

# Simple Low-Overhead Communication-Efficient String Reconciliation and Edit Distance Algorithms

Michael T. Goodrich\*, Gonzalo Navarro<sup>†</sup>, and Claire A. To\*

\*University of California, Irvine

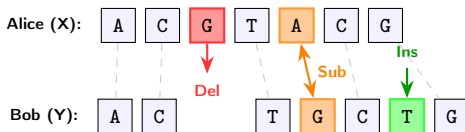
<sup>†</sup>University of Chile

33rd International Symposium on String Processing and  
Information Retrieval  
October 2026

# Motivation: Distributed DNA Comparison

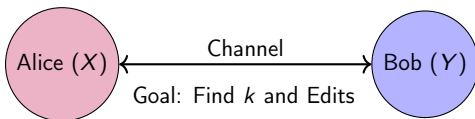
- **Genomic Data:**  
Highly similar sequences (99.9% identical).
- **Task:** Alice sends Bob the edits to transform  $Y$  into  $X$  (or vice versa).
- **Goal:** Communicate only  $k$  edits, not  $n$  characters.

## Types of Edits in String Reconciliation



## Problem Statement: Distributed Edit Distance

- **Given:** String  $X$  (Alice) and  $Y$  (Bob) of length  $\approx n$
- **Find:** Edit distance  $k = L(X, Y)$
- **Constraints:**
  - Parties do not share memory or CPUs.
  - Minimize bits exchanged (Communication Complexity).
  - Minimize CPU cycles (Computational Overhead).



# Prior Work

## Orlitsky (1991): [1]

- **Comm:**  $O(k \log n)$  bits (Optimal)
- **Overhead:**  $n^{O(k)}$  (Exponential!)
- *Issue:* Too slow for large  $k$

## Modern Approaches

(2012-2016): [2, 3, 4, 5]

- **Comm:**  $O(k^2 \log n)$  bits
- **Overhead:**  $O(n \cdot \text{polylog}(n))$
- *Issue:* High constants, complex embeddings

**Goal:** Keep communication low and achieve linear overhead.

# Our Contributions

## 1 General Algorithm (Fast-Sliding):

- **Input:** Any strings (worst-case)
- **Comm:**  $O(k^2 \log n)$  bits
- **Overhead:**  $O(n + k^2)$  time

## 2 Specialized Algorithm (IBLT-Chunking):

- **Input:** Real-world data (e.g., DNA, English text)
- **Comm:**  $O(k \log^5 n)^*$  bits
- **Overhead:**
  - Known upper bound  $k$ :  $O(n)$  time
  - Unknown upper bound  $k$ :  $O(n \log k)$  time

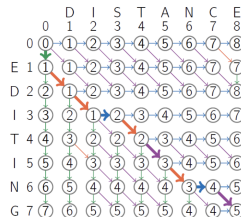
*Methods succeed with high probability, at least  $1 - 1/n^c$ , for  $c \geq 1$ .*

# Classical Edit Distance (Dynamic Programming)

Recurrence for edit distance between  $X[1..i]$  and  $Y[1..j]$ :

$$L(i, j) = \min \begin{cases} L(i-1, j) + 1 & (\text{del}) \\ L(i, j-1) + 1 & (\text{ins}) \\ L(i-1, j-1) + \delta(x_i, y_j) & (\text{match/sub}) \end{cases}$$

- If  $k$  is small, the path stays near the main diagonal.
- For large  $k \geq \sqrt{n / \log_{|\Sigma|} n}$ , we exchange  $X$  and  $Y$  with  $O(n \log |\Sigma|)$  bits and compute  $L(X, Y)$  locally.

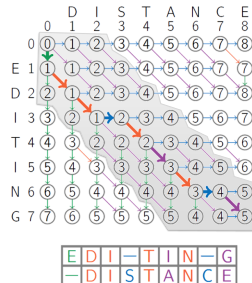


|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| E | D | I | - | T | I | N | - | G |
| - | D | I | S | T | A | N | C | E |

# The "Sliding" Concept

## Landau-Myers-Schmidt: [6]

- Don't compute every cell.
- "Slide" down diagonals as long as characters match ( $X[i] == Y[j]$ ).
- A "slide" costs  $O(1)$  time with Suffix Trees.



## Distributed Problem

Alice doesn't have  $Y$ . Bob doesn't have  $X$ .

They can't communicate their Suffix Trees since the cost is  $\Theta(n)$ !

# Rabin-Karp Polynomial Rolling Hash [7]

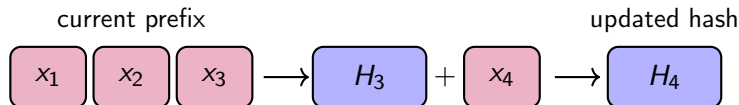
To binary search for the longest common prefix, we need to efficiently compare substrings.

$$rh(x_i x_{i+1} \dots x_j) = (x_i b^{j-i} + x_{i+1} b^{j-i-1} + \dots + x_j b^0) \pmod{p}$$

(where  $b$  is a random base and  $p$  is a large prime)

**Inductive formula:**

$$rh(x_1 x_2 \dots x_i) = rh(x_1 x_2 \dots x_{i-1})b + x_i$$



$$H_4 = H_3 b + x_4$$

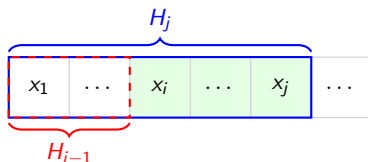
# Comparing Substrings with Prefix Hashes

Precompute prefix hashes:

$$H_i = rh(x_1 x_2 \dots x_i)$$

Then the hash of **any substring**  $x_i \dots x_j$  can be obtained in  $O(1)$ :

$$rh(x_i x_{i+1} \dots x_j) = (H_j - H_{i-1} b^{j-i+1}) \bmod p$$



# Our Solution: Fast-Sliding with Rolling Hashes

To simulate "sliding" without sending strings:

- 1 **Rolling Hash:** Computed and updated in  $O(1)$ .
- 2 **Wavefront:** How far we can slide on diagonal  $d$  with  $h$  edits.

$$M^h(d) = \text{Slide} \left( d, \max \begin{cases} M^{h-1}(d+1) + 1 & (\text{ins}) \\ M^{h-1}(d) + 1 & (\text{sub}) \\ M^{h-1}(d-1) & (\text{del}) \end{cases} \right)$$

**Diagonal  $d$  of  $M^{h-1}$**   
(Substitution)  
+1 Row

**Diagonal  $d+1$  of  $M^{h-1}$**   
(Insertion)  
+1 Row

**Diagonal  $d-1$  of  $M^{h-1}$**   
(Deletion)  
+0 Row

**TARGET:  $M^h(d)$**   
 $\max \begin{cases} \text{Up} + 1 \\ \text{Diag} + 1 \\ \text{Left} + 0 \end{cases}$

## Walkthrough: Setup

|       |  |   |   |   |   |   |   |   |
|-------|--|---|---|---|---|---|---|---|
|       |  | s | i | t | t | i | n | g |
| Start |  |   |   |   |   |   |   |   |
| k     |  |   |   |   |   |   |   |   |
| i     |  |   |   |   |   |   |   |   |
| t     |  |   |   |   |   |   |   |   |
| t     |  |   |   |   |   |   |   |   |
| e     |  |   |   |   |   |   |   |   |
| n     |  |   |   |   |   |   |   |   |

**The  $M$  Table ( $M^h[d]$ )**  
*Stores max row index reached on diagonal  $d$  with  $h$  edits.*

| $d$ | -1        | 0         | 1         |
|-----|-----------|-----------|-----------|
| -1  | $-\infty$ | $-\infty$ | $-\infty$ |

Let  $X = \text{kitten}$ ,  $Y = \text{sitting}$ .

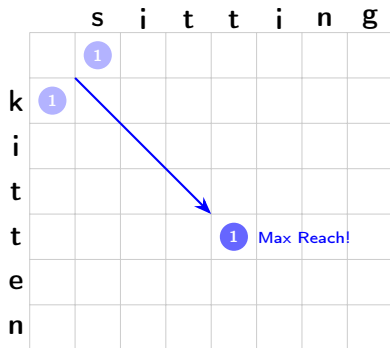
Walkthrough: Wave  $h = 0$ 

|   |                         | s | i | t | t | i | n | g |
|---|-------------------------|---|---|---|---|---|---|---|
| 0 | Mismatch ( $k \neq s$ ) |   |   |   |   |   |   |   |
| k |                         |   |   |   |   |   |   |   |
| i |                         |   |   |   |   |   |   |   |
| t |                         |   |   |   |   |   |   |   |
| t |                         |   |   |   |   |   |   |   |
| e |                         |   |   |   |   |   |   |   |
| n |                         |   |   |   |   |   |   |   |

 $M$  Table Update ( $h = 0$ )

| $d$ | -1 | 0 | 1 |
|-----|----|---|---|
| 0   | -  | 0 | - |

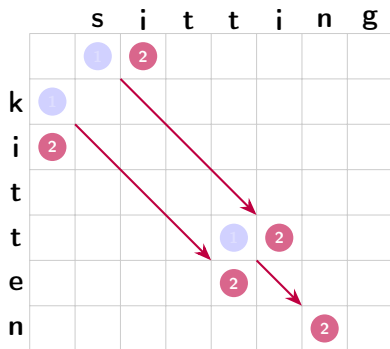
$M^0[0] = \text{Slide}(0, 0) = 0$   
 (Start at 0, slide 0 steps because 'k'  $\neq$  's')

Walkthrough: Wave  $h = 1$  $M$  Table Update ( $h = 1$ )

| $d$ | -1 | 0 | 1 |
|-----|----|---|---|
| 0   | -  | 0 | - |
| 1   | 1  | 4 | 0 |

Key Calculation for  $d = 0$ :

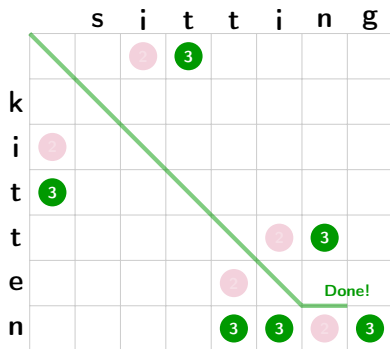
$$\begin{aligned}
 M^1[0] &= \\
 \text{Slide}(0, \max(M^0[-1], M^0[0] + \\
 &\quad 1, M^0[1])) \\
 &= \text{Slide}(0, 0 + 1) = \text{Slide}(0, 1) \\
 &= 4 \text{ (Matches 'itt')}
 \end{aligned}$$

Walkthrough: Wave  $h = 2$  $M$  Table Update ( $h = 2$ )

| $d$ | -2 | -1 | 0 | 1 | 2 |
|-----|----|----|---|---|---|
| 1   | -  | 1  | 4 | 0 | - |
| 2   | 2  | 5  | 6 | 4 | 0 |

Key Calculation for  $d = 0$ :

$$\begin{aligned}
 M^2[0] &= \text{Slide}(0, M^1[0] + 1) \\
 &= \text{Slide}(0, 5) \\
 &= 6 \text{ (Matches 'n')}
 \end{aligned}$$

Walkthrough: Wave  $h = 3$  (Finish) $M$  Table Update ( $h = 3$ )

| $d$ | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
|-----|----|----|----|---|---|---|---|
| 2   | -  | 2  | 5  | 6 | 4 | 0 | - |
| 3   | 3  | 6  | 6  | 6 | 6 | 4 | 0 |

Extend  $M^2[0] = 6 \rightarrow M^3[1] = 6$ .

Target (6, 7) reached!

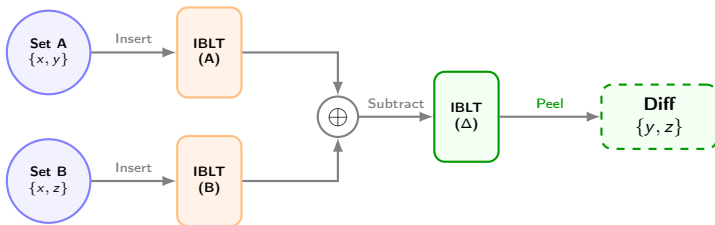
Edit distance = 3.

# Analysis

- **$M$  table:**  $O(k^2)$  cells.
- **Slide cost:**
  - ① **Doubling-Binary Search** (initial approach):  
 $O(\log n)$  tests  $\times 3 \log n$  bits =  
 $O(k^2 \log^2 n)$  bits w.h.p.
  - ② **Noisy Binary Search** (improvement):  $24 \log n$ -bit rolling hash  
 $+ 6 \log n$  tests w.h.p.  $\times 4$  bits/test =  
 $O(k^2 \log n)$  bits w.h.p.
- **Overhead:**  $O(n)$  hash constr.  $+ O(k^2 \log n)$  comparisons.

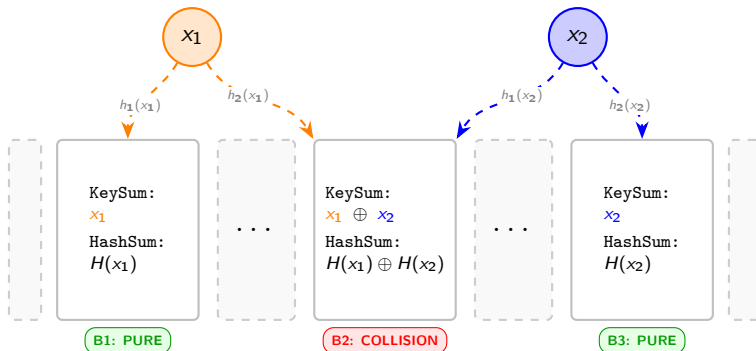
# Invertible Bloom Lookup Tables (IBLT) [8]

- **IBLT:** A hash-based, space-efficient probabilistic data structure that stores a set of items.
  - ① **Insertion/Deletion:** XOR elements into  $\lambda$  buckets.
  - ② **Subtraction:**  $IBLT(A) \oplus IBLT(B) = IBLT(A \Delta B)$ .
  - ③ **Peeling:** Items can be recovered (invertible) from the structure if the number of elements is small relative to the table size.



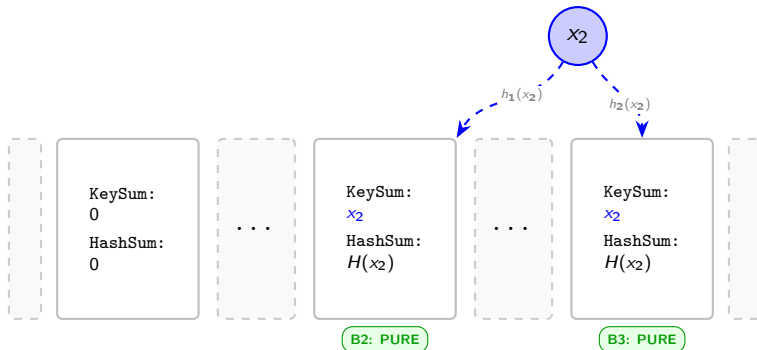
# IBLT Insertion

Inserting items  $x_1$  and  $x_2$  with  $\lambda = 2$  hash functions:

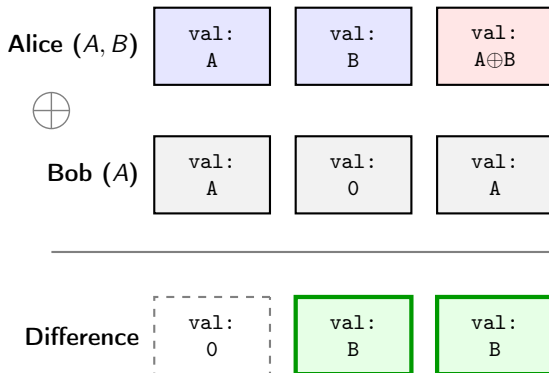


# IBLT Deletion

Removing  $x_1$  makes  $B2$  pure in the *peeling* process:

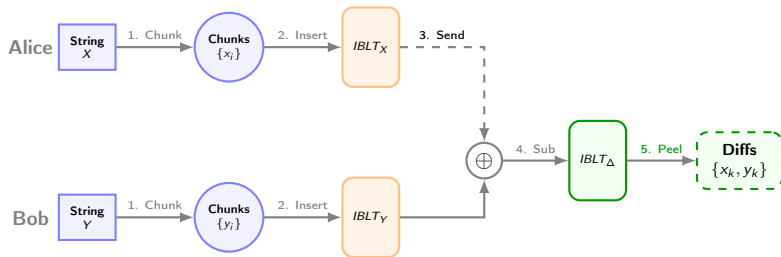


# IBLT Subtraction



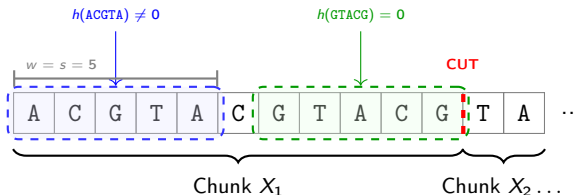
# IBLT-Chunking Algorithm: Overview

- 1 **Chunking:** Divide strings  $X$  and  $Y$  into chunks.
- 2 **Encode:** Insert chunks into IBLTs of size  $O(\kappa \log \eta)$ .
- 3 **Exchange:** Alice sends  $IBLT_X$  to Bob.
- 4 **Subtract:** Bob computes  $IBLT_{diff} = IBLT_X \oplus IBLT_Y$ .
- 5 **Recover:** Bob peels  $IBLT_{diff}$  to find missing/added chunks.



# Content-Defined Chunking (CDC) [9, 10]

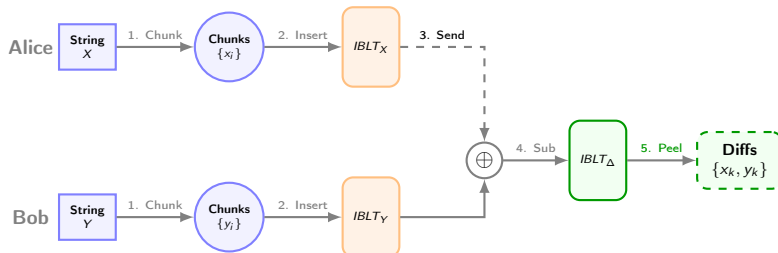
- **Goal:** Divide string  $X$  into  $r$  variable-length chunks.
- **Mechanism:** Sliding window size  $w$  with rolling hash  $h$ .
  - Min chunk size  $s$ .
  - Chunks must be unique, so  $w = s = \max\{d(T_X), d(T_Y)\} + 1$ .
  - Max depth of suffix tree  $d(T_X) = O(\log |X|)$  w.h.p.
- **Cut Condition:**  $h(x_i \dots x_{i+w-1}) = 0$ .
  - Max chunk size is  $O(s + \log |X|)$  w.h.p.



# Encoding and Reconciliation

- **Problem:** Chunks form a set, so order is lost.
- **Solution:** Encode consecutive checksums pairs.

$$S(X) = \underbrace{\{H(X_i) \parallel H(X_{i+1}) \mid i = 1, \dots, r-1\}}_{\text{Key}}$$



- **Reconstruct:** Exchange corresponding differing decoded pairs.

# Analysis

- **IBLT cost:**  $O(\kappa \log \eta)$  cells  $\times O(\log n)$  bits/cell
  - Choose  $0 < \eta \leq n^c$  s.t. decoding  $\kappa$  elements succeeds with probability  $1 - 1/\eta$ .
- **Reconstruction cost:**  $O(ks^2 \log n((s + \log n) \log |\Sigma|))$  bits
  - $\kappa = O(ks^2 \log n)$  differing chunks pairs w.h.p.
  - $O(s + \log n)$  characters/chunk +  $\log |\Sigma|$  bits/character
- **Total cost:** When  $s = O(\log n)$ ,

$$O(k \log^5 n) \text{ bits w.h.p.}$$

- **Overhead:**  $O(n)$  + exponential search for  $k$  if unknown.

# Results

## Theorem (1)

*Fast-Sliding algorithm computes the edit distance in  $O(k^2 \log n)$  bits of communication w.h.p. with  $O(n + k^2)$  time overhead.*

## Theorem (2)

*IBLT-Chunking algorithm runs in  $O(n)^*$  time overhead using  $O(k s^2 \log n ((s + \log n) \log |\Sigma| + \log \eta \log n))$  bits of communication with probability of at least  $1 - 1/\eta$ .*

## References

- [1] A. Orlitsky, “Interactive communication: Balanced distributions, correlated files, and average-case complexity,” in IEEE Symp. on Foundations of Computer Science (FOCS), pp. 228–238, 1991.
- [2] H. Jowhari, “Efficient communication protocols for deciding edit distance,” in European Symp. on Algorithms (ESA), pp. 648–658, 2012.
- [3] D. Chakraborty, E. Goldenberg, and M. Koucký, “Streaming algorithms for embedding and computing edit distance in the low distance regime,” in ACM Symp. on Theory of Computing (STOC), pp. 712–725, 2016.

## References (contd.)

- [4] D. Belazzougui, “Efficient deterministic single round document exchange for edit distance,” arXiv preprint arXiv:1511.09229, 2015.
- [5] D. Belazzougui and Q. Zhang, “Edit distance: Sketching, streaming, and document exchange,” in IEEE Symp. on Foundations of Computer Science (FOCS), pp. 51–60, 2016.
- [6] G. M. Landau, E. W. Myers, and J. P. Schmidt, “Incremental string comparison,” SIAM J. on Computing, vol. 27, no. 2, pp. 557–582, 1998.
- [7] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” IBM Journal of Research and Development, vol. 31, no. 2, pp. 249–260, 1987.

## References (contd.)

- [8] M. T. Goodrich and M. Mitzenmacher, “Invertible Bloom lookup tables,” 2015.
- [9] B. Chapuis, B. Garbinato, and P. Andritsos, “Throughput: A key performance measure of content-defined chunking algorithms,” in IEEE 36th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW), pp. 7–12, 2016.
- [10] W. Xia, X. Zou, H. Jiang, Y. Zhou, C. Liu, D. Feng, Y. Hua, Y. Hu, and Y. Zhang, “The design of fast content-defined chunking for data deduplication based storage systems,” IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 9, pp. 2017–2031, 2020.

$$+ h\{a_n\}^k \gamma 0U$$