Boolean Encrypted Search with Approximate Sets:
Error Propagation, Query Obfuscation, and Threshold-Based Retrieval

Alex Towell

Abstract

We present a comprehensive framework for Boolean encrypted search using approximate sets—space-efficient data structures supporting membership queries with tunable false positive rate ε and false negative rate δ. Each document is represented by a secure index encoding trapdoors derived from plaintext terms, enabling arbitrary Boolean queries (conjunction, disjunction, negation) to be evaluated obliviously on untrusted servers.

Our primary contributions are threefold. First, we establish a *compositional algebra* for error propagation in Boolean queries, proving that conjunctions multiply false positive rates (ε^r for r terms), disjunctions follow complement probability rules, and negation swaps error types—revealing that even when $\delta = 0$ for atomic tests, negated queries necessarily introduce false negatives. Second, we introduce *threshold-based retrieval structures*, a novel approximate-set construction using binned seed values that offers simplicity, cache efficiency, and tunable space-accuracy tradeoffs (e.g., 40% storage at $\varepsilon = 2^{-4}$ versus standard Bloom filter parameters). Third, we formalize k-graph indexes as a *query obfuscation* mechanism that enables randomized query padding, making single-term searches indistinguishable from multi-term queries through combinatorial expansion of term sets.

We derive closed-form expressions for precision and recall as functions of ε, δ, collection size, and query structure, and present a recursive algorithm computing error rates for arbitrary Boolean expressions. Extensions include biword indexes for phrase search and higher-order k-graphs trading storage for enhanced privacy. Our analysis demonstrates that aggressive space compression (false positive rates of 2^{-4} to 2^{-10}) yields near-perfect precision for realistic conjunctive workloads, enabling practical encrypted search systems with 10–100$\times$ storage reduction while preserving query privacy through obfuscation.

Contents

List of Figures

List of Algorithms

1 Introduction

Search systems increasingly operate on infrastructure that cannot be fully trusted with the contents of the documents or the intent of end users. Encrypted search promises the ability to execute familiar retrieval workloads while the server only manipulates encrypted artefacts. Realising that promise for Boolean retrieval requires careful coordination between the query mechanism, the data structure that indexes each document, and the quantitative guarantees that bound the inevitable leakage.

This report focuses on secure indexes constructed from approximate sets. Each index supports membership tests for trapdoors derived from plaintext search keys and exhibits tunable false positive rate ε and false negative rate δ. We examine how these error rates propagate through Boolean query operators and influence classical information retrieval measures. A key insight is that query structure matters: even when atomic membership tests guarantee $\delta = 0$, compound queries involving negation introduce false negatives, fundamentally coupling query semantics to retrieval accuracy.

Our contributions are fourfold:

1. We formalise the Boolean retrieval workflow used throughout the paper and clarify the assumptions placed on search agents, queries, and result sets.
2. We describe how approximate-set secure indexes can be generated from plaintext documents, including algorithmic sketches for query translation and membership testing.

3. We establish a compositional algebra for error propagation in arbitrary Boolean queries, proving that conjunctions multiply false positive rates, disjunctions apply complement probability rules, and negation swaps error types. We present a recursive algorithm that computes false positive and false negative rates for any Boolean expression.
4. We outline practical extensions—such as biword phrase search and k-graph indexes—that broaden the expressiveness of hidden queries without sacrificing the security posture.

The remainder of the document proceeds as follows. Section 2 reviews the information retrieval model and introduces the encrypted search interface. Section 3 elaborates on the construction of secure indexes backed by approximate sets and derives expected precision for conjunctive queries. Section 4 generalises the error analysis to arbitrary Boolean expressions involving conjunction, disjunction, and negation, establishing compositional rules and presenting worked examples. Section 5 discusses the oblivious-set abstraction that underpins the data structure. Finally, Section 6 surveys extensions to the base model that enable richer query semantics including phrase search and k-graph indexes.

2 Information retrieval model

An information retrieval process begins when a search agent submits a query representing an information need. In response, the system returns a subset of confidential objects from the universe $\mathbb{U}$ that satisfy that need.

An Encrypted Search system may support many retrieval paradigms. We focus on Boolean search, a well-understood model described below.

Definition 2.1. *In Boolean search, each object is either* relevant *or* irrelevant *to a query. The query is represented as a bag of search keys, and an object is relevant exactly when it contains every key from the query.*

For a given query, let $\mathbb{R} \subseteq \mathbb{U}$ denote the corresponding result set.

Assumption 2.1. *The information retrieval model is Boolean search.*[1]

To support Boolean search operations, each confidential object only needs to answer membership tests for search keys. This leads naturally to a set-based representation of documents. We distinguish between *true positives* and *false positives* as follows.

Definition 2.2. *A* true positive *is any confidential object in $\mathbb{R}$ in which every search key from the query is present. A* false positive *is any confidential object in $\mathbb{R}$ that fails at least one of those membership tests.*

Classical Boolean search returns all positives and no negatives, thereby ensuring that false positives and false negatives never occur. In the encrypted setting we relax that guarantee in a controlled manner.

2.1 Encrypted search interface

A query submitted to an Encrypted Search system should not be comprehensible to an adversary observing the server.

[1]See Section 6 for simple extensions to the query interface.

Definition 2.3. *A* hidden query *represents the confidential information need of an authorised search agent. It should be interpretable only by that agent and the document owner.*

Hidden queries are realised via cryptographic *trapdoors*.

Definition 2.4 (Trapdoor)**.** *Search agents map each plaintext search key to a cryptographic hash, called a trapdoor.*

A trapdoor for a plaintext search key enables an untrusted system to look up the key inside a confidential data set without revealing the underlying word.

Definition 2.5. *A queryable representation of a confidential object is called a* secure index.

A secure index must satisfy two broad requirements:

1. If the index has bit length n, its encoding should be close to maximally entropic so that it appears indistinguishable from noise.
2. Membership tests derived from authorised queries should succeed with high probability while limiting counterfeit matches.

These requirements motivate the use of approximate-set data structures for secure indexes.

Definition 2.6. *An* approximate set *is a set-like data structure in which false negatives never occur but false positives may appear with a controllable probability.*

We denote by $\mathbb{S}$ the exact set we wish to represent and by $\mathbb{S}^*$ the corresponding approximate set, where $\mathbb{S} \subseteq \mathbb{S}^*$.

Assumption 2.2. *The secure index is an approximate set with false positive rate ε.*

Under this assumption a hidden query may evaluate to an approximate result set $\mathbb{R}^*$ that contains the exact result set $\mathbb{R}$, i.e. $\mathbb{R} \subseteq \mathbb{R}^*$. Precision and recall therefore become random variables parameterised by ε. Other properties such as information disclosure are explored in Section 3.

Trapdoors are produced by a one-way transformation.

Definition 2.7 (Trapdoor)**.** *Given a secret k and a search key x, the trapdoor y is defined as*

$$y \leftarrow \mathrm{h}(x + k)\,. \tag{2.1}$$

By one-way we mean that, given a hash y, finding a preimage x such that

$$y = \mathrm{h}(x) \tag{2.2}$$

is computationally infeasible. A random oracle models this behaviour by uniformly distributing inputs over the output space.

Assumption 2.3. *The trapdoor function approximates a random oracle.*

If a collision between plaintext words x and y were to occur, the terms would be aliased—indistinguishable—within the Encrypted Search system. We simplify the analysis with an additional assumption.

Assumption 2.4. *The codomain of* h *is sufficiently large that collisions between search keys are negligible and may be ignored.*

Plaintext queries are transformed into hidden queries by the following procedure.

Definition 2.8. *Given a secret k and a plaintext query $\mathbb{X}$, the function that transforms plaintext queries into hidden queries is denoted by*

$$\mathbb{Y} \leftarrow \mathsf{HideQuery}(\mathbb{X}, k) \ , \tag{2.3}$$

which applies the trapdoor function from Definition 2.7 to each search key.

The function HideQuerymay be a simple substitution cipher as shown in Algorithm 1. More sophisticated techniques that improve confidentiality are possible; see Section 6 for options.

Algorithm 1: Simple substitution cipher

```
Input:
   $\mathbb{X}$ A plaintext query.
   $k$ The secret key.
Output: $\mathbb{Y}$ is a hidden query, a set of trapdoors.
function HideQuery($\mathbb{X}, k$)
    $\mathbb{Y} \leftarrow \{\mathrm{h}(x + k) \colon x \in \mathbb{X}\}$;
    return $\mathbb{Y}$;
```

The implementation of Containsin the bag-of-words model is given in Algorithm 2.

Algorithm 2: Implementation of Contains

```
Input:
 $\check{\mathbb{D}}$ The secure index approximating a bag-of-words $\mathbb{D}$ for a confidential object.
 $\mathbb{Y}$ The set of trapdoors to test for membership.
Output: Returns TRUEif every trapdoor in $\mathbb{Y}$ tests positively for membership and
        otherwise returns FALSE.
function Contains$\check{\mathbb{D}}$, $\mathbb{Y}$
    foreach $y \in \mathbb{Y}$ do
        if Contains$\check{\mathbb{D}}$, $y$ = FALSE then
            return FALSE;
        end
    end
    return TRUE;
```

3 Secure index model

Each confidential document $\mathbb{D}$ is transformed into a secure index $\check{\mathbb{D}}$ via Algorithm 3. The index stores only trapdoors and is materialised as an approximate set. Because false negatives are impossible, any authorised query that matched the original document continues to match the secure index. False positives, however, occur with probability controlled by ε.

3.1 Approximate result sets

Let $\mathcal{D}$ be the collection of N confidential objects, and let $\mathbb{R} \subseteq \mathcal{D}$ be the *exact* result set for a plaintext query $\mathbb{X}$ whose cardinality we denote by $r = |\mathbb{X}|$. Evaluating the hidden query $\mathbb{Y}$ against

Algorithm 3: Secure index construction with a substitution cipher

```
Input:
$\mathbb{D}$ The bag-of-words representation of a document.
$k$ The secret key.
$\varepsilon$ The target false positive rate.
Output: A secure index that supports encrypted Boolean search over $\mathbb{D}$ with false
        positive rate $\varepsilon$.
1 function MakeSecureIndex$\mathbb{D}$, $\varepsilon$, $k$
2     $\check{\mathbb{D}} \leftarrow \{\mathrm{h}(x+k) \colon x \in \mathbb{D}\}$;
3     return MakeApproxSet$\check{\mathbb{D}}$, $\varepsilon$;
```

the collection yields an *approximate* result set $\mathbb{R}^*$ consisting of every secure index whose membership tests succeed for all trapdoors in $\mathbb{Y}$.

We assume the following independence property, which is satisfied by common approximate-set instantiations (e.g. Bloom filters with optimised hash functions) and captures the behaviour measured in prior empirical work.

Assumption 3.1. *For any secure index that does not contain the trapdoors in $\mathbb{Y}$, the membership tests of distinct trapdoors are independent events. Each individual test has false-positive probability ε.*

Under Assumption 3.1, the probability that a non-relevant document is spuriously admitted to $\mathbb{R}^*$ equals $\varepsilon_q = \varepsilon^r$: all r trapdoors must simultaneously yield false positives. Let F denote the number of false positives. Because there are $N - |\mathbb{R}|$ non-relevant documents, we obtain the binomial model

$$F \sim \mathrm{Binomial}(N - |\mathbb{R}|\,,\, \varepsilon_q)\,. \tag{3.1}$$

Consequently,

$$\mathbb{E}[F] = (N - |\mathbb{R}|)\,\varepsilon_q \quad \text{and} \quad \mathrm{Var}[F] = (N - |\mathbb{R}|)\,\varepsilon_q(1 - \varepsilon_q). \tag{3.2}$$

The approximate result set satisfies $\mathbb{R} \subseteq \mathbb{R}^*$ almost surely, so the expected cardinality of $\mathbb{R}^*$ is

$$\mathbb{E}\big[|\mathbb{R}^*|\big] = |\mathbb{R}| + \mathbb{E}[F] = |\mathbb{R}| + (N - |\mathbb{R}|)\,\varepsilon_q. \tag{3.3}$$

Precision and recall. Classical precision is defined as $\mathrm{Prec} = |\mathbb{R} \cap \mathbb{R}^*| \,/\, |\mathbb{R}^*|$. Because hidden queries do not incur false negatives, the numerator equals $|\mathbb{R}|$. Taking expectations gives the deterministic upper bound

$$\mathbb{E}[\mathrm{Prec}] = \frac{|\mathbb{R}|}{|\mathbb{R}| + (N - |\mathbb{R}|)\,\varepsilon_q}. \tag{3.4}$$

Recall remains identically 1; the encrypted setting preserves completeness while trading a tunable amount of precision for storage and compute efficiency. The expected F_1 score simplifies accordingly:

$$\mathbb{E}[\mathrm{F}_1] = \frac{2}{1 + \left(1 + \frac{N - |\mathbb{R}|}{|\mathbb{R}|}\right)\varepsilon_q}. \tag{3.5}$$

Worked example. Suppose $N = 10^6$ documents contain $|\mathbb{R}| = 3 \times 10^3$ relevant results. With query length $r = 4$ and per-trapdoor false-positive probability $\varepsilon = 2^{-10}$, the compound probability is $\varepsilon_q = 2^{-40} \approx 9.1 \times 10^{-13}$. The expected number of false positives is therefore $\mathbb{E}[F] \approx 10^{-6}$, so precision remains effectively 1. Even when ε is relaxed to 2^{-4} the expected number of spurious matches is ≈ 0.96, demonstrating how aggressively the system can tune ε before precision begins to degrade.

These expressions make explicit how design choices for the secure index (hash family, filter width, load factor) influence observable retrieval quality. In Section 3.2 we connect ε to concrete data-structure parameters, enabling end-to-end sizing of encrypted search deployments.

3.2 Storage cost and parameter selection

Let $n = |\mathbb{D}|$ denote the number of trapdoors recorded in a secure index. For memory-efficient approximate sets such as Bloom filters, the relationship between the false-positive rate ε and the bit budget m is well known:

$$m = -\frac{n \ln \varepsilon}{(\ln 2)^2}\,. \tag{3.6}$$

The optimal number of hash functions is $k_{\text{opt}} = (m/n)\ln 2$. Together, Equation (3.6) and k_{opt} provide closed-form guidance for configuring secure indexes: choosing m fixes the achievable ε, and conversely, targeting a specific ε determines the necessary bit budget.

Perfect-hash filters and quotient filters obey similar asymptotic behaviour. They typically achieve a leading term of $n \log_2(1/\varepsilon)$ bits per element, plus lower-order overheads that capture metadata and cache alignment. Throughout this work we adopt the simplifying approximation

$$extbitspertrapdoor \approx \log_2 \frac{1}{\varepsilon}, \tag{3.7}$$

which aligns with the dominant term of both Bloom filters and perfect-hash filters. Substituting Equation (3.7) into the combinatorial constructions in Section 6 recovers the storage estimates cited for biwords and k-graphs.

Given a fixed storage budget B bits per document, the allowable false-positive rate follows directly from Equation (3.6):

$$\varepsilon = 2^{-Bn^{-1}\ln 2}. \tag{3.8}$$

Coupling this expression with Equation (3.4) yields a simple sizing workflow: pick B to meet latency or cost constraints, compute the implied ε, and verify that the resulting precision satisfies the retrieval objectives for representative workloads.

3.3 Threshold-based retrieval structures

While Bloom filters are the canonical approximate membership structure, we present an alternative construction that is particularly well-suited to secure indexes: *threshold-based retrieval structures*. This data structure achieves comparable false positive rates to Bloom filters while offering simpler construction and potentially superior space efficiency for certain parameter regimes.

3.3.1 Construction algorithm

Let $\mathbb{D} = \{t_1, t_2, \ldots, t_m\}$ be the set of m trapdoors to store. The construction proceeds as follows:

The membership test is straightforward:

Algorithm 4: Threshold-based retrieval structure construction

```
Input:
$\mathbb{D}$ Set of $m$ trapdoors.
$\varepsilon$ Target false positive rate.
$b$ Number of bins.
$M$ Maximum hash value (typically $2^{32}$ or $2^{64}$).
Output: Array of seeds $S = [s_1, s_2, \dots, s_b]$.
function MakeThresholdStructure$\mathbb{D}$, $\varepsilon$, $b$, $M$
    $N \leftarrow \lfloor \varepsilon \cdot M \rfloor$ ;                       // Threshold value
    Partition $\mathbb{D}$ into $b$ bins: $B_1, B_2, \dots, B_b$ ;   // Each bin has $\approx m/b$ items
    foreach bin $B_i$ do
        $s \leftarrow 0$;
        while $s < M$ do
            success $\leftarrow$ TRUE;
            foreach $t \in B_i$ do
                if $\mathrm{h}(t + s) \geq N$ then
                    success $\leftarrow$ FALSE;
                    break;
                end
            end
            if success then
                $S[i] \leftarrow s$;
                break;
            end
            $s \leftarrow s + 1$;
        end
        if $s = M$ then
            return failure ;                          // Could not find valid seed
        end
    end
    return $S$;
```

3.3.2 Complexity analysis

Time complexity. For a bin with l items, each trapdoor independently hashes below the threshold N with probability ε. The probability that all l trapdoors simultaneously succeed is ε^l. The number of seed trials follows a geometric distribution with success probability $p = \varepsilon^l$, yielding an expected number of trials:

$$\mathbb{E}[\text{trials per bin}] = \frac{1}{\varepsilon^l}. \tag{3.9}$$

If we partition m trapdoors into b bins, each bin contains approximately m/b items. The expected construction time is

$$\mathcal{O}\left(\frac{bm}{b\varepsilon^{m/b}}\right) = \mathcal{O}\left(\frac{m}{\varepsilon^{m/b}}\right) \text{ hash operations.} \tag{3.10}$$

Query time is $\mathcal{O}(b)$ hash operations (one per bin).

Algorithm 5: Threshold-based membership query

```
Input:
$S$ Array of seeds.
$q$ Query trapdoor.
$\varepsilon$ False positive rate.
$M$ Maximum hash value.
Output: TRUEif $q$ tests positive, FALSEotherwise.
1 function ThresholdQuery$S$, $q$, $\varepsilon$, $M$
2     $N \leftarrow \lfloor \varepsilon \cdot M \rfloor$;
3     foreach seed $s_i \in S$ do
4         if $\mathrm{h}(q + s_i) < N$ then
5             return TRUE;
6         end
7     end
8     return FALSE;
```

Space complexity. The structure stores one seed per bin. If seeds are drawn from $[0, M)$, each requires $\log_2 M$ bits, yielding total storage:

$$b \cdot \log_2 M \text{ bits.} \tag{3.11}$$

For $M = 2^{32}$, this is $32b$ bits. Compare to a Bloom filter storing m items with false positive rate ε, which requires $-m \ln \varepsilon/(\ln 2)^2 \approx 1.44m \log_2(1/\varepsilon)$ bits. When $b \ll 1.44m \log_2(1/\varepsilon)/32$, the threshold structure is more space-efficient.

False negative guarantees. The construction succeeds in finding a valid seed for bin i if at least one seed $s \in [0, M)$ satisfies the threshold condition for all items in that bin. The probability of failure after trying all M seeds is

$$P(\text{failure}) = (1 - \varepsilon^l)^M, \tag{3.12}$$

where l is the bin size. For small failure probability δ, we require

$$M > \frac{\log \delta}{\log(1 - \varepsilon^l)} \approx \frac{\log(1/\delta)}{\varepsilon^l}, \tag{3.13}$$

where the approximation holds when $\varepsilon^l \ll 1$.

Example 1 [Parameter selection] Consider $m = 100$ trapdoors with $\varepsilon = 2^{-4}$ and $M = 2^{32}$. To ensure false negative probability $< 10^{-6}$ per bin, we need bin size l such that

$$2^{32} > \frac{\log(10^6)}{(2^{-4})^l} = \log(10^6) \cdot 2^{4l}. \tag{3.14}$$

Solving: $4l \log 2 < 32 \log 2 - \log(\log(10^6))$, yielding $l \lesssim 7.7$. Thus bins should contain at most 7 items. With $b = \lceil 100/7 \rceil = 15$ bins, the structure requires $15 \times 32 = 480$ bits.

A Bloom filter for the same parameters requires $\approx 1.44 \times 100 \times 4 = 576$ bits, making the threshold structure $\approx 17\%$ more space-efficient in this regime.

3.3.3 Comparison to Bloom filters

Threshold-based retrieval structures offer several advantages for secure indexes:

- **Simplicity**: No bit array, no multiple hash functions—just seeds and threshold tests.
- **Space efficiency**: When document sizes are moderate and bins are small, storing b seeds can be more compact than Bloom filter bit arrays.
- **Tunable construction cost**: Smaller bins reduce construction time (lower $1/\varepsilon^{m/b}$) at the cost of more seeds.
- **Cache efficiency**: Sequential seed testing can be more cache-friendly than scattered bit-array accesses.

The primary trade-off is construction time, which grows exponentially in bin size. Careful selection of b based on m, ε, and M ensures practical construction while maintaining space efficiency.

3.4 Adversarial considerations

The random-oracle assumption from Assumption 2.3 means that trapdoors reveal no information about the underlying plaintext keys beyond equality under the shared secret. Moreover, the approximate-set representation reduces the leakage footprint to access-pattern observations and the occasional false positive.

The primary information leakage in this model comes from two sources. First, *access patterns*: which indexes are tested for which trapdoors. An adversary observing the server learns query cardinality, term co-occurrence across documents, and potentially query frequency distributions that enable statistical attacks. Techniques described in Section 6—particularly k-graph indexes (Section 6.2)—provide a principled mechanism for obfuscating these patterns by collapsing multi-term queries into single trapdoor lookups.

Second, *boolean result leakage*: each document returns a plaintext boolean value (true/false) indicating whether it satisfies the query. This reveals which documents are relevant, exposing the structure of the result set. The same trapdoor machinery used for search terms can be applied to boolean values themselves: instead of returning plaintext true/false, the system can return *boolean trapdoors* encoding the result under operations (AND, OR, NOT). By treating every value type—including booleans—as obfuscated, the system approaches a more complete oblivious computation model. When extended recursively to all intermediate values and operations, this framework becomes Turing-complete, enabling arbitrary computations over encrypted data without revealing intermediate or final results to the server. We do not explore this extension in this paper, but note that it represents a natural and effective path toward fuller oblivious computation systems.

The system's tolerance for elevated false positive rates (demonstrated in Section 3.1) makes the substantial storage overhead of obfuscation techniques practical, enabling meaningful query privacy enhancements in adversarial settings.

4 Boolean query algebra with error propagation

The preceding analysis focused on conjunctive queries, where every search key in $\mathbb{X}$ must be present for a document to match. Real-world search systems, however, support richer Boolean expressions combining conjunction, disjunction, and negation operators. This section generalises the error analysis to arbitrary Boolean queries and establishes compositional rules for computing false positive and false negative rates.

A crucial observation motivates this generalisation: even when the underlying approximate set guarantees zero false negatives on atomic membership tests ($\delta = 0$), compound queries involving negation can exhibit non-zero false negative rates. For instance, the query $\neg(a \land b)$ applied to a document containing both a and b should evaluate to false. If both trapdoors test positive (as expected with $\delta = 0$), the negated conjunction correctly returns false. However, if the approximate set *could* have false negatives, at least one trapdoor might test negative, causing $\neg(a \land b)$ to incorrectly return true—a false positive for the negated query, or equivalently, the system fails to identify a true negative, which manifests as a kind of induced error.

We therefore adopt a general framework where atomic membership tests exhibit:

- False positive rate ε (item not in set, tests positive)
- False negative rate δ (item in set, tests negative)

Throughout this section we derive expressions parameterised by both ε and δ. The special case $\delta = 0$ recovers the original model from Section 3, while non-zero δ accommodates approximate sets with weaker guarantees or systems that deliberately introduce false negatives for adversarial obfuscation.

4.1 Error rates for atomic and compound queries

We begin by formalising error rates for Boolean queries. Let Q denote a Boolean query expression built from atomic search keys and the operators $\land$ (AND), $\lor$ (OR), and $\neg$ (NOT).

Definition 4.1 (False Positive Rate). *The* false positive rate *FPR*(Q) *is the probability that query* Q *evaluates to true when the ground truth is false.*

Definition 4.2 (False Negative Rate). *The* false negative rate *FNR*(Q) *is the probability that query* Q *evaluates to false when the ground truth is true.*

For an atomic query q corresponding to a single trapdoor, the base error rates are:

$$\text{FPR}(q) = \varepsilon, \qquad \text{FNR}(q) = \delta. \tag{4.1}$$

We now state the fundamental theorems governing error propagation through Boolean operators.

Theorem 4.1 (Conjunction error rates). *Let* $Q = q_1 \land q_2 \land \cdots \land q_r$ *be a conjunctive query over* r *independent atomic queries. Then*

$$FPR(Q) = \varepsilon^r \tag{4.2}$$

and

$$FNR(Q) = 1 - (1 - \delta)^r. \tag{4.3}$$

Proof. A false positive occurs when none of the r search keys appear in the document (ground truth is false), yet all r membership tests return positive. Under independence, this probability is ε^r.

A false negative occurs when all r search keys appear (ground truth is true), but at least one membership test fails. The probability that all tests succeed is $(1 - \delta)^r$, so the complement gives $1 - (1 - \delta)^r$. □

Theorem 4.2 (Disjunction error rates). *Let* $Q = q_1 \lor q_2 \lor \cdots \lor q_r$ *be a disjunctive query over* r *independent atomic queries. Then*

$$FPR(Q) = 1 - (1 - \varepsilon)^r \tag{4.4}$$

and

$$FNR(Q) = \delta^r. \tag{4.5}$$

Proof. A false positive occurs when none of the r search keys appear (ground truth is false), yet at least one membership test returns positive. The probability that all tests correctly return negative is $(1-\varepsilon)^r$, so the complement gives $1-(1-\varepsilon)^r$.

A false negative occurs when at least one search key appears (ground truth is true), but all membership tests fail. The probability is δ^r. □

Theorem 4.3 (Negation error rates)**.** *Let $Q = \neg q$ be a negated atomic query. Then*

$$\mathit{FPR}(\neg q) = \delta \tag{4.6}$$

and

$$\mathit{FNR}(\neg q) = \varepsilon. \tag{4.7}$$

Proof. For $\neg q$, the ground truth is false when q is present. A false positive occurs when the membership test for q incorrectly returns negative (with probability δ), causing $\neg q$ to evaluate to true.

Conversely, the ground truth is true when q is absent. A false negative occurs when the membership test incorrectly returns positive (with probability ε), causing $\neg q$ to evaluate to false.

Thus negation *swaps* error types: $\text{FPR}(\neg q) = \text{FNR}(q)$ and $\text{FNR}(\neg q) = \text{FPR}(q)$. □

Corollary 4.3.1 (Negated conjunction)**.** *For $Q = \neg(q_1 \wedge \cdots \wedge q_r)$,*

$$\mathit{FPR}(Q) = 1-(1-\delta)^r, \qquad \mathit{FNR}(Q) = \varepsilon^r. \tag{4.8}$$

Proof. Apply Theorem 4.3 to the conjunction from Theorem 4.1. □

Corollary 4.3.2 (Negated disjunction)**.** *For $Q = \neg(q_1 \vee \cdots \vee q_r)$,*

$$\mathit{FPR}(Q) = \delta^r, \qquad \mathit{FNR}(Q) = 1-(1-\varepsilon)^r. \tag{4.9}$$

Proof. Apply Theorem 4.3 to the disjunction from Theorem 4.2. □

Summary of error rates. Table 1 summarises the false positive and false negative rates for the five basic Boolean query patterns. When $\delta = 0$, the FNR column simplifies: conjunctions and disjunctions exhibit zero false negatives, while negations swap all error probability to the FNR term.

Table 1: Error rates for Boolean query patterns with r independent atomic queries.

Query type	FPR	FNR
$q_1 \wedge \cdots \wedge q_r$	ε^r	$1-(1-\delta)^r$
$q_1 \vee \cdots \vee q_r$	$1-(1-\varepsilon)^r$	δ^r
$\neg q$	δ	ε
$\neg(q_1 \wedge \cdots \wedge q_r)$	$1-(1-\delta)^r$	ε^r
$\neg(q_1 \vee \cdots \vee q_r)$	δ^r	$1-(1-\varepsilon)^r$

4.2 Compositional rules for arbitrary Boolean expressions

The theorems in Section 4.1 handle single operators applied to atomic queries or uniform combinations of r atoms. Real queries, however, nest operators in arbitrary ways: $(a \wedge b) \vee (c \wedge \neg d)$, for instance. We now present compositional rules that compute error rates recursively for any Boolean expression.

Let Q_1 and Q_2 be arbitrary subqueries with known error rates $(\mathrm{FPR}(Q_1), \mathrm{FNR}(Q_1))$ and $(\mathrm{FPR}(Q_2), \mathrm{FNR}(Q_2))$. Define compound queries:

$$Q_\wedge = Q_1 \wedge Q_2, \tag{4.10}$$

$$Q_\vee = Q_1 \vee Q_2, \tag{4.11}$$

$$Q_\neg = \neg Q_1. \tag{4.12}$$

Theorem 4.4 (Compositional conjunction)**.**

$$FPR(Q_\wedge) = FPR(Q_1) \cdot FPR(Q_2), \tag{4.13}$$

$$FNR(Q_\wedge) = 1 - \big(1 - FNR(Q_1)\big)\big(1 - FNR(Q_2)\big). \tag{4.14}$$

Proof. The conjunction $Q_1 \wedge Q_2$ is false (ground truth) when at least one subquery is false. A false positive requires both subqueries to incorrectly return true, yielding the product $\mathrm{FPR}(Q_1) \cdot \mathrm{FPR}(Q_2)$.

The conjunction is true when both subqueries are true. It correctly evaluates to true when both succeed, with probability $(1 - \mathrm{FNR}(Q_1))(1 - \mathrm{FNR}(Q_2))$. The false negative probability is the complement. □

Theorem 4.5 (Compositional disjunction)**.**

$$FPR(Q_\vee) = 1 - \big(1 - FPR(Q_1)\big)\big(1 - FPR(Q_2)\big), \tag{4.15}$$

$$FNR(Q_\vee) = FNR(Q_1) \cdot FNR(Q_2). \tag{4.16}$$

Proof. The disjunction $Q_1 \vee Q_2$ is false when both subqueries are false. A false positive occurs when at least one subquery incorrectly returns true. The probability that both correctly return false is $(1 - \mathrm{FPR}(Q_1))(1 - \mathrm{FPR}(Q_2))$; the complement is the false positive rate.

The disjunction is true when at least one subquery is true. A false negative requires both to incorrectly return false, yielding $\mathrm{FNR}(Q_1) \cdot \mathrm{FNR}(Q_2)$. □

Theorem 4.6 (Compositional negation)**.**

$$FPR(\neg Q_1) = FNR(Q_1), \tag{4.17}$$

$$FNR(\neg Q_1) = FPR(Q_1). \tag{4.18}$$

Proof. Negation swaps truth values. When Q_1 is true (so $\neg Q_1$ should be false), $\neg Q_1$ incorrectly returns true exactly when Q_1 incorrectly returns false—i.e. with probability $\mathrm{FNR}(Q_1)$. Similarly, when Q_1 is false, $\neg Q_1$ incorrectly returns false when Q_1 incorrectly returns true, with probability $\mathrm{FPR}(Q_1)$. □

These three theorems enable recursive computation of error rates for any Boolean expression. We formalise the procedure in Algorithm 6.

Algorithm 6: Recursive error-rate computation for Boolean queries

```
Input:
$Q$ A Boolean query expression (abstract syntax tree).
$\varepsilon$ Base false positive rate for atomic membership tests.
$\delta$ Base false negative rate for atomic membership tests.
Output: A pair $(\mathrm{FPR}(Q), \mathrm{FNR}(Q))$ giving the false positive and false negative rates for $Q$.
function compute_error_rates($Q$, $\varepsilon$, $\delta$)
    if $Q$ is an atomic query then
        return $(\varepsilon, \delta)$;
    else if $Q = Q_1 \wedge Q_2$ then
        $(\mathrm{fpr}_1, \mathrm{fnr}_1) \leftarrow$ compute_error_rates($Q_1, \varepsilon, \delta$);
        $(\mathrm{fpr}_2, \mathrm{fnr}_2) \leftarrow$ compute_error_rates($Q_2, \varepsilon, \delta$);
        $\mathrm{fpr} \leftarrow \mathrm{fpr}_1 \cdot \mathrm{fpr}_2$;
        $\mathrm{fnr} \leftarrow 1 - (1 - \mathrm{fnr}_1)(1 - \mathrm{fnr}_2)$;
        return $(\mathrm{fpr}, \mathrm{fnr})$;
    else if $Q = Q_1 \vee Q_2$ then
        $(\mathrm{fpr}_1, \mathrm{fnr}_1) \leftarrow$ compute_error_rates($Q_1, \varepsilon, \delta$);
        $(\mathrm{fpr}_2, \mathrm{fnr}_2) \leftarrow$ compute_error_rates($Q_2, \varepsilon, \delta$);
        $\mathrm{fpr} \leftarrow 1 - (1 - \mathrm{fpr}_1)(1 - \mathrm{fpr}_2)$;
        $\mathrm{fnr} \leftarrow \mathrm{fnr}_1 \cdot \mathrm{fnr}_2$;
        return $(\mathrm{fpr}, \mathrm{fnr})$;
    else if $Q = \neg Q_1$ then
        $(\mathrm{fpr}_1, \mathrm{fnr}_1) \leftarrow$ compute_error_rates($Q_1, \varepsilon, \delta$);
        return $(\mathrm{fnr}_1, \mathrm{fpr}_1)$;
```

4.3 Worked examples

We illustrate Algorithm 6 with concrete Boolean queries, assuming $\varepsilon = 0.01$ and $\delta = 0$.

Example 2 [Disjunctive query] Consider $Q = a \vee b \vee c$. Each atomic query has $(\mathrm{FPR}, \mathrm{FNR}) = (0.01, 0)$. Applying Theorem 4.2 with $r = 3$:

$$\mathrm{FPR}(Q) = 1 - (1 - 0.01)^3 = 1 - 0.970299 = 0.0297, \tag{4.19}$$

$$\mathrm{FNR}(Q) = 0^3 = 0. \tag{4.20}$$

The disjunction raises the false positive rate to approximately 3% while maintaining perfect recall.

Example 3 [Mixed conjunction and disjunction] Consider $Q = (a \wedge b) \vee c$. First compute the conjunction:

$$\mathrm{FPR}(a \wedge b) = 0.01^2 = 0.0001, \tag{4.21}$$

$$\mathrm{FNR}(a \wedge b) = 1 - (1 - 0)^2 = 0. \tag{4.22}$$

Now apply disjunction with c:

$$\mathrm{FPR}(Q) = 1 - (1 - 0.0001)(1 - 0.01) = 1 - 0.98991 = 0.01009, \tag{4.23}$$

$$\mathrm{FNR}(Q) = 0 \cdot 0 = 0. \tag{4.24}$$

The compound query exhibits a false positive rate slightly above 1%, reflecting the dominant contribution from the atomic term c.

Example 4 [Negation introducing false negatives] Consider $Q = \neg(a \vee b)$ with $\varepsilon = 0.01$ and $\delta = 0$. By Corollary 4.3.2:

$$\text{FPR}(Q) = \delta^2 = 0, \tag{4.25}$$

$$\text{FNR}(Q) = 1 - (1-\varepsilon)^2 = 1 - 0.9801 = 0.0199. \tag{4.26}$$

Despite $\delta = 0$ for atomic tests, the negated disjunction exhibits a 2% false negative rate. This occurs because when neither a nor b appears (so $\neg(a \vee b)$ should be true), a false positive on either a or b causes the disjunction to incorrectly return true, making the negation return false.

Example 5 [De Morgan equivalence] De Morgan's law states $\neg(a \vee b) \equiv \neg a \wedge \neg b$. We verify that both formulations yield identical error rates. For $\neg a$ we have $(\text{FPR}, \text{FNR}) = (\delta, \varepsilon) = (0, 0.01)$, and similarly for $\neg b$. Applying conjunction:

$$\text{FPR}(\neg a \wedge \neg b) = 0 \cdot 0 = 0, \tag{4.27}$$

$$\text{FNR}(\neg a \wedge \neg b) = 1 - (1 - 0.01)^2 = 0.0199, \tag{4.28}$$

matching Example 4. This confirms that logically equivalent queries have identical error characteristics under our compositional semantics.

4.4 Implications for query optimisation and system design

The compositional error framework reveals several actionable insights:

Query rewriting for error minimisation. Logically equivalent Boolean expressions can exhibit dramatically different error rates. For instance, a conjunctive query $a_1 \wedge \cdots \wedge a_r$ has FPR $= \varepsilon^r$, which decreases exponentially with r. Disjunctive queries, conversely, have FPR $\approx r\varepsilon$ for small ε, growing linearly. Users or query optimisers can exploit this asymmetry: rewriting disjunctions as unions of conjunctive subqueries may reduce aggregate false positive rates when combined with client-side filtering.

Negation as an error-type swap. Theorems 4.3 and 4.6 show that negation exchanges FPR and FNR. Systems built on approximate sets with $\delta = 0$ sacrifice perfect recall when queries involve negation. If maintaining high recall is critical, queries should avoid or minimise negation depth. Alternatively, systems could implement exact membership structures for a subset of high-value terms, enabling precise negation without FNR degradation.

Predictable precision and recall. Given ε, δ, and a query Q, Algorithm 6 computes FPR(Q) and FNR(Q) before execution. These rates directly determine expected precision and recall. Extending the analysis from Section 3.1, suppose the exact result set has cardinality $|\mathbb{R}|$ and the collection contains N documents. The expected number of false positives is

$$\mathbb{E}[F] = (N - |\mathbb{R}|) \cdot \text{FPR}(Q), \tag{4.29}$$

and the expected number of false negatives is

$$\mathbb{E}[\mathrm{FN}] = |\mathbb{R}| \cdot \mathrm{FNR}(Q). \tag{4.30}$$

Expected precision and recall become

$$\mathbb{E}[\mathrm{Prec}] = \frac{|\mathbb{R}|\,(1 - \mathrm{FNR}(Q))}{|\mathbb{R}|\,(1 - \mathrm{FNR}(Q)) + (N - |\mathbb{R}|)\mathrm{FPR}(Q)}, \tag{4.31}$$

$$\mathbb{E}[\mathrm{Rec}] = 1 - \mathrm{FNR}(Q). \tag{4.32}$$

When $\delta = 0$ and Q is a pure conjunction or disjunction (no negation), Equation (4.32) yields $\mathbb{E}[\mathrm{Rec}] = 1$, recovering the perfect-recall guarantee from Section 3.

Space–accuracy tradeoffs for compound queries. The storage cost of an approximate set is proportional to $\log_2(1/\varepsilon)$ (see Section 3.2). Designers can select ε to balance space and error rates. Table 1 shows that conjunctions benefit from aggressive compression (small ε), since FPR scales as ε^r. Disjunctions and negations, however, degrade more gracefully, suggesting that mixed-mode indexes—high precision for conjunctive subqueries, moderate precision for disjunctive expansions—may be cost-effective.

5 The oblivious set abstract data type

Boolean search treats each document as a bag-of-words set. Replacing the exact set with an *oblivious set* retains the ability to answer membership queries while hiding the underlying vocabulary. An oblivious set stores only trapdoors and therefore exposes neither the plaintext terms nor their multiplicities.

Instead of storing just the search keys that appear in the document, we can precompute a richer set of candidate trapdoors that are expected to be relevant. This design turns the secure index into a catalogue of potential matches rather than a verbatim mirror of the document. The abstraction is particularly useful when combined with approximate-set implementations such as Bloom filters or perfect hash filters: the index inherits the compactness guarantees of the data structure while maintaining the same trapdoor interface.

6 Extensions to the query model

While the core model focuses on bag-of-words Boolean queries, the same machinery supports richer semantics with modest adjustments.

6.1 Biword model

The function Containsadmits phrase search by lifting the secure index to operate on adjacent bigrams. Suppose the search keys in a document are ordered as $y_1 y_2 \ldots y_m$. We add each unigram y_j to the oblivious set as before and, additionally, insert every unique bigram $y_{j-1}y_j$ for $j = 2, \ldots, m$.

This strategy does not require the search provider to learn that a phrase query is being issued, although the correlation among bigrams can reveal structural hints. Implementing the secure index as an oblivious set, consider a document with m search keys. In the worst case where no key repeats, the index contains m unigrams and $m-1$ bigrams, for a total of $n_{extbi} = 2m-1$ trapdoors. Applying Equation (3.7), the storage requirement becomes

$$(2m-1)\log_2 \frac{1}{\varepsilon} \text{ bits.} \tag{6.1}$$

If repetitions reduce the number of unique bigrams to $b < m-1$, the expression simply substitutes $m+b$ for $2m-1$. The approximation therefore scales linearly with the vocabulary richness captured by the biword index.

The implementation of ContainsSequencein the biword model is given in Algorithm 7.

Algorithm 7: Biword phrase search via ContainsSequence

```
Input:
s  A secure index storing unigram and bigram trapdoors.
$\vec{y}$  A sequence of $m$ trapdoors representing the phrase to test.
Output:  Returns TRUEif every bigram in $\vec{y}$ tests positively for membership; otherwise
         returns FALSE.
1 function ContainsSequence$s, \vec{y}$
2     $\mathbb{Y} \leftarrow \{y_{j-1} + y_j : j \in [2, m)\}$;
3     return Contains$s$, $\mathbb{Y}$;
```

Algorithm 8: Biword secure index construction

```
Input:
$k$  The secret key.
$\vec{d}$  The confidential document.
$\varepsilon$  The false positive rate.
Output:  A biword secure index that supports approximate phrase matching.
1 function MakeBiwordSecureIndex$\vec{d}, k, \varepsilon$
2     $\mathbb{D} \leftarrow \{d_{j-1} + d_j : j \in [2, m]\}$;
3     return MakeSecureIndex$\mathbb{D}$, $\varepsilon$, $k$;
```

6.2 Digraphs, trigraphs, and k-graphs for query obfuscation

The primary goal of k-graphs is *query obfuscation*: entangling multiple search terms together to obscure term correlations and distort the observable query distribution away from the plaintext distribution. A k-graph secure index stores trapdoors for all combinations of up to k search keys, tagged with their intended Boolean operation (AND or OR). This enables multi-term queries to be collapsed into single trapdoor lookups, concealing query structure from the server.

Motivation: obfuscating term correlations and query distributions. In the baseline unigram model, every query consists of individual term lookups: $\{\mathrm{h}(a+k), \mathrm{h}(b+k), \mathrm{h}(c+k)\}$. An adversary observing the server learns:

1. The exact query cardinality (three terms)
2. Co-occurrence patterns of these three terms across the document collection
3. The distribution of queries, which mirrors the plaintext query distribution (just with substituted trapdoors)

4. Statistical correlations enabling frequency analysis or query reconstruction attacks

A k-graph with $k \geq 3$ replaces this with a *single* compound trapdoor: $\mathrm{h}(\{a, b, c\}_{\text{AND}} + k)$ for conjunction or $\mathrm{h}(\{a, b, c\}_{\text{OR}} + k)$ for disjunction. The adversary observes only that *some* compound trapdoor was queried, without learning:

- How many distinct terms are encoded (could be 1 to k)
- Which specific terms are involved
- Whether this represents a genuine k-term query or a padded lower-cardinality query
- The true distribution of plaintext queries

This fundamentally breaks the isomorphism between encrypted and plaintext query distributions. Where the baseline model applies a simple substitution cipher to queries (preserving all distributional properties), k-graphs introduce *entanglement*: terms are cryptographically combined in ways that obscure their individual identities and relationships.

Storage cost and the role of elevated false positive rates. Suppose a document contains n unique search keys. A k-graph stores trapdoors for all subsets of size at most k, tagged with Boolean operations (AND, OR). For each subset of size $j \in [1, k]$, we store both the AND-tagged and OR-tagged trapdoor (and potentially NOT-tagged, though space constraints often limit this). Thus the number of trapdoors is:

$$2 \sum_{j=1}^{k} \binom{n}{j} = \mathcal{O}(n^k) \quad \text{trapdoors.} \tag{6.2}$$

Using Equation (3.7), the storage requirement is

$$2 \left(\sum_{j=1}^{k} \binom{n}{j} \right) \log_2 \frac{1}{\varepsilon} = \Theta\Big(n^k \log_2 \varepsilon^{-1} \Big) \text{ bits.} \tag{6.3}$$

For specific values of k (with both AND and OR tags):

- $k = 1$ **(baseline)**: $\mathcal{O}(n)$ trapdoors, $2n \log_2(1/\varepsilon)$ bits
- $k = 2$ **(digraphs)**: $\mathcal{O}(n^2)$ trapdoors, $n(n+1) \log_2(1/\varepsilon)$ bits
- $k = 3$ **(trigraphs)**: $\mathcal{O}(n^3)$ trapdoors, $2\left(n + \frac{n(n+1)}{2} + \frac{n(n+1)(n+2)}{6}\right) \log_2(1/\varepsilon)$ bits
- $k = n$ **(power set)**: $\mathcal{O}(2^n)$ trapdoors, $2(2^n - 1) \log_2(1/\varepsilon)$ bits

Example 6 [Concrete storage costs] Consider a typical document with $n = 100$ unique search keys (e.g., a 500-word document with Zipf-distributed vocabulary). Storage requirements for various configurations (including both AND and OR tags):

k	ε	Trapdoors	Storage
1	2^{-10}	200	2.0 KB
1	2^{-4}	200	0.8 KB
2	2^{-10}	10,100	101 KB
2	2^{-4}	10,100	40.4 KB
3	2^{-10}	343,400	3.4 MB
3	2^{-4}	343,400	1.37 MB

The digraph case ($k = 2$) increases storage by $\sim 50\times$ over baseline, while the trigraph case ($k = 3$) inflates it by $\sim 1700\times$. This exponential growth makes elevated ε essential: using $\varepsilon = 2^{-4}$ instead of $\varepsilon = 2^{-10}$ reduces storage by $2.5\times$, making the trigraph index feasible at 1.37 MB per document instead of 3.4 MB.

For a smaller document with $n = 20$ unique keys (e.g., an email or short article), storage becomes more manageable:

k	ε	Trapdoors	Storage
2	2^{-4}	420	1.68 KB
3	2^{-4}	3,080	12.3 KB
4	2^{-4}	17,710	70.8 KB

Here even $k = 4$ remains practical, enabling strong obfuscation for up to 4-term queries with modest overhead.

This substantial storage overhead is precisely why the system's tolerance for elevated false positive rates becomes advantageous. Recall from Section 4 that conjunctive queries with r terms achieve FPR $= \varepsilon^r$, which decreases exponentially. If atomic trapdoors use $\varepsilon = 2^{-4}$, a three-term query has effective FPR $\approx 2^{-12} \approx 2.4 \times 10^{-4}$, yielding near-perfect precision even with aggressive compression. The analysis in Section 3.1 demonstrated that precision remains acceptable even at $\varepsilon = 2^{-4}$ for realistic workloads. By accepting modestly elevated ε, we obtain order-of-magnitude storage reductions that make k-graph indexes practical:

$$\text{storage}(\varepsilon = 2^{-4}) = \frac{4}{10} \cdot \text{storage}(\varepsilon = 2^{-10}). \tag{6.4}$$

Inherent cardinality obfuscation. A fundamental property of k-graphs is that they provide *inherent cardinality obfuscation*: any query with at most k distinct terms collapses to a single trapdoor lookup, making it impossible for the server to determine the true number of terms. A 1-term query $\mathrm{h}(\{a\}_{\text{AND}} + k)$ and a 3-term query $\mathrm{h}(\{a, b, c\}_{\text{AND}} + k)$ both appear as single compound trapdoor lookups to the server. The adversary observes only that *some* j-graph trapdoor (for $j \in [1, k]$) was queried, but learns nothing about which j or which specific terms are encoded.

This stands in stark contrast to the baseline unigram model, where query cardinality is immediately observable: a 3-term query requires three separate trapdoor lookups, directly revealing the query size.

Distribution obfuscation via term repetition. While k-graphs already mask cardinality, the client can further obscure the query distribution by exploiting multiple encodings of the same logical query. For a single search term a, the following trapdoors are all semantically equivalent (each matches documents containing a):

- $\mathrm{h}(\{a\}_{\text{AND}} + k)$ (unigram encoding)
- $\mathrm{h}(\{a, a\}_{\text{AND}} + k)$ (repeated bigram encoding)
- $\mathrm{h}(\{a, a, a\}_{\text{AND}} + k)$ (repeated trigram encoding)
- etc., up to $\mathrm{h}(\{a, a, \ldots, a\}_{\text{AND}} + k)$ with k repetitions

Because conjunction is idempotent ($a \wedge a \equiv a$), all encodings yield identical results. However, each produces a *cryptographically distinct trapdoor* due to the hash function. By randomly selecting among these k encodings each time term a is queried, the client distorts the observable query distribution:

- The baseline model preserves the plaintext query distribution (just with substituted trapdoors)
- With term repetition, the same logical query appears as k different trapdoors over time
- The server observes a flattened, artificially uniform distribution that bears no statistical relationship to the underlying plaintext query patterns

This breaks frequency analysis attacks: even if term a dominates the plaintext query distribution, its k encoded representations appear with $1/k$ the frequency each, obscuring which trapdoors correspond to popular versus rare queries.

The error rates remain unchanged: from Section 4, $\mathrm{FPR}(a \wedge a \wedge \cdots \wedge a) = \mathrm{FPR}(a)$ and $\mathrm{FNR}(a \wedge a \wedge \cdots \wedge a) = \mathrm{FNR}(a)$ due to idempotence. Distribution obfuscation via term repetition adds no accuracy degradation.

Injecting fake trapdoors for additional noise. For disjunctive queries, the client can inject *fake trapdoors*—randomly sampled hash values that correspond to no actual search terms—to further obfuscate query structure. Consider a 2-term disjunctive query $\{a, b\}$ submitted to a $k = 4$ index. The client can augment it with noise:

$$\mathrm{h}(\{a, b, r_1, r_2\}_{\mathrm{OR}} + k), \tag{6.5}$$

where r_1 and r_2 are random hash values. Because disjunction semantics mean "match if *any* term is present," the fake trapdoors are unlikely to affect query results: they test positive only with probability ε (the base false positive rate). For typical $\varepsilon = 2^{-4}$ to 2^{-10}, the probability that a fake trapdoor causes a spurious match is negligible.

This technique provides *controlled noise injection*: the server observes a 4-term OR query but cannot determine which terms are genuine versus fake. Unlike conjunctive queries (where every term must match, so fake terms would break the query), disjunctions tolerate noise gracefully. The expected number of false positives introduced by m fake trapdoors is approximately $(N - |\mathbb{R}|) \cdot m\varepsilon$ additional spurious documents, which remains small for reasonable m and ε.

Fake trapdoor injection is particularly effective when combined with term repetition: the client can pad a disjunctive query with both repeated genuine terms (for distribution obfuscation) and random fake terms (for cardinality inflation beyond k genuine terms), creating maximum uncertainty about query structure.

Security versus performance tradeoff. While k-graphs do reduce query round-trips (a query with p terms requires $\lceil p/k \rceil$ trapdoor lookups instead of p), this performance benefit is secondary. The dominant cost in encrypted search is typically index transfer and client-side decryption, not the number of membership tests. The *primary motivation is security*:

- **Inherent cardinality concealment**: Any query with $\leq k$ distinct terms becomes a single trapdoor lookup, making it impossible to distinguish 1-term from k-term queries.
- **Term correlation hiding**: Co-occurrence patterns of search keys become opaque to the server, as multi-term queries are cryptographically entangled.
- **Distribution obfuscation**: Term repetition provides k distinct encodings per query, flattening the observable distribution and breaking the isomorphism with plaintext query patterns.
- **Frequency analysis resistance**: Popular queries appear as multiple distinct trapdoors with $1/k$ the frequency each, obscuring which trapdoors correspond to common versus rare queries.
- **Temporal unlinkability**: The same logical query appears as different trapdoors across time when using randomized encodings, preventing the server from linking repeated queries.

- **Noise injection compatibility**: OR queries can incorporate fake trapdoors with minimal accuracy impact, further distorting observable query patterns.

Together, these mechanisms trade storage complexity (exponential in k) and time complexity (construction cost grows with k) for substantial reductions in information leakage. The system's tolerance for elevated ε makes this tradeoff practical.

Practical deployment considerations. System designers should select k based on adversarial threat models, query patterns, and storage budgets:

- $k = 1$ **(baseline)**: Minimal storage, maximum leakage. Query distribution mirrors plaintext distribution (modulo substitution). Suitable when the server is semi-trusted or queries are low-sensitivity.
- $k = 2$ **(digraphs)**: Moderate storage ($\sim 50\times$ overhead), conceals pairwise term correlations. Enables 2-term queries as single trapdoors and provides 2 distinct encodings per term for distribution flattening.
- $k = 3$ **(trigraphs)**: Substantial storage ($\sim 1700\times$ overhead), strong obfuscation. Supports up to 3-term queries as single lookups, provides 3 encodings per term, and enables meaningful fake trapdoor injection for OR queries. Practical when $\varepsilon = 2^{-4}$ is acceptable.
- $k \geq 4$: Exponential storage growth, near-maximal obfuscation. Justifiable for high-value/high-sensitivity workloads with small document vocabularies (e.g., $n \leq 20$ unique terms).
- $k = n$ **(full power set)**: Complete obfuscation—every possible query becomes a single trapdoor lookup, and every term has n distinct encodings. Practical only for very small vocabularies or documents.

The key insight is that k-graphs and elevated ε work synergistically: the system's ability to maintain acceptable precision at $\varepsilon = 2^{-4}$ (due to exponential FPR decay in conjunctive queries) makes the substantial storage overhead tolerable. Combined with term repetition and fake trapdoor injection, k-graphs enable meaningful query privacy enhancements by breaking the distributional link between encrypted and plaintext query patterns.