\mathcal{Q}}(u) = \Pr_{q \leftarrow \mathcal{Q}}[q(\bar{u}) = 1]$ for the query distribution $\mathcal{Q}$ over monotone (Boolean) queries. We discuss the time $t_{\mathcal{Q}}$ and describe how to efficiently compute $\mathsf{P}_{\mathcal{Q}}(u)$ for some interesting and natural classes of distributions.

Single keywords. For the case of searching single keywords, assuming all keywords are searched with equal probabilities, $\mathcal{Q}$ would be a uniform distribution over $\{w_1, \ldots, w_m\}$. We denote *Hamming weight* of a vector, $\bar{v} \in \{0,1\}^m$, as $\mathsf{HW}(\bar{v}) = |\{i \mid \bar{v}[i] = 1\}|$. The local cost of an internal node u with a vector $\bar{u} \in \{0,1\}^m$ would be equal to the normalized Hamming weight of its label: $\mathsf{P}_{\mathcal{Q}}(u) = \mathsf{HW}(\bar{u})/m$. For a more general, possibly non-uniform distribution $\mathcal{Q}$ over single keywords search queries, $\mathsf{P}_{\mathcal{Q}}(u)$ is still easily computable as it will be equal to the corresponding weighted version of the Hamming weight: $\mathsf{P}_{\mathcal{Q}}(u) = \sum_{\bar{u}[i]=1} \Pr[\mathcal{Q} = w_i]$. The time $t_{\mathcal{Q}}$ to compute $\mathsf{P}_{\mathcal{Q}}(\cdot)$ in all these cases will be in $O(m)$.

Conjunctions of multiple keywords. If $\mathcal{Q}$ is uniformly distributed over the set of two keyword conjunctive queries (i.e., $q = w_i \wedge w_j$ for random $i \neq j$), we can again efficiently compute the local cost of a node u based on the Hamming weight of its label $\bar{u}$:

$$\mathsf{P}_{\mathcal{Q}}(u) = \Pr_{i \neq j \leftarrow [m]}[\bar{u}[i] \wedge \bar{u}[j] = 1] = \binom{\mathsf{HW}(\bar{u})}{2} / \binom{m}{2}$$

which is simply proportional to $\mathsf{HW}(\bar{u}) \cdot (\mathsf{HW}(\bar{u}) - 1)$ (multiplied by a constant $1/(m(m-1))$ that is independent of u). This argument generalizes to the case of $\mathcal{Q}$ being a uniform distribution over k keywords leading to (for a constant c depending on m):

$$\mathsf{P}_{\mathcal{Q}}(u) = \binom{\mathsf{HW}(\bar{u})}{k} / \binom{m}{k} = c \cdot \prod_{0 \leq i \leq k-1} (\mathsf{HW}(\bar{u}) - i).$$

Like the case of single keyword, the time $t_{\mathcal{Q}}$ to compute $\mathsf{P}_{\mathcal{Q}}(u)$ would be $O(m)$ because computing the Hamming weight of the node is sufficient for $t_{\mathcal{Q}}$ using a pre-computed table of corresponding costs as a function of Hamming weight.

Disjunctions of multiple keywords.. For the case where $\mathcal{Q}$ is uniformly distributed over OR queries of k keywords, a similar argument to the AND case leads to $\mathsf{P}_{\mathcal{Q}}(u) = 1 - \binom{m - \mathsf{HW}(u)}{k} / \binom{m}{k}$ which also can be computed in time $t_{\mathcal{Q}} \leq O(m)$.

General samplable distributions. In most real-life scenarios, the distribution $\mathcal{Q}$ over the Boolean queries is not as simple as the cases discussed above, and it is rather a mixture of different types of queries that could be potentially described by an efficient sampling procedure that *generates* q according to the implicit distribution $\mathcal{Q}$. In this case, we let $G(R)$ be an efficient algorithm describing a process that outputs $q \leftarrow G(R)$ when R is chosen at random.

In the setting of samplable $\mathcal{Q}$, it might be computationally infeasible to compute the local cost function $\mathsf{P}_{\mathcal{Q}}(u)$ *exactly*, however, we can still overcome this obstacle by *approximating* $\mathsf{P}_{\mathcal{Q}}(u)$ with arbitrary small additive error ε. The intuition here is to use multiple samples from $q \leftarrow G(R)$ and see whether $q(\bar{u}) = 1$ or $q(\bar{u}) = 0$. Using these samples, one can approximate the probability $\mathsf{P}_{\mathcal{Q}}(u)$, by taking the *average* of $q(\bar{u})$ for $q \leftarrow G(R)$. More formally, if we sample k independent samples $q_1, \ldots, q_k$ from the distribution $\mathcal{Q}$ and take $\alpha = \frac{1}{k} \cdot \sum_{i \in [k]} q_i(u)$, by Hoeffding's inequality, with probability $1 - 2e^{-\varepsilon^2 k}$ over the choice of the random samples, it holds that $|\alpha - \mathsf{P}_{\mathcal{Q}}(u)| \leq \varepsilon$.

If we want to get at most ε additive errors with probability $1 - \delta$, we need $k = O(\log(1/\delta)/\varepsilon^2)$ samples from $q \leftarrow G(R)$. Assuming it takes time t_G to generate $q \leftarrow G(R)$, and that it takes time t_C to compute $q(\bar{u})$, the time $t_{\mathcal{Q}}$

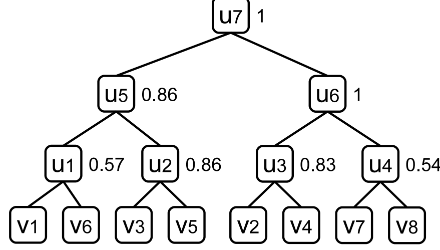

Figure 2: A random search tree on the given file vectors.

to ε approximate $\mathsf{P}_\Omega(u)$ with error $\leq \delta$ would be $O(t_G \cdot t_C \cdot \log(1/\delta)/\varepsilon^2)$. Therefore, if we want to get $\delta = \mathrm{negl}(\lambda)$ for security parameter λ, we can get an ε-approximation for $\mathsf{P}_\Omega(u)$ with probability $1 - \mathrm{negl}(\lambda)$ by running in time $t_\Omega = \omega(t_G \cdot t_C \cdot \log \lambda/\varepsilon^2)$. In this case, because the error probability of getting (an ε approximation for) every $\mathsf{P}_\Omega(u)$ is $\mathrm{negl}(\lambda)$ and that we run in $\mathrm{poly}(\lambda)$ the total error remains $\mathrm{poly}(\lambda) \cdot \mathrm{negl}(\lambda) \leq \mathrm{negl}(\lambda)$.

Case of large query set Q. An interesting case for which we can apply the above techniques for samplable Ω is the case where we are given a *large* set Q explicitly and we know that q is uniformly distributed over Q. In this case, computing the local cost/weight values of each pair of nodes takes time $t_\Omega = \Omega(|Q| \cdot m)$, leading to total running time of at least $\Omega(n^2 \cdot |Q| \cdot m)$ for all of our algorithms of our algorithms which might be prohibitive depending on $|Q|$ and m. To remedy this inefficiency, we might opt to *approximate* the cost of the new nodes by using k samples from $|Q|$. This leads to $t_\Omega \approx k \cdot t_C$ where t_C is the time needed to compute $q(\bar{u})$ which, for $k \ll |Q|$ could significantly improve the running time.

3.4 An Illustrative Example

We give an example to better understand the concept of local and global costs and how our algorithms generate the optimized search trees. Assume there are $n = 8$ files and the vectors are given as: $v_1 = 1000011010$, $v_2 = 1110101001$, $v_3 = 1111011000$, $v_4 = 1000010111$, $v_5 = 1010000111$, $v_6 = 0110001011$, $v_7 = 1101000010$, and $v_8 = 1000011001$. Let Ω be the set of all conjunctive queries with two keywords that result in at least one matching file.

Figure 2 presents a balanced binary tree constructed on a random permutation of the given file vectors. None of our algorithms is used here. First, the set of pairs at the leaves is fixed, and then the tree is build on it toward the root. The local cost of each internal node is given beside the node. The average global cost of the tree is $\mathbb{E}_{q \leftarrow \Omega}[|\mathrm{Vis}_q(T)|] = 2 \times (0.57 + 0.86 + 0.83 + 0.54 + 0.86 + 1 + 1) + 1 = 12.32$. Therefore, processing a conjunctive query with two keywords requires visiting 12.32 nodes of this tree, on average.

Figure 3 illustrates how our level-optimal algorithm generates the optimized search tree. Starting from the file vectors at leaf level, the perfect weighted matching algorithm [15, 17] is applied to generate one level above with the minimum sum of local costs. This is repeated until the root is generated. The average global cost of this tree is $\mathbb{E}_{q \leftarrow \Omega}[|\mathrm{Vis}_q(T)|] = 9.38$. In this small example, we observe $\approx 24\%$ improvement on average search cost compared to the random tree.

The search tree generated by our best-first algorithm is shown step by step in Figure 4. The average global cost of the tree is $\mathbb{E}_{q \leftarrow \Omega}[|\mathrm{Vis}_q(T)|] = 9.34$. This is the optimum search tree found by exhaustive search for this example. Thus, for this example, even our level-optimal algorithm generates a search tree with global costs very close to the optimum tree. Section 4 reports on experiments that confirm this is commonly the case, and Appendix B.2 proves that it is asymptotically optimal, at least for single-keyword and disjunctive queries.

3.5 Leakage

Any operation in an SSE scheme reveals some information to the server. Controlling the leakage is important especially considering the increasing number of attacks on SSE scheme that exploit these leakages to reveal the outsourced encrypted data [20, 29, 5, 34, 42].

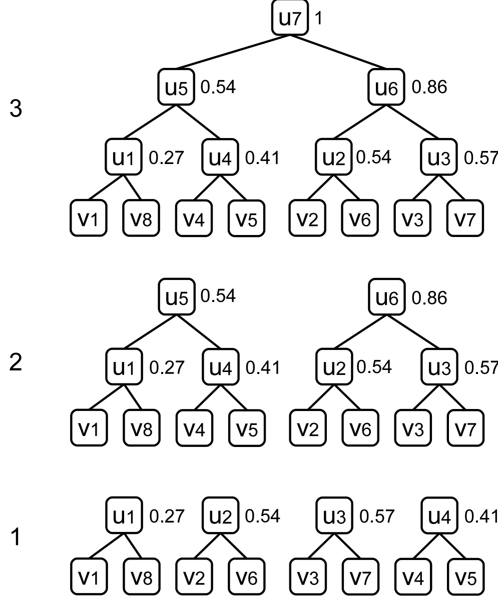

Figure 3: Search tree generated by level-optimal algorithm.

The minimal leakage in almost all of the existing schemes includes the number and sizes of the outsourced files, (an upper bound on) the number of total (keyword, file identifier) pairs, the *access pattern* leakage (the set of encrypted files matching a search query) and *search pattern* leakage (how many times a query is searched for; i.e., the history of queries) [13, 9, 21, 39]. A tree-based index, in addition, leaks the set of nodes visited during processing a query that we call the *tree pattern* leakage. We investigate how much extra information will be leaked by the tree pattern leakage. We separate the single-keyword and disjunctive multi-keyword queries from the conjunctive multi-keyword queries.

For single-keyword and disjunctive multi-keyword queries, the visited nodes are determined by the result, so the tree pattern leakage would carry no information beyond the access pattern leakage.

However, the case of conjunctive queries is different and complicated since the search tree is based on OR operations. Consider two queries: $q_a = w_1 \wedge w_2$ and $q_o = w_1 \vee w_2$. If q_o is evaluated true at any internal node u, there is at least one leaf node satisfying q_o on the subtree rooted at u. This is not the case for q_a since w_1 and w_2 may have opposite values at both children of u, making q_a false. This means that the search starts from the root, goes ahead till u (and some other paths as well) and stops at u without reaching a result. As an example, assume half of the files contain 'Female' and 'Dept1' and the other half contain 'Male' and 'Dept2', and they are assigned to every other leaf of the search tree. The query $q_a = Female \wedge Dept2$ traverses the whole tree and returns no result. We are not aware of any algorithm that can process general conjunctive queries in sublinear time, without visiting extra nodes.

Therefore, there may be always some extra nodes visited along paths that do not lead to any satisfying elements. Moreover, the server observes the set of processed tree nodes. (An adversary observing the communication can also realize the fact by comparing the access pattern with the query process time.) The extra nodes reveal that the query includes a part of the form $q_1 \wedge q_2$ and each child satisfies either q_1 or q_2.

It seems that this extra leakage cannot be prevented unless an algorithm avoiding processing extra nodes is found for processing conjunctive queries. Otherwise, the adversary can distinguish two datasets of the same size by choosing an appropriate query that returns the same result set on both datasets, with a considerable difference in processing times.

Our optimizations rearrange the search tree in a way that the number of extra nodes visited to process a query is reduced on average. Therefore, we expect the above-mentioned leakage be reduced by this optimization (though visiting a large part of the tree for each query also reduces the leakage). On the other hand, this rearrangement may reveal some information about similarity of the files over the time. Similarly to Blind Seer, in case of optimized *balanced* search trees (i.e. the trees produced by our level optimal and hybrid algorithms) we can also randomly

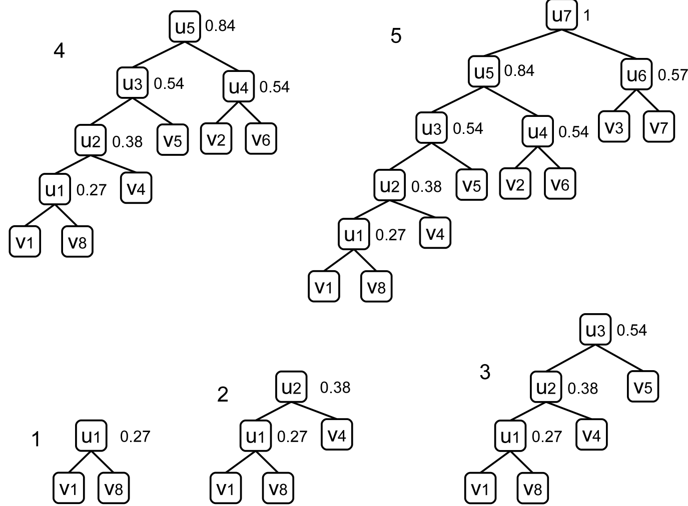

Figure 4: Search tree generated by our best-first algorithm.

permute the nodes of the constructed tree before outsourcing it to the server. The hope is that this final rearrangement will decrease the information carried by the leakage function about the data. We note, however, that the proofs of security in Blind Seer do not depend on this randomization and only rely on the above mentioned leakage functions. Therefore, the random trees and our optimized trees need the same leakage function for the security proof, as they both satisfy the security definitions in common usage, but also leak some information that may be exploited by an adversary. The consequences of this leak is not well understood in practice, and we leave further exploration of the impact of our tree rearrangements, as well as opportunities to control leakage by rearranging trees, as open problems.

We believe this leakage is acceptable in many situations, as it is in fact the same exact type of leakage that is allowed in previous works [36, 21] for proving the security. However, one can apply different methods to reduce the leakage. If the conjunctive queries are limited to two or three keywords, one can build a large index including all those conjunctive queries as single-keyword queries. While eliminating the aforementioned leakage, this cannot be extended to contain all combinations of keywords as it is growing exponentially.

Another option would be to follow some of the non-satisfying paths to a leaf. This will increase the size of query results, but it makes the AND queries similar to OR queries and prevents the above leakage. Applying our optimizations together with this method reduces the overhead on top of the original query results.

4 Performance

We evaluate performance of our tree construction algorithms in the context of previous tree-based schemes [18, 21, 36] where our optimizations are directly applicable. Since those schemes do not assume any specific arrangement, we model their search trees as a (uniformly) randomly generated tree and compare that to our constructed trees. Our goal is to investigate how our optimizations affect the search cost, which as previously argued, is proportional to the number of nodes visited during processing a search query for all of these schemes. Thus, instead of measuring the operation times, we compute and use the "global search cost", defined in Section 3 to be simply the number of nodes visited during the search.

Since we describe the cost as the average number of nodes visited during processing a query, our cost function is independent of any mechanism used for encrypting and later processing the nodes. Our optimizations are applicable to any tree-based searchable encryption scheme, including Goh's [18], Kamara and Papamanthou's [21], and Blind Seer [36]. We find that whatever mechanism such as secure computation is used for processing the nodes, our optimized trees improve the average search cost by a factor of up to two compared to randomly-constructed trees.

Setup. We implemented a prototype of our efficient search trees employing all of the best-first, level optimal, and

hybrid algorithms in Matlab R2015b to evaluate their performance. We use a binary vector $\bar{f}_i \in \{0,1\}^m$ to represent the keywords in each file f_i, where m is the dictionary size, and build the tree over these vectors. For simplicity, we use vectors without encryption since the focus of this work is on the performance of the search operation run over our trees. Indeed, once the optimized tree is constructed, it can be encrypted and any other technique such as Bloom filters can be used to encode the vectors. We only store the name of the respective file with each vector.

Dataset. We use randomly selected subsets of the Enron email archive (`http://www.cs.cmu.edu/~enron/`) as the test dataset. The dictionary is the union of all words of length between 5 and 10. The keyword length has no effect on the search cost as all keywords are processed similarly. The file size does not directly affect the search cost (except that longer files are likely to contain more keywords).

Experiments. We perform three types of experiments. The first two series of experiments investigate random samples for large numbers of files as well as exhaustive searches (to find the optimal tree) for a small number of files. In the first set of experiments, we generate search trees employing all three of our proposed algorithms as well as 100 randomly generated trees constructed over the same set of file vectors and query distributions, and then we compare their search costs. In the second set of experiments, we exhaustively enumerate all possible search trees built on the same data and query distribution for $n = 8$ files and then compare them against our algorithms. The third set of experiments studies how the tree optimized for one query type/distribution behaves for other query types/distributions.

For large keyword sets, the query space for Boolean queries consisting of 3 (and more) keywords becomes very large. Since it does not fit into our machine's memory, we employ the approximation technique described in Section 3.3 to sample a random subset of size 5,000,000 of the total query space which approximates the original query distribution with a very high accuracy. Utilizing the same approximation, we randomly sample 500 cells of the file vectors to find the minimum matchings for the hybrid algorithm. Once the set of matchings is determined, the costs are all computed using the original file vectors. Again, this approach for dealing with large query sets (that leads to large time for computing $\mathsf{P}_{\mathbb{Q}}(u)$) is a special case of the general method described in Sectoin 3.3. By the Hoeffding bound, using 500 randomly selected cells, the approximated local cost function of each node will be within $\pm 1/10$ additive approximation of the real local costs with probability $1 - 2e^{-500/10^2} > 0.98$.

4.1 Comparison with Random Balanced Trees

We randomly select different numbers of files from the Enron email archive, extract the words whose lengths are between 5 and 10 to form the dictionary and file vectors, perform four kinds of experiments on them, and compare the results. First, we apply both our best-first and level-optimal algorithms on the resulting file vectors. Then, we generate 100 random search trees on the same file vectors and compute their average search cost. These three sets of experiments are run for $n \leq 2^{10}$ files. We further use our hybrid algorithm to process $n > 2^{10}$ files. Finally, we compare the search cost of single-keyword and monotone Boolean queries with two and three keywords on the search trees generated these three ways.

As expected, the best-first algorithm generates search trees with the lowest average search cost. However, the resulting trees may not be necessarily balanced binary trees, and hence, they have worst case search costs linear in n and require linear rounds of interactions during search. The cost of processing search queries with the optimized search trees generated by our level-optimal algorithm is close to the trees built by the best-first algorithm, while preserving the logarithmic height. For example, our level-optimal and best-first algorithms process a two-keyword conjunctive query for $n = 1024$ and $m = 11308$ with cost 37.99 and 37.57, respectively (while a randomly-generated search tree requires cost 71.41 on average). Table 1 (in Appendix A) provides details on these results.

Since building the tree for $n > 2^{10}$ files using the level-optimal algorithm is expensive, we use the hybrid algorithm (Algorithm 3) for those experiments. The hybrid algorithm performs closely to the level-optimal algorithm and approximately halves the global search cost in our experiments. Figure 5 shows the results for single-keyword queries. The x-axis shows the logarithm (in base two) of the number of files and the y-axis shows the logarithm (in base two) of search costs. Until the point $x = 10$ ($n = 2^{10}$ files), the results of the best-first and level-optimal algorithms are shown. After this point, it presents the cost of trees generated by our hybrid algorithm against the random trees.

We observe the same behavior for Boolean queries with multiple keywords as in the single-keyword case. For example, we investigated conjunctive and disjunctive queries with two and three keywords and observed that the search costs for trees generated by our level-optimal algorithm are very close to that of the trees built employing

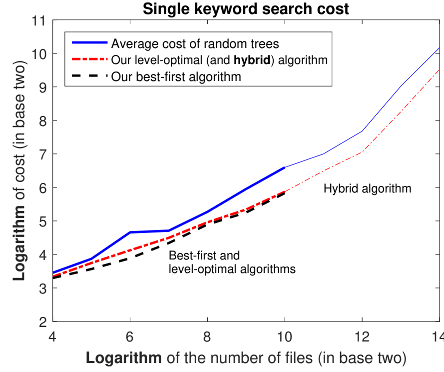

Figure 5: Average cost of single-keyword queries.

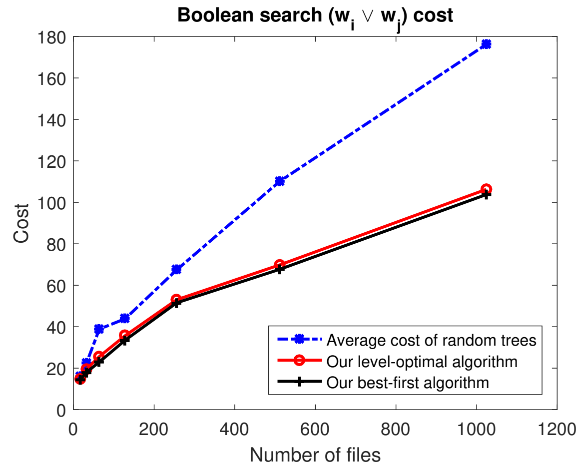

Figure 6: Average cost of 2-keyword disjunctive queries.

our best-first algorithm. For disjunctive queries, we also observe up to 2X improvement by our efficient search trees compared to randomly-built trees as shown in Figures 6 and 7.

4.2 Comparison Against All Possible Trees

With n leaf nodes, there are $\Pi_{i=1}^{n-1}(2i-1)$ different valid binary trees. Since this increases very rapidly with n, the highest number of the form $n = 2^k$ that we could try exhaustively was $n = 2^3$. The number of all such trees for $n = 8$ is 135,135. (It becomes more than 6×10^{15} for $n = 16$.) We select eight files randomly from the Enron email archive, process them as before to build the dictionary and file vectors, and compare results of our optimized search trees against the minimum cost of all possible trees.

The experiment is repeated with different randomly selected sets of eight files. The details of these experiments are given in Table 2 (in Appendix A). The exhaustive search among all possible balanced trees finds slightly better trees than our level-optimal algorithm algorithm in only 4 out of 25 experiment sets. The difference between the two costs in the worst case is only 0.07. Similarly, our best-first algorithm finds search trees very close in cost to the optimum tree. Checking all possible trees found slightly better trees in 13 out of 25 experiments. The two costs differ by only 0.09 in the worst case.

We also observe that for reasonable dictionary sizes (i.e., greater than 1000), the average search cost of trees generated by our level-optimal algorithm is similar to that of the trees generated by the best-first algorithm. This means that we are not sacrificing performance to limit the search to balanced trees, so there is no need to risk leaking

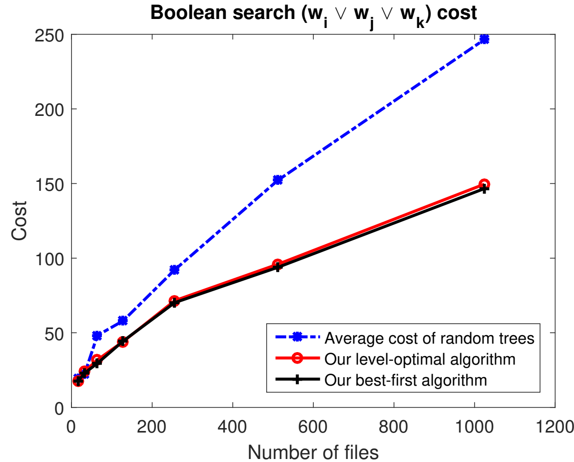

Figure 7: Average cost of 3-keyword disjunctive queries.

information or sacrificing depth guarantees with differently shaped trees.

4.3 Uncertain Query Distributions

In many cases, the actual distribution of queries is not known, so it is important to see how differences in the query distribution impact the results. We perform two sets of experiments to analyze 1) how optimizing the search tree based on the single-keyword query distribution affects the cost of Boolean queries with two and three keywords, and 2) how optimizing the search tree based on an estimated distribution over the two-keyword conjunctive queries affects the cost of single-keyword and Boolean queries with two and three keywords on the original distribution.

For the first case, we construct the optimized search tree given the single-keyword query distribution, and measure the costs of Boolean queries with two and three keywords on the resulting tree. We find that search trees optimized using our heuristics assuming the single-keyword query distribution still work well for Boolean queries. In almost all cases, the difference in cost between the two cases is less than one; i.e., less than one extra node is processed on average. For example, the average search cost of conjunctive queries with two and three keywords on their respective optimized trees are 37.99 and 31.97, respectively, for $n = 1024$ and $m = 11308$. The same costs on a search tree that is optimized using the single-keyword query distribution are 38.83 and 32.36, respectively. Table 3 (in Appendix A) provides detailed results.

This was expected since processing a Boolean query includes a combination of single-keyword query processes. The importance of this observation becomes even more clear when we consider the fact that single-keyword based optimizations are generally easier (i.e., the trees are constructed faster).

For the second case, we want to see if optimizing the search tree for an 'estimated' distribution (on two-keyword conjunctive queries, for example) leads to good optimizations for the uniform distribution of other queries. Therefore, we generate an estimated query distribution as follows: Keep picking a keyword proportional to its frequency, and stop the process with some fixed probability $p = \frac{1}{2}$ after any pick. This will generate queries with different number of keywords. (The expected average query length is two.) Then, we select the resulting queries of length two (treating them as conjunctive queries), and build the optimized search tree according to a uniform distribution on these queries. Finally, we measure the average search cost of our five groups of queries on the constructed tree. The results of this experiment together with a similar experiment on the uniform distribution over all two-keyword conjunctive queries are given in Table 4 (in Appendix A). The average search costs on the tree that is optimized for the estimated distribution are very close to, and in some cases even better than the respective values on the other search tree that is optimized for the original distribution. For example, the average search cost of disjunctive queries with two keywords on the search trees that are optimized based on the original and estimated query distributions for $n = 1024$ and $m = 11308$ are 107.24 and 108.19, respectively; while the same costs are 151.79 and 151.26, respectively, for disjunctive queries

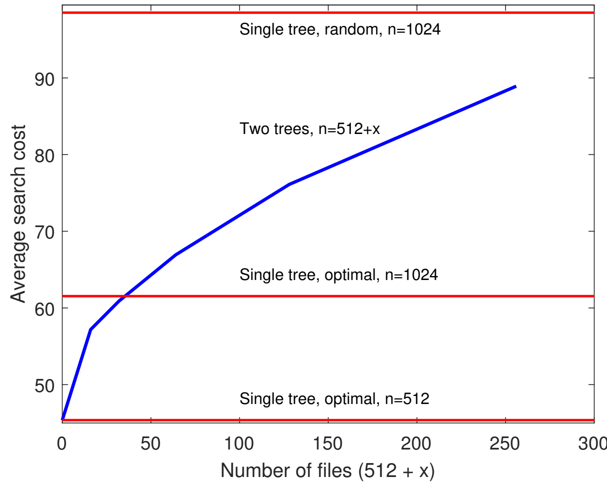

Figure 8: Average search cost in two-tree setting.

with three keywords. I.e., the optimized tree based on the estimated query distribution performs slightly better in the latter case.

5 Discussion

In this section, we discuss issues related to dynamicity of our algorithms and open questions.

Dynamic search trees. Our optimized tree has a deterministic structure that is determined by the set of files and their keywords, making it suitable for *static* outsourced data. Since the position of each file (vector) in the tree depends on all other files, a file insertion/deletion may change the tree completely and require the tree to be rebuilt. However, if the updates are rare, we can support it as follows.

Similar to Blind Seer [36], we employ another tree (or list) storing the recently added files. This will be a random tree that supports file addition/deletion. When the size of this tree reaches a certain threshold, both trees are removed and a new optimized search tree for the currently existing files is constructed. The update frequency and the threshold amount determine the frequency of rebuild. Setting the threshold to n, the random tree can grow up to the size of the optimized tree, after which a rebuild is required. This results in amortized update cost $O(n^2 + n \cdot t_Q)$ for our level-optimal algorithm, and $O(n \cdot \log n + n \cdot t_Q)$ for our hybrid and best-first algorithms.

Search operates on both trees, finds files satisfying the query on both of them, and returns them to the client. Note that the search cost of the second tree would be at most two times of the cost of the optimized tree. Hence, the search cost remains asymptotically the same. Figure 8 presents the total search cost for an optimized tree with 512 files and a random tree with different files (from 16 to 256) assuming that deletions do not happen in between. As the figure shows after 256 file insertions, the total search cost of both trees becomes 88.92. The average search cost in this two-tree setting is still acceptable while it allows file insertions.

We can assume the two trees as the children of a bigger tree. The search process starts the root of this bigger tree and continues on the children. Both trees have similar leakages that if combined, it would be equal to the leakage of the bigger tree. I.e., the total leakage of these two trees would not exceed the leakage of an optimized or random tree of size both of them.

Deletion removes a file and updates the corresponding search tree accordingly; though the resulting optimized tree would not necessarily be the optimal one. However, we may accept it as the optimal one till the next rebuild if deletions are rare. Note that this does not affect the leakage of the scheme since the adversary cannot decrypt the older

versions of the updated parts of the index anymore.

Open questions. Our experimental analysis of our algorithms in Section 4 suggest that when it comes to real data, all our algorithms lead to search time that beat random trees by a constant factor and are probably within a constant factor of optimal search produced by optimal search trees. In our theoretical analyses of Section B we proved that some of these intuitive statements are in fact true when it comes to our level optimal algorithm. However, a theoretical analysis of the optimality of our best first and hybrid algorithms are left as interesting directions for future work. As we showed, when it comes to small number of files, an exhaustive search shows that our best-first algorithm is very close to being optimal. As another direction for future work one can also try to find other heuristic algorithms for finding the minimum (perfect) matching in weighted graphs and derive new forms of hybrid algorithms along the line of our Algorithm 3. For some of those algorithms, one might be able to prove a concrete bound on the optimality of the produced trees.

6 Conclusion

We proposed three heuristic algorithms for constructing efficient balanced and unbalanced trees for searchable encryption. Our algorithms are based on the observation that the way data is distributed in a search tree has a big impact on its efficiency. Our algorithms rearrange the tree in a way that the number of visited nodes/paths during a search is minimized based on the distribution of the input queries. Our optimized search trees improve the search cost by a factor of up to two compared to randomly-built trees as is confirmed by our extensive experiments. All three algorithms are applicable on the existing tree-based SSE schemes.

The two balanced variants (i.e., the level-optimal and the hybrid algorithms) always produce balanced trees that eliminates the leakage through the shape of the tree before search operations are performed. These two algorithms are also of special importance since they guarantee a worst case logarithmic (in the number of files) height and thus support parallel search of at most logarithmic time. We also found experimentally that the level-optimal algorithm produces trees that perform very close to the optimal trees. Therefore, we can use optimized balanced trees and have (almost) the better search cost of optimized unbalanced trees while benefiting from the extra features guaranteed by balanced search trees.

References

[1] Burton H Bloom. Space/time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] Andrej Bogdanov and Luca Trevisan. Average-case complexity. *Foundations and Trends in Theoretical Computer Science*, 2(1), 2006.

[3] Prosenjit Bose, Luc Devroye, Karim Douieb, Vida Dujmovic, James King, and Pat Morin. Odds-on trees. *arXiv preprint arXiv:1002.1092*, 2010.

[4] Raphael Bost. $\sigma o \varphi o \varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1143–1154. ACM, 2016.

[5] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[6] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *Network and Distributed System Security Symposium*, 2014.

[7] David Cash, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. *IACR Cryptology ePrint Archive*, 2013.

[8] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, pages 442–455. Springer, 2005.

[9] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT'10*, pages 577–594. Springer, 2010.

[10] Yangjun Chen and Weixin Shen. On the intersection of inverted lists. In *Proceedings of the International Conference on Foundations of Computer Science (FCS)*, page 51, 2015.

[11] Siu-Wing Cheng and Man-Kit Lau. Adaptive point location in planar convex subdivisions. In *International Symposium on Algorithms and Computation*, pages 14–22. Springer, 2015.

[12] J Shane Culpepper and Alistair Moffat. Efficient Set Intersection for Inverted Indexing. *ACM Transactions on Information Systems*, 29(1), 2010.

[13] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS'06*, 2006.

[14] Roberte De Prisco and Alfredo De Santis. New lower bounds on the cost of binary search trees. *Theoretical computer science*, 156(1-2):315–325, 1996.

[15] Jack Edmonds. Paths, Trees, and Flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.

[16] Harold Neil Gabow. Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs. 1974.

[17] Zvi Galil. Efficient Algorithms for Finding Maximum Matching in Graphs. *ACM Computing Surveys*, 18(1):23–38, 1986.

[18] Eu-Jin Goh. Secure indexes. Technical report, Cryptology ePrint Archive, Report 2003/216, 2003.

[19] John Iacono and Wolfgang Mulzer. A static optimality transformation with applications to planar point location. *International Journal of Computational Geometry & Applications*, 22(04):327–340, 2012.

[20] M Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS'12*, 2012.

[21] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. *Financial Cryptography and Data Security, FC*, 2013.

[22] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM CCS*, pages 965–976, 2012.

[23] Andrew Kane and Alejandro López-Ortiz. Intersections of inverted lists., 2016.

[24] Maria Klawe and Brendan Mumey. Upper and lower bounds on constructing alphabetic binary trees. *SIAM Journal on Discrete Mathematics*, 8(4):638–651, 1995.

[25] Kaoru Kurosawa and Yasuhiro Ohtaki. Uc-secure searchable symmetric encryption. In *Financial Cryptography and Data Security*. Springer, 2012.

[26] Lawrence L Larmore. A subquadratic algorithm for constructing approximately optimal binary search trees. *Journal of Algorithms*, 8(4):579–591, 1987.

[27] Eugene L Lawler. *Combinatorial Optimization: Networks and Matroids*. Courier Corporation, 2001.

[28] Christos Levcopoulos, Andrzej Lingas, and Jörg-R Sack. Heuristics for optimum binary search trees and minimum weight triangulation problems. *Theoretical Computer Science*, 66(2):181–203, 1989.

[29] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[30] Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5(4):287–295, 1975.

[31] Kurt Mehlhorn and Guido Schäfer. Implementation of $O(nmlogn)$ Weighted Matchings in General Graphs: the Power of Data Structures. *Journal of Experimental Algorithmics*, 7:4, 2002.

[32] Tarik Moataz and Abdullatif Shikfa. Boolean symmetric searchable encryption. In *8th ACM SIGSAC symposium on Info., computer and communications security*, pages 265–276, 2013.

[33] SV Nagaraj. Optimal binary search trees. *Theoretical Computer Science*, 188(1):1–44, 1997.

[34] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[35] Muhammad Naveed, Manoj Prabhakaran, Carl Gunter, et al. Dynamic Searchable Encryption via Blind Storage. In *IEEE Symposium on Security and Privacy*, 2014.

[36] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind Seer: A Scalable Private DBMS. In *35th IEEE Symposium on Security and Privacy*, 2014.

[37] Giovanni Sacco. Fast block-compressed inverted lists. In *Database and Expert Systems Applications*, pages 412–421. Springer, 2012.

[38] Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker. Towards an information theoretic analysis of searchable encryption. In *International Conference on Information and Communications Security*, pages 345–360. Springer, 2008.

[39] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. *NDSS'14*, 2014.

[40] Peter Van Liesdonk, Saeed Sedghi, Jeroen Doumen, Pieter Hartel, and Willem Jonker. Computationally Efficient Searchable Symmetric Encryption. In *Secure Data Management*. 2010.

[41] Raymond W Yeung. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, 37(3):564–572, 1991.

[42] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium 2016*, pages 707–720. USENIX Association.

A Detailed Performance Results

Table 1 presents results of our experiments comparing our optimized search trees against the random balanced trees. There are five groups, each including three experiments: the *Avg* column shows the average search costs of 100 randomly-constructed trees, the *LOA* column gives the average costs of search trees generated by our level-optimal algorithm, and the *BFA* column provides the average costs of search trees generated by our best-first algorithm; all on the same set of files and query distribution. Comparing the three columns in each group, the table illustrates that the best-first algorithm generates search trees with the lowest average search cost in almost all groups. (The *BFA* column has the smallest values in all five groups.) However, the resulting trees may not be necessarily balanced binary trees (with logarithmic heights). Such trees have worst case search costs linear in n and require linear rounds of interactions during search. Further, they may leak more information (e.g., about the number of keywords in the files) through the shape of the constructed tree. The optimized search trees generated by our level-optimal algorithm process search queries with costs close to the trees built by the best-first algorithm, while preserving the logarithmic height.

Table 1: Comparison of average search costs of our optimized trees against the random trees.

#Files	Dict. size	Single-keyword search cost			Boolean search cost: AND, 2 keywords			Boolean search cost: OR, 2 keywords			Boolean search cost: AND, 3 keywords			Boolean search cost: OR, 3 keywords		
		Avg	*LOA*	*BFA*	*Avg*	*LOA*	*BFA*	*Avg*	*LOA*	*BFA*	*Avg*	*LOA*	*BFA*	*Avg*	*LOA*	*BFA*
32	1687	14.61	13.40	11.83	12.17	11.28	7.91	22.39	19.51	18.02	11.33	10.91	9.35	26.10	23.97	22.96
64	1776	25.26	17.44	14.79	20.24	13.91	8.88	38.85	25.62	22.97	19.01	13.48	11.93	48.22	31.82	29.51
128	3238	26.14	22.56	20.26	19.05	16.43	12.07	43.83	35.66	33.49	16.60	15.06	13.38	58.02	44.44	44.10
256	4656	38.62	31.11	29.72	28.22	22.44	20.45	67.46	53.01	51.54	22.24	18.96	14.30	92.19	71.52	70.22
512	5848	61.98	40.53	38.10	41.33	25.72	23.29	110.20	69.85	67.71	32.40	22.18	19.87	152.21	95.93	93.89
1024	11308	96.84	58.54	56.84	71.41	37.99	37.57	176.19	106.24	103.75	58.07	31.97	30.58	246.43	149.54	146.57

Avg denotes the average search cost of 100 random trees generated on the same input. *BFA* and *LOA* refer to our best-first and level-optimal algorithms, respectively, showing the average search cost for trees built with each algorithm.

Table 2 shows the results of five groups of experiments: one single-keyword and four Boolean searches. Each group includes four experiments on the same data employing our best-first and level-optimal algorithms (the *LOA* and *BFA* columns) and exhaustive search among all balanced (the *min B* column) and all possible trees (the *min U* column) with 8 nodes. For many of the experiments, the tree found by our algorithm is indeed optimal (denoted with "-" in the table). When our heuristics do not find the optimal arrangement, the tree found is very close to optimal.

Table 2: Exhaustive test results.

Dict. size	Single-keyword cost				Boolean search cost: AND, 2 keywords				Boolean search cost: OR, 2 keywords				Boolean search cost: AND, 3 keywords				Boolean search cost: OR, 3 keywords			
	LOA	*min B*	*BFA*	*min U*	*LOA*	*min B*	*BFA*	*min U*	*LOA*	*min B*	*BFA*	*min U*	*LOA*	*min B*	*BFA*	*min U*	*LOA*	*min B*	*BFA*	*min U*
328	7.57	-	5.85	-	7.09	-	5.30	-	9.09	-	7.73	-	7.01	-	4.61	-	11.03	-	9.02	-
770	7.46	-	6.96	-	7.09	-	5.77	5.75	9.75	9.68	9.25	-	7.02	-	4.77	-	11.00	-	10.69	-
1289	7.73	-	7.08	7.07	7.19	-	5.86	5.84	9.96	-	9.31	-	7.06	7.05	4.79	-	11.24	-	10.66	-
1591	7.76	-	7.45	7.36	7.22	7.21	6.21	6.19	10.20	-	9.92	9.89	7.06	7.05	5.04	5.03	11.61	-	11.41	11.40
1726	7.82	-	7.51	7.50	7.31	-	7.01	6.98	10.42	-	10.07	10.05	7.15	-	6.72	6.70	11.97	-	11.62	11.60

Comparison of search costs of our optimized trees against the minimum cost of all possible trees for $n = 8$. *LOA* refers to our level-optimal algorithm and *BFA* denotes our best-first algorithm. *min B* and *min U* stand for minimum cost among all balanced and unbalanced trees, respectively. '-' indicates that the best cost found is the same as the cost found by our algorithm for that experiment.

Table 3 contains results of five groups of experiments for single-keyword and Boolean queries. We optimize the search tree according to the single-keyword query distribution whose results are given in the *Opt* column of the single-keyword experiment, and then use it to process the Boolean queries too. The *S-Opt* columns in the Boolean queries part indicate the respective search costs on this optimized tree. The *Opt* columns in the Boolean queries part show the search costs when the tree is optimized according to their respective distributions.

Table 4 contains the results of five groups of queries using two trees, one is optimized based on a uniform distributions over all two-keyword conjunctive queries (column *Reg*) and the other is optimized based on an estimated uniform distributions over two-keyword conjunctive queries (column *Est*). Costs of all other queries are computed using these two trees. The table illustrates that the average search costs of similar queries using the two optimized trees do not vary considerably. This tells us that even in cases that we do not have access to the original query distribution, we can continue with an estimated one and end up with similar results.

B Theoretical Analysis

In this section, we provide the formal intuition behind our three algorithms of Section 3, and then we will focus to analyze our level-optimal algorithm of Section 3. First, we develop an way to compute the search cost based on "local costs"; i.e., how the internal nodes contribute to the "global cost" of the search operation. This applies to all of our algorithms and is the main intuition behind how they work. Then we prove two theorems about the optimality of

Table 3: Average search costs on a tree optimized for single-keyword query distribution.

#Files	Dict. size	Single-keyword cost		Boolean search cost												
				AND, 2 keywords			OR, 2 keywords			AND, 3 keywords			OR, 3 keywords			
		Avg	*Opt*	*Avg*	*S-Opt*	*Opt*	*Avg*	*S-Opt*	*Opt*	*Avg*	*S-Opt*	*Opt*	*Avg*	*S-Opt*	*Opt*	
32	1687	14.61	13.4	12.17	11.52	11.28	22.39	19.82	19.51	11.33	11.09	10.91	26.10	24.58	23.97	
64	1776	25.26	17.44	21.24	14.32	13.91	38.85	25.81	25.62	19.01	13.64	13.48	48.22	32.01	31.82	
128	3238	26.14	22.56	19.05	16.86	16.43	43.83	36.69	35.66	16.6	15.44	15.06	58.02	48.03	46.10	
256	4656	38.62	31.11	28.22	22.87	22.44	67.46	53.3	53.01	22.24	19.33	18.96	92.19	72.49	71.52	
512	5848	61.98	40.53	41.33	26.57	25.72	110.2	70.16	69.85	32.40	22.70	22.18	152.21	96.81	95.93	
1024	11308	96.84	58.54	71.41	38.87	37.99	176.19	106.85	106.24	58.07	32.26	31.97	246.43	150.81	149.54	

Opt shows results of optimizing each search tree for the respective query distribution, while *S-OPT* denotes the costs of Boolean queries on a search tree optimized for the single-keyword query distribution. *Avg* denotes the average search cost of 100 random trees generated on the same input and query distribution.

Table 4: Average search costs on two trees optimized based on the uniform and an estimated query distributions.

#Files	Dict. size	Single-keyword cost		Boolean search cost							
				AND, 2 keywords		OR, 2 keywords		AND, 3 keywords		OR, 3 keywords	
		Reg	*Est*	*Reg*	*Est*	*Reg*	*Est*	*Reg*	*Est*	*Reg*	*Est*
32	1687	13.44	13.59	11.28	11.47	20.34	21.07	11.10	11.11	24.37	24.52
64	1776	18.16	18.34	13.91	14.37	27.11	27.13	13.72	13.75	32.49	32.41
128	3238	23.01	23.08	16.43	16.75	37.22	37.17	15.43	15.43	47.84	47.97
256	4656	31.11	32.04	22.44	22.85	53.43	53.34	19.33	19.33	72.45	72.33
512	5848	41.43	41.04	25.72	26.18	70.14	70.16	22.70	22.88	97.09	97.21
1024	11308	58.54	59.12	37.99	38.71	107.24	108.19	32.27	33.15	151.79	151.26

The columns titled *Reg* show the search costs on the optimized tree for a uniform distribution on all two-keyword conjunctive queries, while *Est* columns contain the costs on an optimized search tree for an estimated distribution on two-keyword conjunctive queries.

our level-optimal algorithm with respect to single keyword or disjunctive query search in various settings when we compare its efficiency to both balanced and unbalanced trees.

B.1 Characterizing the Search Cost

The following definition captures the class of search trees (not limited to OR trees) that could be used to find the leaves satisfying a given Boolean formula q by simply running a search starting from the root:

Definition B.1 (Valid search trees) *Suppose T is a rooted tree such that all $u \in T$ have vector labels $\bar{u} \in \{0,1\}^m$. We say a tree T is a* valid *search tree for a query q (over m Boolean variables) if $\forall v \in L(T)$, if $q(v) = 1$ then $q(u) = 1$ also holds for all nodes u that are on the path from f to the root (including the root). A* set *$\mathcal{T}$ of trees is valid for a* set *Q of queries, if all $T \in \mathcal{T}$ is valid for all $q \in Q$.*

Notation. We let $\text{Sat}_q(T)$ be the set of nodes of T whose labels satisfy q, namely $\text{Sat}_q(T) = \{u \mid u \in T, q(\bar{u}) = 1\}$. By $\text{Vis}_q(T)$ we denote the set of nodes that we visit if we start a search from the root and visit the nodes as follows. Visit u either if u is the root or if the parent of u satisfies q (i.e., $q(\pi(u)) = 1$). Note that when we search for the Boolean query q, the actual output of the search operation will only contain the leaves that satisfy q. Namely, the output would be the identifiers of the files in $\text{Sat}_q(T) \cap L(T)$.

Using valid trees for search. If a family of trees $\mathcal{T}$ is valid for a set of Boolean search queries Q, we can indeed use any $T \in \mathcal{T}$ to perform a search operation for any $q \in Q$ by running a search algorithm that simply starts from the root, and finds all the nodes in $\text{Sat}_q(T)$ as follows: for every visited node, if $q(u) = 1$, the search proceeds to visit its children $\Gamma(u)$. The validity condition then guarantees that if $q(u) = 1$ the search, starting from the root, will eventually reach u. Recall that by $\text{Vis}_q(T)$ we denote the set of all nodes that are visited in such search operation for query q.

The following lemma shows the intuitive fact that OR trees are valid for searching the set of monotone Boolean formulas (i.e., Boolean formulas that avoid using NOT):

Lemma B.1 *Let M be the set of all monotone Boolean formulas. The set of OR trees are valid for the query set M.*

Proof B.1 *If $v \in L(T)$ for an OR tree T and $u \in \Pi(v)$ is one of its predecessors, $\bar{v}[i] = 1$ always implies $\bar{u}[i] = 1$ as well. Since all $q \in Q$ are monotone, $q(u) = 1$ implies $q(v) = 1$.*

Cost of search. Suppose T is a valid tree for searching query $q \in Q$. If we use T to make a search starting from the root to find $\mathrm{Sat}_q(T)$, it happens that some of the visited nodes like u in this search might *not* satisfy q (i.e., $q(u) = 0$) and only get visited because $q(\pi(u)) = 1$. Even though, when we reach such u we will no longer visit the subtree rooted at u, yet, in order to find the actual cost of the search operation, we will have to take into account the cost of visiting such nodes as well. Namely, the size of the set $\mathrm{Vis}_q(T)$ (and not just $|\mathrm{Sat}_q(T)|$) would be proportional to the search cost.

The following lemma allows us to characterize the search cost $|\mathrm{Vis}_q(T)|$ based on the "local cost" functions that depend on the internal nodes. One interesting aspect of the following lemma is that it captures the cost of the search solely based on the internal nodes of T, even though there may be leaves in $L(T)$ that are visited but do not satisfy q.

Lemma B.2 (Search cost of fixed queries) *If T is a valid binary tree for query q, and r is the root of T, then $|\mathrm{Vis}_q(T)| = 2 \cdot |\mathrm{Sat}_q(T) \cap I(T)| + 1$.*

Proof B.2 (Proof of Lemma B.2) *For every non-root internal node $u \in I(T)$, if $q(u) = 1$, then by the validity of T, searching q would lead to visiting u as well as visiting its two children $\{x, y\}$ who might or might not satisfy q. For every such u, we will count its 2 children nodes visited during the search. (In this counting method, we do* not *count the node u itself). This way, we will count every visited node in $\mathrm{Vis}_q(T)$ (including the leaf nodes some of which will be returned as final answers to the search operation) exactly once. That is because we will visit the parent of every $u \in \mathrm{Vis}_q(T), u \neq r$ exactly once! The only exceptional node in $\mathrm{Vis}_q(T)$ that is not counted in this double counting argument is the root node r that does not have any parent. Since we always start the search from the root, and thus visit it, the actual cost of searching q will be $|\mathrm{Vis}_q(T)| = 2 \cdot |\mathrm{Sat}_q(T) \cap I(T)| + 1$.*

Now, suppose $\mathcal{Q}$ is a *distribution* over a set of queries Q. For a fixed valid tree T for query set Q, the *average* cost of searching $q \leftarrow \mathcal{Q}$ over T would be equal to $\mathbb{E}_{q \leftarrow \mathcal{Q}}[|\mathrm{Vis}_q(T)|]$ which is the expected number of nodes in T that are visited in searching a random query sampled according to $q \leftarrow \mathcal{Q}$. The next lemma characterizes the "global search cost" of a random query $q \leftarrow \mathcal{Q}$ based on the summation of the "local costs" $\Pr_{q \leftarrow \mathcal{Q}}[q(u)]$ over internal nodes $u \in I(T)$.

Lemma B.3 (Characterizing the Search Cost) *Let $\mathcal{Q}$ be a distribution over the set of queries Q for which T is a valid tree. Then the expected number of nodes visited during the search for q is equal to $\mathbb{E}_{q \leftarrow \mathcal{Q}}[|\mathrm{Vis}_q(T)|] = 1 + 2 \cdot \sum_{u \in I(T)} \mathsf{P}_{\mathcal{Q}}(u)$ where $\mathsf{P}_{\mathcal{Q}}(u) = \Pr_{q \leftarrow \mathcal{Q}}[q(u) = 1]$.*

Proof B.3 *The lemma follows from a simple application of the linearity of expectation to Lemma B.2. More formally, by Lemma B.2 and linearity of expectation we have*

$$\mathop{\mathbb{E}}_{q \leftarrow \mathcal{Q}}[|\mathrm{Vis}_q(T)|] = 1 + 2 \cdot \mathop{\mathbb{E}}_{q \leftarrow \mathcal{Q}}[|\mathrm{Sat}_q(T) \cap I(T)|]$$

which, by applying the linearity of expectation again, equals

$$1 + 2 \cdot \sum_{u \in I(T)} \Pr_{q \leftarrow \mathcal{Q}}[q(u)] = 1 + 2 \cdot \sum_{u \in I(T)} \mathsf{P}_{\mathcal{Q}}(u).$$

Lemma B.3 is particularly useful when one tries to find an optimal tree $T \in \mathcal{T}$ from a family of trees (e.g., OR trees) that minimizes the cost of a random query according to the distribution $q \leftarrow \mathcal{Q}$. The reason is that, in this case, we will have to minimize a summation over evaluations of the function $\mathsf{P}_{\mathcal{Q}}(u) = \Pr_{q \leftarrow \mathcal{Q}}[q(u) = 1]$ for all *internal* newly

constructed nodes $u \in I(T)$. Because of this, $\mathsf{P}_{\mathcal{Q}}(u)$ indeed behaves like the *local cost* that the node u contributes (on average) to the search of a sampled query $q \leftarrow \mathcal{Q}$.

Equivalent cost function. Suppose we are given a set of leaf nodes $L = \{(f_1, \bar{f}_1), \ldots, (f_n, \bar{f}_n)\}$ where $\bar{f}_i \in \{0,1\}^m$ is the vector label of the node f_i. Suppose $\mathcal{Q}$ is a query distribution over monotone Boolean formulas with m variables, and suppose an OR tree T is constructed over leaves L. As we proved in lemma B.3, it holds that $\mathbb{E}_{q \leftarrow \mathcal{Q}}[|\mathrm{Vis}_q(T)|] = 1 + 2 \cdot \sum_{u \in I(T)} \mathsf{P}_{\mathcal{Q}}(u)$ where $\mathsf{P}_{\mathcal{Q}}(u) = \Pr_{q \leftarrow \mathcal{Q}}[q(u) = 1]$ for any distribution $\mathcal{Q}$ over monotone Boolean formulas. Because of this characterization, in this section and for sake of simplicity, we will work with $\sum_{u \in I(T)} \mathsf{P}_{\mathcal{Q}}(u)$ as the cost of the search operation, even though it is the sum of local costs and is only proportional to the actual search cost. In fact, since we can assume each keyword appear in at least one file, it holds that the vector label of the root is $\bar{r} = (1,1,\ldots,1)$, where r is the root of a tree T over leaves $\{f_1, \ldots, f_n\}$. Therefore, we can always skip visiting the root and directly visit its children $\Gamma(r)$. This way, the number of visited nodes will be *exactly* $2 \cdot \sum_{u \in I(T)} \mathsf{P}_{\mathcal{Q}}(u)$.

B.2 Optimality of Algorithm 2 for Single-Keyword and disjunctive Queries

In this subsection we prove theorems about the optimality of our level optimal algorithm both compared to leveled trees as well as general (even unbalanced) trees when the query distribution is uniform over single keyword or when it is an arbitrary distribution over disjunctive queries. We start by defining some notation and then will describe our theorems proving properties about the optimality of our level-optimal algorithm..

Definition B.2 *We define the following notations.*

- $\mathsf{SLC}_{\mathcal{Q}}(T) = \sum_{u \in I(T)} \mathsf{P}_{\mathcal{Q}}(u)$ *(i.e., Sum of Local Costs for L) denotes the sum of the local costs of searching a sampled $q \leftarrow \mathcal{Q}$.*
- $\mathsf{Opt}_{\mathcal{Q}}(L) = \min_{T, L(T) = L} \mathsf{SLC}_{\mathcal{Q}}(T)$ *is the optimal (average) search cost among all (possibly unbalanced)* OR *trees for given leaves L.*
- $\mathsf{OptBal}_{\mathcal{Q}}(L)$ *denotes the minimum cost among all balanced* OR *trees of depth $\lceil \log |L| \rceil$ for a set of given leaves L:*

$$\min_{T \text{ of height } \leq \lceil \log n \rceil, |L| = n, L(T) = L} \mathsf{SLC}_{\mathcal{Q}}(T).$$

- $T_{\mathsf{BF}}(L)$ *and* $T_{\mathsf{LO}}(L)$ *are, in order, the tree that is the result of running our best-first algorithm (Algorithm 1) and level-optimal algorithm (Algorithm 2) over leaf nodes L. Thus,* $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{BF}}(L))$ *and* $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}(L))$ *are the corresponding average search costs.*

Theorem B.1 *Our level-optimal algorithm (Algorithm 2) is optimal up to a logarithmic factor among* all *(even unbalanced) trees when $\mathcal{Q}$ is any distribution over disjunctive queries. Namely, for such $\mathcal{Q}$ it holds that* $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}(L)) = O(\log n \cdot \mathsf{Opt}_{\mathcal{Q}}(L))$.

Proof B.4 *Let T_d be any* OR *tree of depth d over leaf nodes L and let T be an arbitrary* OR *tree with leaf nodes L. Then for any $q \leftarrow Q$, if no file has the keyword in q then we visit exactly one node in both trees, so in the following suppose the keyword in q is in at least one file. In that case, we claim* $\mathsf{SLC}_q(T_d) \leq d \cdot \mathsf{SLC}_q(T)$. *The reason is that if we are searching a disjunctive query q and reach a node u such that $q(u) = 1$, we can confidently say that there is at least one f in the subtree rooted at u that satisfies q. More formally, suppose when we search q the returned output set is $S = L(T) \cap \mathrm{Sat}_q(T)$ for an arbitrary* OR *tree over leaves L. Then any node* satisfying *query q is on the path from some $f \in S$ to the root. Therefore,* $\mathsf{SLC}_q(T_d) \leq |S| \cdot d$. *On the other hand, there are at least $|S|$ nodes satisfying q while searching q over any T. Therefore, when T_d is any balanced tree of depth $d = O(\log n)$, it will have average search cost at most $O(\log n)$ times any (even unbalanced) optimal tree.*

Theorem B.1 is optimal. Note that in Theorem B.1 we *cannot* decrease the approximation factor to $o(\log n)$. For example, consider n files that have disjoint keyword sets and the ith file has 2^{i-1} keywords (and the dictionary size will be $m = 2^n - 1$). In this case, the best-first algorithm solution will be of total cost less than $2^{n+2} \approx 4m$, while the level-optimal solution will be of total cost $2 \cdot (2^n - 1) \cdot \log n \approx 2m \log n$. For this example it holds that $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}(L)) / \mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{BF}}(L)) \geq \Omega(d) = \Omega(\log n)$ when $\mathcal{Q}$ is a single keyword query distribution uniformly distributed over all keywords in the dictionary. Therefore, the gap between $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}})$ and $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{BF}})$ cannot be as

small as $o(\log n)$. On the other hand, our empirical results for larger n (see Section 4) also suggests a close relation between the balanced and unbalanced tree as our best-first and level-optimal algorithms reach close results. Thus, we conjecture that a more general theorem (than our Theorem B.1) holds for a broader class of query distributions.

The next theorem proves a *constant factor* approximation (as opposed to the logarithmic approximation of B.1) when we restrict the comparison to balanced trees.

Theorem B.2 *Suppose $L = \{(f_1, \bar{f}_1), \ldots, (f_n, \bar{f}_n)\}$ is a given set of leaf nodes with their vector labels in $\{0,1\}^m$, and suppose $\mathcal{Q}$ is distributed uniformly over single keyword searches $\{w_1, \ldots, w_m\}$. If $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}(L)) = \alpha \cdot (n-1)$ for constant $\alpha > 1/2$, and if n is even, then Algorithm 2 achieves $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}(L)) \le \frac{\alpha}{2\alpha-1}(\mathsf{OptBal}_{\mathcal{Q}}(L))$.*

Interpretation. Note that $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}(L))$ is always bounded by $n-1$ because it is simply the summation of $\mathsf{P}_{\mathcal{Q}}(u) = \Pr_{q \leftarrow \mathcal{Q}}[q(u) = 1] \in [0,1]$ over all internal nodes $u \in I(T_{\mathsf{LO}})$, and there are exactly $n-1$ such nodes. Theorem B.2 states that if we run our level-optimal algorithm and get T_{LO} that is *dense* in the sense that the number of 1 coordinates in the produced label vectors (in $\{0,1\}^m$) is more than the number of zero coordinates in the vector labels of the internal nodes (i.e., , $\alpha > 1/2$), then our Algorithm 2 indeed achieves a constant approximation of the optimal balanced solution (that gives us $\mathsf{OptBal}_{\mathcal{Q}}(L)$). In other words, either the solution that we get is not that costly (i.e., the cost is at most half of what it could potentially be), or we get a costly solution, but we would also know that this is *inherent* up to a constant approximation. In fact, the approximation factor $\frac{\alpha}{2\alpha-1}$ gets *better* (i.e., goes to 1) if the obtained solution becomes costlier.

We now prove Theorem B.2.

Proof B.5 (Proof of Theorem B.2) *First we need to define some notation.*

Notation. *Recall that $\mathsf{HW}(\bar{v})$ denotes the hamming wait of a vector $\bar{v} \in \{0,1\}^m$. For two vectors $\bar{u}, \bar{v} \in \{0,1\}^m$, by $\mathsf{HD}(\bar{u}, \bar{v})$ we denote $\mathsf{HW}(\bar{u} \oplus \bar{v})$ where $\bar{u} \oplus \bar{v}$ is their componentwise XOR. Also, by $\mathsf{ZW}(\bar{v}) = m - \mathsf{HW}(\bar{v}) = |\{i \mid \bar{v}[i] = 0\}|$ we denote its* zero weight*.*

First we ask the reader to recall the notations $\mathsf{HW}(\cdot), \mathsf{ZW}(\cdot)$, and $\mathsf{HD}(\cdot)$ (for Hamming weight, zero weight, and Hamming distance) defined in Section 2. In the following, $\mathcal{Q}$ refers to the fixed uniform distribution over single keyword search queries, and because L is fixed, we might drop that and simply write $T_{\mathsf{LO}} = T_{\mathsf{LO}}(L)$.

The main idea behind the proof of our Theorem B.2 is to show that our level-optimal algorithm will maximize *the number of* 0 *coordinates in the vector labels of the internal nodes within approximation factor two. Let the total "Hamming weight" of T_{LO} be defined as $\mathsf{HW}(T_{\mathsf{LO}}) = \mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}) \cdot m$. It holds that $\mathsf{HW}(T_{\mathsf{LO}}) = \sum_{u \in I(T_{\mathsf{LO}})} \mathsf{HW}(\bar{u})$, because $\mathsf{P}_{\mathcal{Q}}(u)$ is equal to $\mathsf{HW}(\bar{u})/m$ for every node u and its label $\bar{u}$, and $\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}) = \sum_{u \in I(T_{\mathsf{LO}})} \mathsf{P}_{\mathcal{Q}}(u)$. Now, let the total "zero weight" of T_{LO}, denoted by $\mathsf{ZW}(T_{\mathsf{LO}})$, be defined as $\mathsf{ZW}(T_{\mathsf{LO}}) = m(n-1) - \mathsf{HW}(T_{\mathsf{LO}})$ which is the total number of zero coordinates in the labels of the internal nodes of T_{LO}. Finally, let T_* be the optimal balanced tree that achieves $\mathsf{SLC}_{\mathcal{Q}}(T_*) = \mathsf{OptBal}_{\mathcal{Q}}(L)$.*

Claim B.3 *It holds that $\mathsf{ZW}(T_{\mathsf{LO}}) \ge \mathsf{ZW}(T_*)/2$.*

Before proving the above claim, we show how to use it to prove Theorem B.2.

Proving Theorem B.2 by Claim B.3. *Since it holds that*

$$\mathsf{HW}(T_{\mathsf{LO}}) = \mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}}) \cdot m = \alpha \cdot (n-1)m$$

therefore $\mathsf{ZW}(T_{\mathsf{LO}}) = (1-\alpha)m(n-1)$. Thus by Claim B.3, $\mathsf{ZW}(T_*) \le 2\mathsf{ZW}(T_{\mathsf{LO}}) = 2(1-\alpha)m(n-1)$, and consequently

$$\mathsf{HW}(T_*) = m(n-1) - \mathsf{ZW}(T_*) \ge m(n-1)(1 - 2(1-\alpha)) = (2\alpha - 1)m(n-1).$$

Therefore, we get $\frac{\mathsf{SLC}_{\mathcal{Q}}(T_{\mathsf{LO}})}{\mathsf{OptBal}_{\mathcal{Q}}(L)} \le \frac{\alpha}{2\alpha-1}$ which proves Theorem B.2.

In the following, we will prove Claim B.3. In doing so, without loss of generality, we can assume that every coordinate $i \in [m]$ is "non-redundant" defined as follows: there are at least two leaf nodes f, g such that $\bar{f}[i] = 0$ and $\bar{g}[i] = 1$. That is because if a coordinate is redundant, it would hold that $\bar{f}[i] = 1$ for all leaf nodes f. In that case, we can first remove all the redundant coordinates from all the nodes, prove the claim for the non-redundant case, and then add the redundant coordinates back, while the claim remains true.

Claim B.4 *Let $u \in I(T)$ be any internal node in an arbitrary OR tree T and let $u = \pi(x) = \pi(y)$. It holds that*

$$\mathsf{HW}(\bar{x}) + \mathsf{HW}(\bar{y}) + \mathsf{HD}(\bar{x}, \bar{y}) = 2\mathsf{HW}(\bar{u})$$

Proof B.6 *Consider any coordinate i where $\bar{u}[i] = 1$. Then this coordinate is counted twice in $2\mathsf{HW}(\bar{u})$. However, it also means that $\bar{x}[i] = 1$ or $\bar{y}[i] = 1$. If $\bar{x}[i] = \bar{y}[i] = 1$ then $\mathsf{HD}(\bar{x}, \bar{y})$ contributes for coordinate i, and if exactly one of $\bar{x}[i] = 1$ or $\bar{y}[i] = 1$ holds, then $\mathsf{HD}(\bar{x}, \bar{y})$ also contributes one to the coordinate i.*

Suppose T is an arbitrary OR tree. By summing up the equation of Claim B.4 over all the internal nodes, $\mathsf{HW}(\bar{u})$ for each node appears on the LHS once, except for the root, and $\mathsf{HW}(\bar{u})$ for each internal node appears on the RHS once. If we let $\mathsf{HD}(\bar{u}) = \mathsf{HD}(\bar{x}, \bar{y})$ for $u = \pi(x) = \pi(y)$, then:

$$\sum_{x \in L(T)} \mathsf{HW}(\bar{x}) + \sum_{u \in I(T)} \mathsf{HD}(\bar{u}) = \mathsf{HW}(\bar{r}) + \sum_{u \in I(T)} \mathsf{HW}(\bar{u}) \tag{1}$$

where r is the root node. Since we assume there is no redundant coordinate, for every coordinate i there is at least one internal node w for which the coordinate i is different between the two children of w (i.e., the node w is the first common node of the paths going from the two leaf nodes f, g where $v_f[i] \neq v_g[i]$ to the root). This shows that $\sum_{u \in I(T)} \mathsf{HD}(\bar{u}) \geq m \geq \mathsf{HW}(r)$. Thus, it holds that:

$$\sum_{u \in L(T)} \mathsf{HW}(\bar{u}) \leq \sum_{u \in I(T)} \mathsf{HW}(\bar{u}).$$

Switching to zero weights, we get:

$$\begin{aligned}
\sum_{u \in L(T)} \mathsf{ZW}(\bar{u}) &= m(n-1) - \sum_{u \in L(T)} \mathsf{HW}(\bar{u}) \\
&\geq m(n-1) - \sum_{u \in I(T)} \mathsf{HW}(\bar{u}) \\
&= \sum_{\bar{u} \in I(T)} \mathsf{ZW}(\bar{u}).
\end{aligned}$$

Thus, we conclude that the total zero weights of the leaves are always at least half of the total zero weights:

$$\sum_{v \in L(T)} \mathsf{ZW}(v) \geq \frac{1}{2} \sum_{u \in L(T) \cup I(T)} \mathsf{ZW}(u). \tag{2}$$

Now, we observe that Algorithm 2 maximizes the total Hamming weights of each layer (when $\mathcal{Q}$ is the uniform distribution over single keywords), conditioned on the labels of their children. In particular, assuming n is even and $n/2 = k$, Algorithm 2 minimizes the total Hamming weights of the labels of the parents $\{g_1, \ldots, g_k\}$ of the leaves $\{f_1, \ldots, f_n\}$, which is equivalent to minimizing their zero weight. By applying Equation 2 over tree T' with leaves $\{g_1, \ldots, g_k\}$ we conclude that $\sum_{i \in [k]} \mathsf{ZW}(\bar{g}_i)$ in T_{LO} alone (even without considering the zero weight of the nodes above them) is at least half of the total zero weight of any *balanced tree, and that holds in particular for the the optimal balanced tree T_*. This finishes the proof of Claim B.3 and Theorem B.2.*