Slate: 分散ログ同期のための追記最適化マークルツリー

鷹見虎男

2025 年 12 月 22 日

概要

大規模分散システムにおいて、分散するレプリカ間でのトランザクションログの迅速な修復と同期は、可用性・整合性・障害回復の性能に大きく影響する。現実的な分散ログは追記偏重のアクセス特性を持ち、その同期性能は分岐 (フォーク) の検出と修復の効率に大きく依存する。一方で、従来の方法はそのようなログ上で末尾からの線形探索で分岐を検出するため大規模なデータセットに対して非効率であり、適用可能な局面は限定的である。本論文は、大きな履歴遡及が頻繁に発生する状況でも効率的に機能する、追記最適化と時間局所性に基づくをアクセス特性を持つ Merkle Tree である *Slate*（Stratified Hash Tree）を提案する。本論文ではまた、Slate において最新エントリから任意の位置のデータに到達するまでの I/O 回数分布がハミング距離に従うことを示し、分布の傾向により I/O が最新近傍で優先的に削減されることを理論化する。これに基づき、I/O がコスト支配的な一般的な状況において、$O(\log n)$ I/O での分岐位置の同定と $O(n)$ I/O での同期転送を満たす、暗号論的に確立された安全性に基づく修復手順を与える。さらに、証明サイズと比較ラウンド数について上界を導き、追記偏重のワークロードでの現実的な期待コストが最新データからのハミング距離に支配される非対称な性能特性を示す。

Rust の実装を用いた評価では、従来の Merkle Tree 構造に比べ、最新付近に偏る差分同期で転送量・探索 I/O・キャッシュミスが一貫して低減することを確認した。Slate は分散トランザクションログ、非同期レプリケーションログ、ブロックチェーン、および Git に類するハッシュグラフの分岐検出と修復に適用可能であり、追記最適化・時間局所性・認証性を両立する実用的基盤を提供する。

Keywords: Stratified Hash Tree, Slate, 分散ログ, 追記最適化, 時間的局所性, 分岐検出, 部分転送, 認証付きデータ構造

Slate はデータ列を追記順に層として保存し、各エントリが過去の Merkle Tree への指数間隔の参照を持つことで、二次記憶装置に最適化された追記性能と、最近追加されたデータのアクセスがより少ない I/O 回数の傾向を持つ高速なアクセスを提供する。

1 INTRODUCTION

1.1 動機

多くの大規模分散システムでは、レプリカ間の順序付き更新ログの複製が可用性・整合性・障害回復の性能に依存する。分散ログは現実的な多くのワークロードで追記偏重 (append-heavy) であり [1]、さらに最新近傍に参照が偏る時間的局所性を持つ。ログ末尾の確定していない (未コミット) データは、レプリカ間の自然な遅延やネットワーク分断、一時的な故障、構成変更により、複数のレプリカで異なる (分岐する) 可能性がある。このような状況において、レプリカ間の各ログの分岐を効率的に検出し、修復する効率が、システムのコミットまでのレイテンシや回復時間 (RTO/RPO) に直接影響する。

Merkle Tree [2] またはハッシュツリーは順序付き集合を暗号論的な厳密さの下で効率的に比較・検証することができるデータ構造である。この検証を部分木へ再帰的に適用することで、差異の有無のみならず、順序付き集合のどこが異なるかを効率的に特定することができる。Merkle Tree の亜種である History Tree [3] は、追記専用を前提とした動的な構造を持ち、第三者が発行済みの証明書を監査するための CT (Certificate Transparency) [4] で現実のシステムで稼働している。

ログを History Tree 構造に適用することで効率的な分岐検出を行うことができるが、History Tree は証明書のような信頼性の高いデータの改ざん検出や発行証跡として使用することに焦点を当てており、この構造を追記最適化する試みや、ログの分岐検出やその修復を行う議論は我々の知る限り行われていない。

本研究は History Tree を分散ログの分岐検出と修復の用途に拡張し、二次記憶装置に対して追記最適化された直列化構造と、最新に近いエントリをより効率的に参照できる時間局所性 (temporal locality) を持つデータ構造である **Slate** (Stratified Hash Tree) を提案する。そして、本論文では Slate において最新エントリから任意の葉に到達するまでの I/O 回数がハミング距離に従うことを示し、その上限を示すことで、最新近傍において I/O 回数が優先的に削減されることを示す。

1.2 問題設定

本論文は単調増加するインデックス $i \in \mathbb{N}$ によって識別される値の列であるログ $D = (v_1, v_2, \ldots, v_n)$ を想定する。ログは追記 (append) と切り詰め (truncate) のみによって変更され、個別の値が更新されることはない。2 つの非ビザンチンレプリカ A と B が保持するそれぞれのログを D_A と D_B とし、両者の**最長共通接頭辞長** (longest common prefix; LCD) を $d = \mathrm{LCD}(D_A, D_B)$ と定義する。本文の目的は次の二点である:

1. **インデックス参照** (index reference): 想定するアクセスモデルは、**時間局所性** (temporal locality) に基づき、最新に近い値に統計的に偏ると仮定する。形式的には $\delta = n - i$ に対して $\Pr[\mathrm{get}(i)] = f(\delta)$ が δ の単調減少関数 (例えば幾何分布、Zipf 近似、指数減衰など) である。
2. **分岐点検出** (fork detection): レプリカ間のインタラクション数と転送量を最小化して、2 つのログ D_A と D_B の LCD 長 d を特定する。言い換えると、D_A と D_B で異なる値が最初に現われる位置 $d+1$ を (存在すれば) 特定する。
3. **修復** (repair): 検出した d を用いて、片方のログの接尾辞をもう片方のログに転送して整合 (同期) する。これは、例えば D_A を $(v_1, v_2, \ldots, v_d)$ に切り詰め、他方の D_B の接尾辞 $(v_{d+1}, \ldots)$ を転送して追加する。

これらは、追記偏重の大規模ログ運用下で分岐と再同期が頻発する環境において、最小のインタラクションと I/O で LCD の特定と修復を実現することを目的とする。

各値 v_i に適用可能な暗号学的ハッシュ関数 h が定義されている。本論文では、セキュリティパラメータ λ に対して計算量的に原像困難性、第二原像困難性、衝突困難性を満たし、任意の多項式時間攻撃者の成功確率が $\mathsf{negl}(\lambda)$ であるような、標準的なハッシュ関数族 $H_\lambda : \{0,1\}^* \to \{0,1\}^\lambda$ を想定する。H に属するドメイン分離されたハッシュ関数を $h_A, h_B, \ldots$ のように表記する。

この設定のもと、我々は（1）**最新から過去への探索を対数段数で保証**し、（2）**距離の確率分布が二項分布に従うこと**を示し、（3）**分岐検出・部分転送・修復**の各手順に対する**計算／通信上界**を与える。特に、追記偏重ワークロードでは差分が最新近傍に集中するため、期待コストが**最新側のハミング距離**で支配される非対称な性能特性が得られる。

1.3 本研究の貢献

本研究は、効率的な分岐検出 (差異位置検出) が可能な追記最適化されたデータ構造である Slate を提案する。Slate は以下に述べる技術的な貢献により、既存の Merkle Tree およびその亜種である History Tree の手法を拡張する:

1. **追記最適化ハッシュツリーの導入**: 従来の Merkle Tree が静的なデータセットに対する完全性検証に特化していたのに対し、本研究では追記操作に最適化された動的なログ構造への拡張を実現した。これにより、History Tree と類似する成長パターンを持ちながら、追記に適した構成を明確に定義し、効率的な差分検出と直列化を可能にする新しいハッシュツリーの構造を確立した。
2. **時間局所性に基づくアクセス効率の最適化**: ハミング距離の上限 $c_u(x) \sim \log(n-x)$ と下限 $c_\ell(x) \sim h - \log x$ を数学的に定義し、この境界に従い、最新近傍で I/O 効率が高く、最悪ケースでも通常の二分木探索と同等な木構造を開発した。この結果、最新データへのアクセスが $O(1)$ で、最新から離れるにつれて $O(\log(n-x))$ で増加し、最悪ケースでも $O(\log n)$ 回の I/O 操作で完了する非対称アクセス特性を開発し、ログの分岐点の特定を従来の $O(n)$ から $O(\log n)$ に改善した。
3. **部分永続的データ構造の効率的なエンコード**: 部分永続化の効率的なバージョン階層を 2 進数展開として自然にエンコードする手法を確立し、Driscoll ら [5] の一時的な永続化手法で必要とされていた複雑なポインタ管理やバージョンスタンプを排除した。各世代 i で追加されるノードを単一のエントリ e_i に集約ことで、シンプルかつ効率的な部分永続化を実現した。
4. **並行アクセスの安全性保証**: 永続的データ構造の特徴を利用し、SWMR (Single-Write/Multi-Read) 並行実行モデルに基づく安全な並行アクセスを可能にした。これは、スナップショット一貫性により、複数の読み取り操作が単一の追記操作と競合なく並行実行できることを保証する (切り詰め操作を採用する場合でも、コミット指標の保護領域では並行実行可能であることを示す)。
5. **決定論的な木構造の構築アルゴリズム**: 葉ノード数 n の 2 進数展開 $n = \sum_{h \geq 0} b_h 2^h$ に基づく決定論的な構築手法を確立し、同一データに対して常に同一の木構造が生成されることを保証した。
6. **効率的な直列化形式**: 追記操作が単一のファイル追記で完結する直列化形式を設計し、二次

記憶装置への I/O を最小化した。空間複雑性を $\Theta(n \log n)$ におさえながら、実用的な性能を達成した。

これらの貢献により、本研究は従来の Merkle Tree および History Tree を拡張し、理論的な優位性と実装の実用性を両立する新しいデータ構造を確立した。これは分散システムにおける効率的なログ同期と検証に適用可能な新しい基盤技術を提供する。

2 Preliminaries

2.1 Merkle Tree

典型的な Merkle Tree [2] は完全二分木であり、すべてのノードはそのノードを根とする部分木のダイジェストを表すハッシュ値を持つ。各葉ノードは葉に含まれるデータ要素 d_i のハッシュ値 $H(d_i)$ をハッシュ値として持つ。各中間ノード $b_{i,j}$ は 2 つの子ノードのハッシュ値の連結から計算されたハッシュ値 $H(l.hash \,||\, r.hash)$ を持つ。各要素に対する包含証明 (inclusion proof) を生成するための時間計算量および空間計算量は共に $O(\log n)$ である。

2.2 History Tree

History Tree [3] は要素の動的な追加操作をサポートするように拡張した Merkle Tree の亜種である。この木構造は 1 つ以上の完全二分部分木と、それらを連結するノードで構成されている。ここで最も高い完全二分部分木を最左に、高さの降順で左から右に並んでいるものとする。History Tree に追加される新しいデータは、まず木構造の最も右の葉ノードとして配置され、これを高さ $h = 0$ の完全二分部分木とみなす。次に、左の 2 つの木構造を接続する中間ノードを再帰的に構築する。この過程で、複数の完全二分部分木が統合され 1 つの完全二分部分木となることがある。この追加操作は $O(\log n)$ で完了する。

2.3 Hash Function

Slate では Merkle Tree における第二原像攻撃を防止する目的でノードの高さ j ごとにドメイン分離したハッシュ関数を使用する。本論文では原像耐性、第二原像耐性、衝突耐性を持つ一般的な暗号論的ハッシュ関数 $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\lambda$ を仮定する。ここで λ はハッシュ出力のビット長である。Slate では $\mathcal{H}$ からドメイン j で分離されたハッシュ関数 h_j を使用する。これは、具体

的には、高さ j を入力の接頭辞として用いたハッシュ関数 $h_j(m) = \mathcal{H}(j \,||\, m)$ を使用できる。ここで $||$ はビット列の連結演算子を表している。

2.4 Log Structure

本論文では、すべてのデータが時系列順に末尾へのみ追記され、一度記録されたデータは不変となる追記専用 (append-only) のデータ構造を**ログ構造** (log structure) とする。$\mathbb{N}$ を自然数全体の集合 $\{1, 2, \ldots\}$、$\mathbb{N}_0$ を非負値 $\mathbb{N} \cup \{0\}$ としたとき、離散インデックス $i \in \mathbb{N}$ と値集合 V に対し、ログ (log) L は長さ $n \in \mathbb{N}_0$ を持つ有限列 (1) として定義する。

$$L = \langle x_1, x_2, \ldots, x_n \rangle, \quad x_i \in V, i \in \mathbb{N} \tag{1}$$

単なる順序付きデータ列とは異なり、ログ構造では既存のデータの変更や任意の位置への挿入・削除を許さないことを示唆しており、履歴の完全性と監査可能性が構造的に保証される。

2.5 Temporal Locality

時間的局所性 (temporal locality) とは、時刻 t にアクセスされたデータ要素が時刻 $t + \Delta t$ においても高い確率でアクセスされる性質を指す。例えば Web リクエストにおいては時間経過に対し重尾 (tail-heavy) な対数正規分布なることが知られており [6, 7]、ブロックチェーンのようなappend-only のデータ構造においても最近追加されたデータが次にアクセスされる要素である確率が重尾 (heavy-tailed) な分布、特にパレート分布やべき乗則に従うことが知られている [8]。これは、最近追加されたデータを優先的にキャッシュするデータ構造を用いることでキャッシュ効率や I/O 性能を改善できることを示唆している。

2.6 Hamming distance

2 つの n ビット列 $x, y \in \{0, 1\}^n$ のハミング距離 (Hamming distance) $d_H(x, y)$ は異なるビットの数として定義される。これは式 (2) に示すように XOR 演算と popcount CPU 命令により効率的に計算できる。

$$d_H(x, y) = |\{i : x_i \neq y_i\}| = \mathrm{popcount}(x \oplus y) \tag{2}$$

完全二分木に含まれる 2 つの葉のインデックス $i, j \in \{0, 1, \ldots\}$ のハミング距離 d_H は、それ

らの葉ノードの最小共通祖先 (lowest common ancestor) の高さを決定する。具体的には、最小共通祖先はルートから深さ $\lfloor \log_2(i \oplus j) \rfloor$ に位置する。例えば、インデックス $i = 5 = 101_2$ と $j = 7 = 111_2$ の場合、$i \oplus j = 010_2$ より、最小共通祖先は深さ 1 にあり、両方の葉ノードは高さ 2 の部分木内に存在する。

2.7 Terminology

Stratified Structure (persistent structure, eigentree), Leveled cache

3 Slate: Structure and Algorithm

本セクションでは Slate の構造的特徴と基本操作を形式的に定義する。Slate は History Tree に基づいており、最近追加されたデータとそれに関連するノードが局所化され末尾に追記される直列化形式に特徴を持つ。木構造はその多くの領域が完全二分部分木 (PSBT) で構成されている。最近追加された新しいデータは上層の小さな PSBT に配置され、古いデータは徐々に下層の PSBT にマージされる。この構造により、append 操作に $O(1)$ の I/O、get 操作にはハミング距離に応じた幾何分布的な I/O コストを持つようになる。

以下、データ構造の定義（3.1 節）、基本操作（3.2 節）、計算複雑度の解析（3.3 節）、正しさの証明（3.4 節）の順に説明する。

3.1 Data Structure

Slate は History Tree [3] に基づいた追記操作が可能な Merkle Tree [2] であり、その葉ノードをそれぞれ 1 つのデータに対応させることで順序付きのデータ列を表現する。本論文では n 個の葉ノードを持つ木構造を T_n と表記し、特に T_n の成長過程に注目するとき n-世代の木と表現する。ここで $T_0 = \emptyset$ である。便宜上、あるノード b が木構造 T に含まれていることを $b \in T$ と表記し、T_1 が T_2 の部分木であることを $T_1 \subseteq T_2$ と表記する。

添字: $i \in \mathbb{N}$ を 1 から始まる自然数とし、$i-1$ 世代の木構造 T_{i-1} に i 番目のデータ d_i を追加 (統合) するために発生するノードを $b_{i,j}$ とする。j は $b_{i,j}$ をルートとする部分木の中で最も遠い葉ノードまでのホップ数、つまり $b_{i,j}$ の高さを表している。i と j はそれぞれ式 (3) のように表すことができる。

$$i \in \{1, 2, \ldots\} : 0 \leq j \leq \lceil \log_2 i \rceil \tag{3}$$

$j = 0$ は葉ノードのケースであり、$b_{i,0} = b_i$ と表す。また、T_n に含まれる任意のノード $b_{i,j}$ をルートとしてすべての子孫ノードを含む部分木を $T_{i,j} \subseteq T_n$ と表す。

完全二分木: 本論文は木構造が完全二分木 (perfect binary tree) であることを強調するために $'$ 記号を用いる。T'_n は n 世代目の木構造が完全二分木であることを意味する。また $T'_{i,j}$ は部分木 $T_{i,j}$ が完全二分木であることを意味し、このとき $T'_{i,j}$ のルートを $b'_{i,j}$ と表す。

3.1.1 部分的永続データ構造

Slate の木は関数型プログラミングの文脈で部分的永続データ構造 [5] (partially persisten data structure) である。任意の完全二分部分木のルートノード $b'_{i,j} \in T_n$ は**永続性** (persistent) を持つ。つまり n 以降のすべての世代の木 T_m においても同じノード $b'_{i,j}$ が存在し続ける。ノード $b_{i,j}$ をルートとする部分木が完全二分木ではない場合、$b_{i,j}$ はその世代に限り存在する**一過性** (ephemeral) であり、それ以降の世代では木構造に現われることはない。

T_n に含まれる任意のノード $b_{i,j}$ が永続性ノードかどうか、言い換えると、$b_{i,j}$ をルートとする部分木が完全二分木かどうかは式 (4) で評価することができる。

$$\begin{cases} i \bmod 2^j = 0 & \text{if } b_{i,j} \text{ is persistent} \\ i \bmod 2^j \neq 0 & \text{otherwise, it is ephemeral} \end{cases} \tag{4}$$

すべての葉ノード $b_{i,0}$ は $i \bmod 2^0 = 0$ であることから永続性ノードであることは明らかである。

木構造 T_n は、互いに独立した (部分構造の関係にない) 部分木の中で最大と見なされる完全二分木の集合で構成されている。本論文では、そのような完全二分部分木を**層別木** (stratum tree) と呼ぶ。

3.1.2 エントリ

木構造への追記操作 $T_{i-1} \xrightarrow{d_i} T_i$ の過程で新規に計算されるノードの集合を**エントリ** (entry) と定義し e_i で表す。e_i に含まれるすべてのノードの添字 i (世代番号) はすべて同じであることに注意。ここで e_i を構成するノードの添字 j を求める。

i の 2 進数表現に対し、最右桁から連続する 0 の個数を得る関数を $\mathrm{ctz}(i)$ とする。言い換えると $\mathrm{ctz}(i)$ は i の最も右に位置する 1 の位置を意味している。i の 2 進数表現において、0 から $\mathrm{ctz}(i)$ 桁は永続性ノードを表しており、$\mathrm{ctz}(i) + 1$ 桁以降に現われる 1 は一過性ノードを表して

いる。それぞれ j の集合に変換すると式 (5) と (6) のようになる。

$$\mathcal{J}'(i) = \{\mathrm{ctz}(i), \mathrm{ctz}(i) - 1, \ldots, 0\} \tag{5}$$

$$\mathcal{J}(i) = \left\{ p + 1 \;\middle|\; p \in \mathbb{N}_0, \left\lfloor \frac{i}{2^p} \right\rfloor \bmod 2 = 1, p > \mathrm{ctz}(i) \right\} \tag{6}$$

$\mathcal{J}'$ と $\mathcal{J}$ はそれぞれ永続性ノードと一過性ノードを区別している。これより、エントリ e_i は式 (7) のように表すことができる。

$$e_i = \{b'_{i,j} \mid j \in \mathcal{J}'(i)\} \cup \{b_{i,j} \mid j \in \mathcal{J}(i)\} \tag{7}$$

e_n の空間計算量および構築にかかる時間計算量は共に $O(\log_2 n)$ である。後述するように、エントリは直列化形式においての保存単位であり、時間的局所性を空間的局所性に変換する、本論文において重要なパーツである。

3.1.3 木構造の構築

Slate は、それぞれの層別木を一世代限りの一過性ノードで接続して決定論的な木を構築する。図 1 は $n = 22$ の木構造 T_{22} を表しており、3 つの層別木 $T'_{16,4}, T'_{20,2}, T'_{22,1}$ を 2 つの一過性ノード $b_{22,5}, b_{22,3}$ で接続して木を成している様子を表している。

T_n に含まれる層別木の数は n の 2 進数表現での 1 の個数 $\mathrm{popcount}(n)$ に等しく、ぞれぞれの層別木の高さ j は、n の 2 進数表現において対応する 1 が現われる位置と等しい (最右桁の位置を 0 とする)。高さ j の完全二分木は 2^j 個の葉ノードを含んでいることから、n は式 (8) のようにそれぞれの層別木の葉ノード数の和で表すこともできる。

$$n = \sum_{j \geq 0} \alpha_j \, 2^j \quad (\alpha_j \in \{0, 1\}) \tag{8}$$

ここで α_j は n の 2 進数表現の j 桁目の値を意味する。これより T_n に含まれる各層別木のルートノードの集合 $\mathcal{B}'_n$ は式 (9) のように表すことができる。

$$\mathcal{B}'_n = \left\{ b'_{i,j} \;\middle|\; j \in \mathbb{N}_0, \left\lfloor \frac{n}{2^j} \right\rfloor \bmod 2 = 1, i = \left\lfloor \frac{n}{2^j} \right\rfloor \cdot 2^j \right\} \tag{9}$$

$\mathcal{B}'_n$ が 1 つのノードのみを含む場合、そのノードは $b_{n,\log_2 n}$ でなければならず、木構造全体が完全二分木 T'_n であることを示している。そうでない場合、複数の層別木を結合する一過性ノードを生成する必要がある。二項演算子 $\otimes$ を左右のオペランドを子に持つ新しいノードを生成する演算としたとき、降順に並べた $\mathcal{B}'_n$ に対する右畳み込み操作 $\mathrm{reduce\_right}(\otimes, (\mathcal{B}'_n)_\downarrow)$ は一過性ノード

を生成しながら最終的に T_n のルートノードを生成する。

命題 3.1. 木構造 T_n に含まれる層別木は、n の *2* 進数表現における数値 *1* に対応しており、その *1* が出現する位置を高さとする。

命題 3.2. エントリ e_n は、n の *2* 進数表現において最も右に位置する *1* に対応する層別木のルートノードを含んでおり、それより左に位置するすべての層別木のルートノードを参照している。

ノードの接続: ここで、T_n に含まれる任意の非葉ノード $b_{i,j}, j > 0$ に対して、left(i, j) を左の子ノードの添字を得る関数、right(i, j) を右の子ノードの添字を得る関数とする。まず、T_n に含まれる任意の非葉ノード $b_{i,j}$ に対して左の子ノードの添字は式 (10) で求めることができる。

$$\mathrm{left}(i,j) = \left(\left\lfloor \frac{i-1}{2^{j-1}} \right\rfloor \cdot 2^{j-1}, j-1 \right) = (i - 2^{j-1}, j-1) \tag{10}$$

一過性ノードの場合、接続の変則性から j にギャップが発生するため、右ノードの求め方が異なる。

$$\mathrm{right}(i,j) = \begin{cases} (i, j-1) & \text{if } b_{i,j} \text{ is persistent} \\ (i, \mathrm{ctz}(i)) & \text{else if } j = \min \mathcal{J}(i) \\ (i, \max\{k \in \mathcal{J}(i) \mid k < j\}) & \text{otherwise} \end{cases} \tag{11}$$

任意の非葉ノード $b_{i,j}$ とその右の子ノードは同じエントリ e_i に属することから、任意のノードの右の子ノードの i 成分は常に $b_{i,j}$ と同じ値である。

直列化形式: エントリの時系列順の列 $S_n = (e_1, e_2, \ldots, e_n)$ は木構造 T_n を直列化したログ構造である。図 2 は T_6 から T_7 への木構造の遷移と、直列化形式の変化を示している。

データ列と直列化された木構造の空間計算量は $O(n \log n)$ である。

例 2: T_n に含まれる完全二分部分木の集合を $\mathcal{T}'_n = \{T'_{i,j}\}$

3.1.4 被覆範囲

任意の部分木 $T_{i,j} \subseteq T_n$ に含まれる最右葉ノードは b_i である。最左葉ノード $b_{\mathrm{i\_min}(i,j)}$ のインデックスは式 (12) で求めることができる。

$$\mathrm{i\_min}(i,j) = \left\lfloor \frac{i-1}{2^j} \right\rfloor \times 2^j + 1 = 1 - ((i-1) \bmod 2^j) \tag{12}$$

したがって部分木 $T_{i,j}$ の被覆範囲は $[\mathrm{i\_min}(i,j), i]$ である。

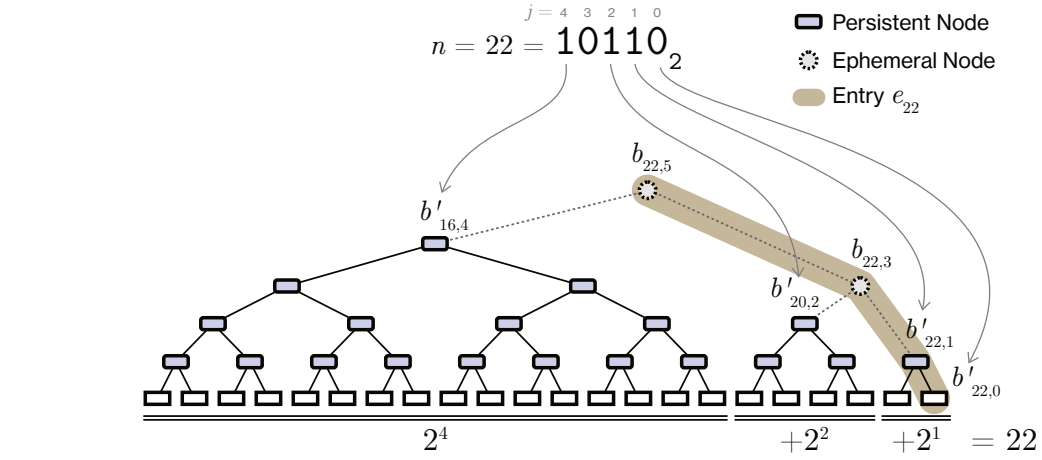

図1: T_{22} の木構造: 2 進数表現と層別木 (独立した完全二分部分木) の対応を視覚的に表している。22 の 2 進数表現 10110_2 から、T_{22} には $\mathcal{B}'_{22} = \{b'_{16,4}, b'_{20,2}, b'_{22,1}\}$ をルートとする 3 つの独立した層別木が含まれており、それらを連結する一過性ノード $\{b_{22,5}, b_{22,3}\}$ によって単一の木構造を成している。T_{22} のルートから b_{22} に至る経路上のノードはエントリ e_{22} を構成する。

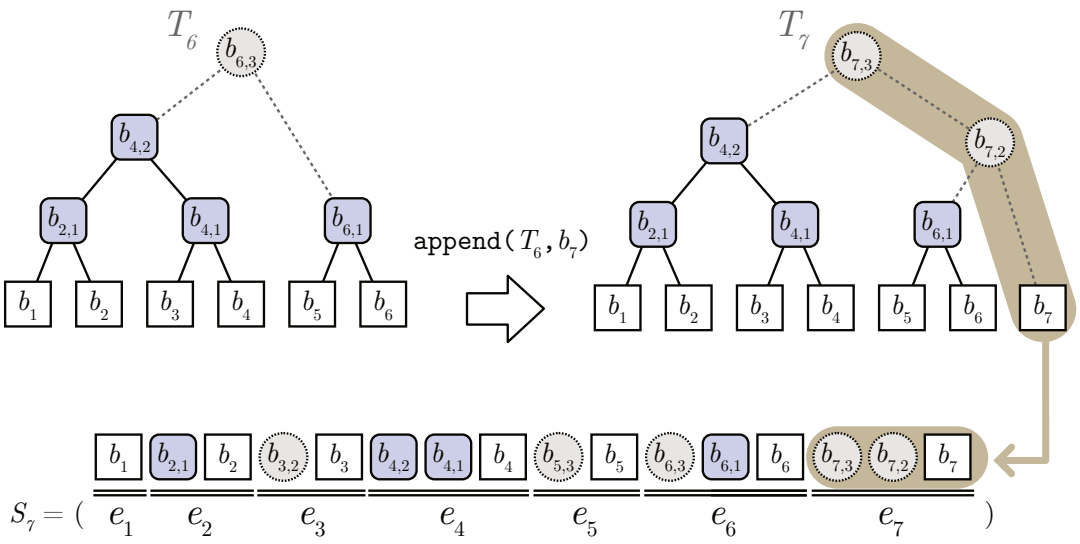

図2: T_6 から T_7 への木構造の遷移と直列化形式。

3.2 Append Operation

追記 (append) は、既存の木構造 T_{n-1} にデータ d_n を追加して T_n を生成する操作である。これは新しいエントリ e_n を生成し、既存の直列化形式に連結する操作と等価である。このセクションでは、追記のための操作手順を示し、e_n を構築するためには、直前のエントリ e_{n-1} に含まれるノード、またはそのノードが参照しているノードで十分であることを示す。

データ追加に伴って発生するノードは次のように構築する。

1. d_n に対応する葉ノード $b_n = \{h_0(d_n), d_n\}$ を生成し、T_{n-1} に含まれる層別木の並びの右に配置する。例えば $n = 7$ の場合、T_6 に含まれる層別木のルートノードの並びに b_7 を連結した $\Sigma_7 = (b_{4,2}, b_{6,1}, b_7)$ を作成する。
2. 木の並び Σ_n を右から 2 個のペアごとに中間ノードで接続するように畳み込む。左右のノードを連結する新しいノードを生成する演算子を $\otimes$ としたとき、$\mathrm{reduce\_right}(\otimes, \Sigma_n)$ を実行する。$b_{i_l,j_l} \oplus b_{i_r,j_r} = b_{i_r,j_l+1} = \{h_{j_l+1}(b_{i_l,j_l}.hash \,||\, b_{i_r,j_r}.hash)\}$

前述の通り、n の 2 進数符号は T_n に含まれる層別木を表していることから、追記による木構造の成長は $n-1$ から n への符号の変化で考えることができる。特に、木構造の成長により複数の層別木がマージされより大きな層別木となることを二進累積すると言う。

命題 3.3. 任意の $n \geq 1$ について、エントリ e_n は直前のエントリ e_{n-1} によって保持される状態と新規データ d_n のみから完全に計算可能である。

Proof. $n = 1$ の基底ケースでは $e_1 = \{b_1\}$ となることから d_1 のみから e_1 を計算可能であることは明らかである。

$n-1 \to n$ の帰納ステップについて考える。$n-1$ の 2 進数表現において、最下位ビットから連続する 1 の個数 (つまり最初に 0 が現われる位置) を $\mathrm{cto}(n-1)$ とする。追記操作では、最下位ビットから連続する各 1 に対応する層別木が新しい葉 b_n とマージされ、高さ $\mathrm{cto}(n-1)$ の新しい層別木が形成される (二進累積)。命題 3.2 より、マージされる各層別木のルートノードは e_{n-1} に含まれるノードのいずれかと直接接続していることから、この新しい層別木を構築するために必要なノードは e_{n-1} から知ることができる。

次に、$\mathrm{cto}(n-1)$ より大きい桁ではインクリメントの影響を受けず、$n-1$ と n とで同一の値であることから、e_n は e_{n-1} が参照している層別木のルートノードをそのまま引き継ぐことができる。したがって、e_n を構築するためには e_{n-1} と d_n があれば十分である。 □

命題 3.3 は、Slate で効率的な追記を行うためには、高速なメモリ上に現在のエントリ e_n のみを保持していれば十分であることを示している。以降、本論文では、特に指定しない限り、木構造 T_n において最新のエントリ e_n が高速なメモリ上に保持されていると想定する。

追記操作の複雑度: 木構造 T_{n-1} に新しいデータ d_n を追記する操作では、エントリ e_n を構築するのに要する時間複雑性は $O(\log n)$、エントリ e_n 自身の空間複雑性は $O(\log n)$、エントリ e_n を直列化形式に永続化する I/O 命令複雑性は $O(1)$ である。

3.3 Get Operation

本セクションでは Slate における参照操作を形式的に定義する。get 操作は、木構造 T_n 中の任意のインデックス $i \in [1, n]$ に対応する葉 b_i およびデータ要素 d_i を参照し、また直列化形式の文脈ではその記録位置を特定する。

$$\mathrm{get}(T_n, i) \to d_i \tag{13}$$

3.3.1 木構造における論理的経路

補題 3.1 (層別木の被覆範囲)**.** 木構造 T_n に含まれる任意の層別木 $T'_{i,j} \subseteq T_n$ の被覆範囲は式 *(14)* で表される。

$$L(T'_{i,j}) = [i - 2^j + 1, i] = [\left\lfloor \frac{i-1}{2^j} \right\rfloor \times 2^j + 1, i] \tag{14}$$

Proof. 式 (9) より、T_n に高さ j の層別木が存在するための必要条件は $\lfloor n/2^j \rfloor \bmod 2 = 1$ である。このとき、層別木のルートノード $b'_{i,j}$ は $i = \lfloor n/2^j \rfloor \times 2^j$ である。高さ j の完全二分木は 2^j の葉ノードを含み、最右葉は b_i であるため、最左葉は b_{i-2^j+1} である。したがって被覆範囲は式 (14) となる。 □

補題 3.2 (XOR による層別木の特定)**.** 木構造 T_n において、任意の葉ノード b_i $(1 \le i \le n)$ を含む層別木の高さ j は式 *(15)* で表すことができる。

$$j = \lfloor \log_2(n \oplus (i-1)) \rfloor \tag{15}$$

ここで $\oplus$ はビット単位の排他的論理和 *(XOR)* を表す。

Proof. 式 (8) の通り、n と $i-1$ の 2 進数表現をそれぞれ $n = \sum_{k\ge 0} \alpha_k 2^k$、$i-1 = \sum_{k\ge 0} \beta_k 2^k$

、$\alpha_k, \beta_k \in \{0,1\}$ とする。$0 \leq i-1 < n$ より、n と $i-1$ の 2 進数表現を上位桁から比較すると、ある位置 $k=p$ で初めて $\alpha_p = 1$ かつ $\beta_p = 0$ となる。この位置 p より上位のすべての桁 kp では $\alpha_k = \beta_k$ が成り立つ。

XOR 演算の定義より、$n \oplus (i-1)$ の桁 k は $\alpha_k \neq \beta_k$ のとき 1、そうでないとき 0 となる。位置 p は $\alpha_k \neq \beta_k$ を満たす最大の桁 k であるため、$n \oplus (i-1)$ で最も上位の 1 は p に位置する。したがって $p = \lfloor \log_2(n \oplus (i-1)) \rfloor$ である。

位置 p において $\alpha_p = 1$ であるため、命題 3.1 より、T_n には高さ p の層別木 $T'_{i^\dagger,p}$ が存在する。補題 3.1 より、この層別木の被覆範囲は $[i^\dagger - 2^p + 1, i^\dagger]$ である。kp となる桁では $\alpha_k = \beta_k$ であるため

$$i - 1 = \sum_{k \geq 0} \beta_k 2^k < \sum_{kp} \alpha_k 2^k + 2^p = i^\dagger$$

また、$\beta_p = 0$ かつ $\beta_k \in \{0,1\}$ より

$$i - 1 = \sum_{k<p} \beta_k 2^k + \sum_{kp} \beta_k 2^k \geq \sum_{kp} \alpha_k 2^k = i^\dagger - 2^p$$

したがって $i^\dagger - 2^p \leq i - 1 < i^\dagger$ であり、$i^\dagger - 2^p + 1 \leq i < i^\dagger$ となる。これは葉 b_i が高さ p の層別木の被覆範囲に含まれていることを示す。 □

補題 3.3 (層別木内のローカルインデックス)**.** 木構造 T_n において、葉 b_i を含む層別木 $T'_{i^\dagger,j}$ とする。層別木 $T'_{i^\dagger,j}$ 内での b_i の *0-indexed* なローカルインデックス k は、式 *(16)* に示すように $i-1$ の下位 j 桁で与えられる。

$$k = (i-1) \wedge (2^j - 1) \tag{16}$$

ここで $\wedge$ はビット単位の論理積を表す。

Proof. 層別木 $T'_{i^\dagger,j}$ の被覆範囲は $[i^\dagger - 2^j + 1, i^\dagger]$ であるため、葉 b_i のローカルインデックスは:

$$k = i - (i^\dagger - 2^j + 1) = i - 1 - (i^\dagger - 2^j)$$

補題 3.2 の証明より、n と $i-1$ は、2 進数表現において桁 j より上位のすべての桁が一致する。ここで $T'_{i^\dagger,j}$ の左隣の層別木の最右葉 $b_{i^\dagger - 2^j}$ の位置は:

$$i^\dagger - 2^j = \sum_{mj} \alpha_m 2^m = \sum_{mj} \beta_m 2^m = (i-1) \wedge \neg(2^{j+1} - 1)$$

で表される。これは $i-1$ の 2 進数表現の下位 j 桁を 0 に設定した値と等しい。これより:

$$k = (i-1) - ((i-1) \wedge \neg(2^{j+1}-1)) = (i-1) \wedge (2^j - 1)$$

したがって k は $i-1$ の下位 j 桁と等しい。 □

命題 3.4 (完全二分木における経路決定)**.** 高さ h の一般的な完全二分木 T' において、葉ノードを左から順に $0, 1, 2, \ldots, 2^h - 1$ とインデックス付けする。ルートノードから葉 k $(0 \leq k < 2^h)$ への経路は、k の h 桁 *2* 進数表現*:*

$$k = \sum_{m=0}^{h-1} k_m 2^m, \quad k_m \in \{0, 1\}$$

において、上位ビットから順に、ビット $k_m = 0$ なら左枝子ノードへ、$k_m = 1$ なら右枝子ノードへ移動することで到達できる。形式的には、高さ h' $(0 < h' \leq h)$ のノードから $h'-1$ のノードの遷移方向は*:*

$$\text{direction}(k, h') = \begin{cases} \textit{left} & \textit{if } k_{h'-1} = 0 \\ \textit{right} & \textit{if } k_{h'-1} = 1 \end{cases}$$

で決定する。

Proof. 高さ h に関する帰納法で証明する。

基底ケース: $h = 1$ の完全二分木は 2 つの葉を持ち、左の葉がインデックス 0、右の葉がインデックス 1 である。従って $k = 0$ のとき左枝子ノード、$k = 1$ のとき右枝子ノードを正しく選択する。

帰納ステップ: 高さ $h-1$ 以下の完全二分木で命題が成立すると仮定し、高さ h の場合を示す。高さ h の完全二分木は、ルートの左枝と右枝にそれぞれ高さ $h-1$ の完全二分部分木を持つ。葉のインデックス付けに対し、左枝は $[0, 2^{h-1} - 1]$ を被覆し、右枝は $[2^{h-1}, 2^h - 1]$ を被覆する。

葉 k について、$k < 2^{h-1}$ のとき、k の $h-1$ 桁目は 0 であり、葉 k は左枝の部分木に含まれているため左枝に移動する。$k \geq 2^{h-1}$ のとき k の $h-1$ 桁目は 1 であり、葉 k は右枝の部分木に含まれるため右枝に移動する。補題 3.3 に従い、どちらの部分木へ移動したとしても、k の下位 $h-1$ 桁はその部分木のローカルインデックスを表しており、帰納法により葉 k に到達する。したがって、k の最上位の $h-1$ 桁目が最初の遷移方向を決定し、残りの桁が部分木内での経路を決定する。 □

定理 3.1 (2 進数表現による統一的経路決定)**.** 木構造 T_n において、任意の葉 b_i $(1 \le i \le n)$ への経路は n と i の *2* 進数表現のみから以下の手順で決定できる。

1. $j = \lfloor \log_2(n \oplus (i-1)) \rfloor$ とする。
2. T_n のルートから $\mathrm{popcount}(\lfloor n/2^j \rfloor)$ 回、右枝子ノードに移動する。
3. $j \neq \mathrm{ctz}(n)$ であれば *1* 回左枝子ノードに移動する。
4. $i-1$ の *2* 進数表現の下位 j 桁を上位から順に読み、0 なら左、1 なら右に移動する。

Proof. 補題 3.1 と 3.2 より、手順 1, 2, 3 は n と $i-1$ の 2 進数表現から葉 b_i を含む層別木を特定し、そのルートノードまで移動する。命題 3.4 より、手順 4 はその層別木のルートから葉 b_i まで移動する。 □

定理 1 より、T_n のルートから葉 b_i までの経路長 (ホップ数) は式 (17) で表される。

$$\mathrm{hops}(n,i) = \mathrm{popcount}(\lfloor n/2^{j+1} \rfloor) + 1[j \neq \mathrm{ctz}(n)] + j \tag{17}$$

ここで $j = \lfloor \log_2(n \oplus (i-1)) \rfloor$ であり、$1[\cdot]$ は指示関数とする。popcount と j はそれぞれ $O(\log n)$ であることから、T_n のルートから葉 b_i まで到達する時間複雑性は $O(\log n)$ である。

例: T_{22} から葉 b_{18} への経路について考える。

$$\begin{aligned}
n &= 22 = 10110_2 \\
i &= 18 = 10010_2 \\
n \oplus (i-1) &= 00111_2 \\
j &= \lfloor \log_2(n \oplus (i-1)) \rfloor = 2 \\
\lfloor n/2^{j+1} \rfloor &= 22 >> 3 = 00010_2 = 2 \\
\mathrm{popcount}(\lfloor n/2^{j+1} \rfloor) &= \mathrm{popcount}(2) = 1 \quad \text{(move right branch once)} \\
\mathrm{ctz}(n) &= \mathrm{ctz}(22) = 1 \neq j \quad \text{(move left branch)} \\
k &= (i-1) \wedge (2^j - 1) = 10001_2 \wedge 00011_2 = 01_2
\end{aligned}$$

したがって経路は:

$$b_{22,5} \xrightarrow{\text{right}} b_{22,3} \xrightarrow{\text{left}} b'_{20,2} \xrightarrow{\text{0:left}} b'_{18,1} \xrightarrow{\text{1:right}} b_{18}$$

であり、ホップ数は $\mathrm{hops}(22, 18) = 4$ となる。

3.3.2 直列化形式における物理的 I/O コスト

Slate の直列化形式 $S_n = (e_1, e_2, \ldots, e_n)$ において、各エントリ e_i を I/O (seek+read) の単位とする。ここで最新のエントリ e_n はメモリ上に保持されていると仮定する。式 (7) より、エントリ e_i に含まれるノードは、木 T_i から葉 b_i への経路上のすべてのノードと一致する。この経路は、一過性ノードの連鎖と最右の層別木の右辺を経由し、常に右枝方向へ降下する。つまり右枝方向の移動は単一のエントリ内での移動に閉じており、I/O が発生することはない。

木構造 T_n のルートから葉 b_i への経路において、エントリ e_n がメモリ上に保持されているとき、I/O が発生するのは左枝子ノードへの移動時のみである。したがって I/O 回数は経路上の左枝移動回数に等しい。

定理 3.2 (I/O 回数)**.** 木構造 T_n において、エントリ e_n から葉 b_i $(1 \le i \le n)$ を参照するための *I/O* 回数 $\mathrm{IO}(n, i)$ は式 *()* で与えられる。

$$\mathrm{IO}(n, i) = 1[j \neq \mathrm{ctz}(n)] + j - \mathrm{popcount}((i-1) \wedge (2^j - 1)) \tag{18}$$

ここで $j = \lfloor \log_2(n \oplus (i-1)) \rfloor$ である。

Proof. 定理 1 より、T_n のルートから葉 b_i への経路は、1) ルートから $\mathrm{popcount}(\lfloor n/2^{j+1} \rfloor)$ 回の右枝移動、2) $j \neq \mathrm{ctz}(n)$ であれば 1 回の左枝移動、3) 層別木内で $k = (i-1) \wedge (2^j - 1)$ の 2 進数表現の各桁ごとの左右枝への移動、の 3 フェーズで構成される。フェーズ 1 はすべて右枝移動であるため I/O は発生しない。フェーズ 2 は 0 回か 1 回の左枝移動である。フェーズ 3 では、左枝移動の回数は k の j 桁 2 進数表現における 0 (左枝移動) の個数と一致する。したがって $j - \mathrm{popcount}(k) = \mathrm{popcount}(\neg k)$ 回の I/O が発生する。ここで $\neg k$ は k のビット単位の反転である。 □

定義 3.1 (層別木内ハミング距離)**.** 高さ j の層別木 $T'_{i^\dagger, j}$ において、葉 b_i のローカルインデックス $k = (i-1) \wedge (2^j - 1)$ に対し、層別木内ハミング距離を式 *(19)* のように定義する。

$$d_H(j, k) = \mathrm{popcount}((2^j - 1) \oplus k) \tag{19}$$

ここで $2^j - 1$ は層別木 $T'_{i^\dagger, j}$ の最右葉のローカルインデックスである。

補題 3.4 (層別木内ハミング距離と左枝移動回数の等価性)**.** 高さ j の完全二分において、ルート

から葉 k $(0 \leq k < 2^j)$ への経路上の左枝移動回数は $d_H(j,k)$ に等しい。

$$j - \mathrm{popcount}(k) = d_H(j,k) \tag{20}$$

Proof. $2^j - 1$ の 2 進数表現は j 桁の 1 の並びである。k は $i-1$ の下位 j 桁である。したがって $(2^j-1) \oplus k$ は k を j 桁 2 進数表現でビット反転させた値になる。したがって:

$$d_H(j,k) = \mathrm{popcount}((2^j-1) \oplus k) = \mathrm{popcount}(\neg k) = j - \mathrm{popcount}(k)$$

命題 3.4 より、完全二分木のルートから葉 k への経路において、k の各ビットが 0 なら左枝移動、1 なら右枝移動である。したがって、左枝移動回数は k の j 桁 2 進数表現における 0 の個数、すなわち $j - \mathrm{popcount}(k)$ である。 □

定理 3.3 (I/O 回数のハミング距離表現)**.** 木構造 T_n において、エントリ e_n から葉 b_i への *I/O* 回数は式 *(21)* で与えられる。

$$\mathrm{IO}(n,i) = 1[j \neq \mathrm{ctz}(n)] + d_H(j,k) \tag{21}$$

ここで*:*

- $j = \lfloor \log_2(n \oplus (i-1)) \rfloor$ *(*葉 b_i を含む層別木の高さ*)*
- $k = (i-1) \wedge (2^j - 1)$ *(*層別木内の *0-indexed* なローカルインデックス*)*

Proof. 定理 3.2 と補題 3.4 より明らか。 □

I/O 回数は以下の 2 つの成分の和として解釈できる。

1. 層別木への到達コスト $1[j \neq \mathrm{ctz}(n)]$: 葉 b を含む層別木が最右でなければ 1、最右なら 0。
2. 層別木内の距離コスト $d_H(j,k)$: 層別木の最右葉 $b_{i^\dagger}$ と目的の葉 b_i のローカルインデックス間のハミング距離。

層別木 $T'_{i^\dagger,j}$ の被覆範囲は $[i^\dagger - 2^j + 1, i^\dagger]$ である。葉 b_i の 0-indexed なローカルインデックス k は:

$$k = i - (i^\dagger - 2^j + 1) = i - i^\dagger + 2^j - 1$$

である。これを変形して:

$$2^j - 1 - k = 2^j - 1(i - i^\dagger + 2^j - 1) = i^\dagger - i$$

したがって、ハミング距離は層別木の最右葉との差 $i^{\dagger} - i$ に依存する。

命題 3.5 (I/O 回数とハミング距離の関係)**.** 木構造 T_n において、エントリ e_n から葉 b_i への *I/O* 回数 $\mathrm{IO}(n, i)$ と相対ハミング距離 $d_H(n, i-1)$ の間には式 *()* に示す関係が成り立つ。

$$\mathrm{IO}(n, i) = 1[j \neq \mathrm{ctz}(n)] + d_H(n, i-1) - 1 + \Delta(n, i) \tag{22}$$

ここで $j = \lfloor \log_2(n \oplus (i-1)) \rfloor$ であり、$Delta(n, i)$ は以下に定義する補正項である*:*

$$\Delta(n, i) = \mathrm{popcount}(n \wedge (2^j - 1)) - 2 \times \mathrm{popcount}(n \wedge (i-1) \wedge (2^j - 1)) \tag{23}$$

定理 3.4 (I/O 回数と相対ハミング距離の関係)**.** $j = \lfloor \log_2(n \oplus (i-1)) \rfloor$ としたとき、以下の性質を持つ*:*

1. 上限*:* $\mathrm{IO}(n, i) \leq j + 1 = \lfloor \log_2(n \oplus (i-1)) \rfloor + 1$
2. 下限*:* $\mathrm{IO}(n, i) \geq 1[j \neq \mathrm{ctz}(n)]$
3. 最近傍での効率*:* $\mathrm{IO}(n, n) = 0$

3.4 Truncate Operation

ある世代 n から、それより小さな世代 $m < n$ に戻すには、直列化形式 $S_n = (e_1, e_2, \ldots, e_m, \ldots, e_n)$ を切り詰めて $S_m = (e_1, e_2, \ldots, e_m)$ にすることができる。

3.5 Difference Detection Protocol

Merkle Tree における包含証明 (inclusion proof) [2] (または所属証明、認証パス) とは、特定のデータ d_i がツリーに含まれていることを、ツリー全体を開示することなく暗号論的に検証可能にする小さなデータ構造である。この包含証明はルートから葉までの経路上の兄弟ノード (sibling node) のハッシュ値の順序付き列である。Slate における差異検出でも包含証明と類似したハッシュ値の列を用いるが、その構造はやや異なっており、また目的も包含の証明ではない。本論文では、Slate の差異検出のために木を縦方向に割って断面の各ハッシュ値 (層) をサンプリングした列を**層サンプル** (stratum sample) を導入する。

3.5.1 Merkle Hash

一般的な Merkle Tree では、任意のノード $b_{i,j}$ が持つハッシュ値 $c_{i,j}$ は式 (24) を満たす。

$$c_{i,j} = \begin{cases} h_j(c_{i_l,j_l} \,||\, c_{i_r,j_r}) & \text{if } j > 0 \\ h_0(d_i) & \text{otherwise} \end{cases} \tag{24}$$

ここで c_{i_l,j_l} と c_{i_r,j_r} はノード $b_{i,j}$ の左右の子ノードのハッシュ値を表している。Merkle Tree の性質より、ハッシュ値 $c_{i,j}$ は対応するノード $b_{i,j}$ をルートとする部分木に含まれるすべてのデータの暗号論的ダイジェストを意味する。したがって、2 つの木構造 T_n^A と T_n^B のルートハッシュ c_n^A, c_n^B が異なる場合、暗号論的に T_n^A と T_n^B には異なるデータが少なくとも 1 つ含まれていることを意味している。

3.5.2 Stratum Sampling

Slate の木構造 T_n に含まれる任意の部分木 $T_{i,j} \subseteq T_n$ のルートノード $b_{i,j}$ から、部分木の最右葉ノード b_i への経路を式 (25) のように定義する。

$$\mathcal{P}_{i,j} = \{b_{i,k} \mid 0 \le k \le j\} \tag{25}$$

この経路は、高さ j のルートノードから 0 の葉まで、常に右枝に向かって降下する。式 (10) より、$\mathcal{P}_{i,j}$ 上のノード $b_{i,k}(k > 0)$ の左子ノードを式 (26) のように定義する。

$$\text{leftchild}(b_{i,k}) = \begin{cases} b_{i-2^{k-1},k-1} & \text{if } k > 0 \wedge i \ge 2^{k-1} \\ \perp & \text{otherwise} \end{cases} \tag{26}$$

ここで $\perp$ は左子ノードが存在しないことを示す。

定義 3.2 (層サンプル)**.** 木構造 T_n に含まれる任意の部分木 $T_{i,j} \subseteq T_n$ において、$T_{i,j}$ のルートノード $b_{i,j}$ から部分木の最右葉ノード b_i に至る経路 $\mathcal{P}_{i,j}$ 上の各ノードの左子ノードのハッシュ値と、最右葉 b_i のハッシュ値の集合を**層サンプル** *(stratum sample)* と定義する。

$$\mathcal{C}_{i,j} = \{c_{i',j'} \mid b_{i',j'} = \text{leftchild}(b_{i,k}), b_{i,k} \in \mathcal{P}_{i,j}, b_{i',j'} \neq \perp\} \cup \{c_{i,0}\} \tag{27}$$

層サンプルの各 $c_{i',j'}$ の添字は式 (10) と式 (11) で求めることができる。

補題 3.5 (層サンプルの高さ上限)**.** $j > 0$ である層サンプル $\mathcal{C}_{i,j}$ に含まれる任意のハッシュ値

$c_{i',j'}$ に対応するノード $b_{i',j'}$ の高さは、ルートノード $b_{i,j}$ より必ず小さい。

$$\forall c_{i',j'} \in \mathcal{C}_{i,j}, j > 0 : j' < j \tag{28}$$

Proof. 定義 3.2 より、層サンプル $\mathcal{C}_{i,j}$ のすべてのハッシュ値は以下のいずれかのノードに由来する:

1. ルートノード $b_{i,j}$ から最右葉 b_i への経路上の各ノードの左子ノード
2. 最右葉 b_i 自身 (高さ $j = 0$ のノード)

$j > 0$ より 2. のケースは除外できる。ルートノード $b_{i,j}$ は $\mathcal{P}_{i,j}$ に含まれるノードのなかでもっとも大きい高さ j を持つ。式 (10) より、$\mathcal{P}_{i,j}$ 上の各ノードの左子ノードの高さはその親ノードより厳密に 1 つ小さい。したがって、層サンプル $\mathcal{C}_{i,j}$ に含まれるすべてのハッシュ値 $c_{i',j'}$ に対して $j' < j$ が成立する。 □

補題 3.6 (層サンプルの完全被覆性)**.** 木構造 T_n に含まれる任意の部分木 $T_{i,j} \subseteq T_n$ において、その層サンプル $\mathcal{C}_{i,j}$ の各ハッシュ値 $c_{i',j'}$ に対応するノード $b_{i',j'}$ を頂点とする各部分木は、部分木 $T_{i,j}$ の葉ノード全体をカバーする。

$$L(T_{i,j}) = \bigcup_{c_{i',j'} \in \mathcal{C}_{i,j}} L(T_{i',j'}) = [i - ((i-1) \bmod 2^j), i] \tag{29}$$

ここで $L(T_{i,j})$ は部分木 $T_{i,j}$ の被覆範囲 *(式 (12) も参照)* を表す。

Proof. $T_{i,j}$ のルート $b_{i,j}$ から葉 b_i までの経路上のノード集合を $\mathcal{P}_{i,j} = (v_k, \ldots, v_0)$ とする。ここで v_0 は葉ノード、v_k はルートノードである。各 v_j に対してその兄弟ノードを s_j とする。二分木構造より、各段階で:

$$L(v_{j+1}) = L(v_j) \cup L(s_j), \quad L(v_j) \cap L(s_j) = \emptyset$$

が成り立つ。これを再帰的に適用すると:

$$L(v_k) = L(v_0) \cup \bigcup_{j=1}^{k} L(s_j)$$

を得る。$L(v_k)$ はルートノードの被覆範囲 $L(b_{i,j})$ である。また $L(v_0) = L(b_i)$ である。$\bigcup_{j=1}^{k} L(s_j)$

は $\mathcal{P}_{i,j}$ の各左子ノードの被覆範囲を表している。この構造は定義 3.2 の層サンプル $\mathcal{C}_{i,j}$ と同じであるため、層サンプル $\mathcal{C}_{i,j}$ は木 $T_{i,j}$ の葉を完全に被覆する。各 s_j は異なる高さで $L(v_j)$ の補集合として現われるため、互いに独立している。

部分木コミットメント列は包含証明に葉ノード b_i のコミットメント (ハッシュ値) を含めた集合であるため、任意の $1 \leq i \leq n$ に対し、部分木コミットメント列 $C_{n,i}$ は T_n に含まれるすべての葉ノードを被覆する。 □

補題 3.7 (層サンプルの暗号論的ダイジェスト性)**.** 層サンプル $\mathcal{C}_{i,j}$ は、木 $T_{i,j}$ のルートハッシュ $c_{i,j}$ と同様に、その木に含まれるすべての葉に含まれるデータの暗号論的ダイジェストを表す。

Proof. 補題 3.6 とハッシュ関数 h_j の衝突困難性より、層サンプルはその木のすべての葉ノードを被覆するため、各部分木のルートハッシュ値の集合は木全体のダイジェストと同等である。 □

補題 3.8 (層サンプルの等価性)**.** *2* つの n-世代の木構造 T_n^A と T_n^B の部分木 $T_{i,j}^A \subseteq T_n^A$ と $T_{i,j}^B \subseteq T_n^B$ において、双方の層サンプルが等しい場合、暗号論的に $T_{i,j}^A$ と $T_{i,j}^B$ の被覆範囲に含まれる葉のデータはすべて等しい。

$$\mathcal{C}_{i,j}^A = \mathcal{C}_{i,j}^B \quad \Longleftrightarrow \quad \forall k \in [\mathrm{i\_min}(i,j), i] : d_k^A = d_k^B$$

Proof. 補題 3.7 とハッシュ関数 h_j の衝突困難性より、葉のケースでは $\mathcal{C}_{i,0}^A = \mathcal{C}_{i,0}^B$ は $h_0(d_i^A) = h_0(d_i^B)$ を意味しているため $d_i^A = d_i^B$ は明らか。中間ノードのケースも同様に $c_{i,j}^A = c_{i,j}^B$ ならば双方の $T_{i,j}$ の被覆範囲の葉のデータは暗号論的に等しい。補題 3.6 より、層サンプル $\mathcal{C}_{i,j}$ は $T_{i,j}$ のすべての葉のデータを被覆するため、双方の $T_{i,j}$ に含まれるデータはすべて等しいことが証明される。 □

3.5.3 Difference Detection using the Stratum Sample

差異位置の検出は 2 者間での層サンプルを 1 回以上交換することによって達成される。A と B の 2 者がそれぞれツリー T_{n_a} と T_{n_b} を保持していると想定する。ここで A と B は $n = \min(n_a, n_b)$ とする n-世代のツリー T_n 上で動作することに合意しているものとする (世代が大きい方の木は n 世代のスナップショットを使用する)。

発起者である A は、B の持つデータ列との差異を検出するために、まず最初のラウンドで自身の木構造 T_n^A 全体を対象とする層サンプル $\mathcal{C}_{n,\lceil \log_2 n \rceil}^A$ を作成して B に送信する。B はその各要素を自身の同じ部分木の層サンプル $\mathcal{C}_{n,\lceil \log_2 n \rceil}^B$ と比較する。双方の層サンプルが同一であれば、

補題 3.8 より、A と B の n-世代までのデータ列に差異がないことが暗号論的に証明され、B は $\perp$ で応答する。

一方、層サンプルに含まれる 1 つ以上のハッシュ値が異なっていた場合、それらに対応する部分木の範囲に差異が存在することが明らかになる。ここで我々は、データ列の分岐を検出することを目的としているため、最初に出現する差異の位置を特定したいと考えている。したがって、差異のあるハッシュ値 $c^A_{i^*,j^*} \neq c^B_{i^*,j^*}$ の中で i^* が最も小さい部分木に目的の差異が含まれていることになる。

ここで $j^* = 0$ であれば葉ノードの差異を意味しており、位置 i^* のデータ $d^A_{i^*}$ と $d^B_{i^*}$ が異なることが明らかとなるため、B は $\mathrm{OK}\{i^*\}$ で応答する。そうでない場合、B は b_{i^*,j^*} をルートとする部分木の層サンプル $\mathcal{C}^B_{i^*,j^*}$ を作成して A に送信し、送信役/受信役を入れ替えて第二ラウンドを開始する。このインタラクションを繰り返すことにより、A と B のどちらかは最終的に目的の差異の位置 i^* を発見する。

差異検出を定式化した関数として表現すると式 (30) のようになる。

$$\mathrm{finddiff}(\mathcal{C}^{\mathrm{sender}}_{i,j}, \mathcal{C}^{\mathrm{receiver}}_{i,j}) = \begin{cases} \perp & \text{if } \mathcal{C}^{\mathrm{sender}}_{i,j} = \mathcal{C}^{\mathrm{receiver}}_{i,j} \\ \tilde{i} & \text{if } \delta_{i,j} = \{(\tilde{i}, 0)\} \\ \mathrm{finddiff}(\mathcal{C}^{\mathrm{receiver}}_{\tilde{i},\tilde{j}}, \mathcal{C}^{\mathrm{sender}}_{\tilde{i},\tilde{j}}) & \text{otherwise} \end{cases} \tag{30}$$

ここで $\delta_{i,j}$ は $\mathcal{C}^A_{i,j}$ と $\mathcal{C}^B_{i,j}$ において $c^A_{i^*,j^*} \neq c^B_{i^*,j^*}$ であるような添字 (i^*, j^*) の集合、$\tilde{i}, \tilde{j}$ は $\delta_{i,j}$ の中で最も小さい i^* を持つ添字ペアを表す。

$$\begin{aligned} \delta_{i,j} &= \{(i^*, j^*) \mid c^A_{i^*,j^*} \in \mathcal{C}^A_{i,j}, c^B_{i^*,j^*} \in \mathcal{C}^B_{i,j}, c^A_{i^*,j^*} \neq c^B_{i^*,j^*}\} \\ \tilde{i} &= \min\{i^* \mid (i^*, j^*) \in \delta_{i,j}\} \\ \tilde{j} &= j^* : (i^*, j^*) \in \delta_{i,j}, i^* = \tilde{i} \end{aligned}$$

初期実行は $\mathrm{finddiff}(\mathcal{C}^A_n, \mathcal{C}^B_n)$ で開始する。

定理 3.5 (差異検出の正当性)**.** T^A_n と T^B_n を世代 n のツリーとし、$D^A[1..n]$ と $D^B[1..n]$ をそれぞれのデータ列とする。ハッシュ関数 h が暗号論的に衝突困難であると仮定する。このとき、$\mathrm{finddiff}(\mathcal{C}^A_n, T^B_n)$ は以下を満たす*:*

1. $D^A[1..n] = D^B[1..n]$ ならば $\perp$ を返す。
2. そうでなければ $\tilde{i} = \min\{i \in [1, n] \mid D^A[i] \neq D^B[i]\}$ を返す。

Proof. 補題 3.7 より、$\mathcal{C}_n$ に含まれるすべてのハッシュ値が等しい場合、T_n^A と T_n^B は同じデータ列を持っており、差異は存在しない ($\perp$ となる)。

Δ を D_n^A と D_n^B の差異の位置の集合とする。つまり $\Delta = \{i \in [1,n] \mid D^A[i] \neq D^B[i]\}$ である。ある部分木 $T_{i,j} \subseteq T_n$ のカバーする範囲に差異が含まれている場合、$T_{i,j}$ のカバー範囲に Δ の少なくとも一つの要素が含まれている。

$$\Delta \cap L(T_{i,j}) \neq \emptyset$$

この場合、層サンプル $\mathcal{C}_{i,j}$ のうち、少なくとも 1 つのハッシュ値が異なっているはずである。補題 3.5 より、異なるハッシュ値 $c_{\tilde{i},\tilde{j}}$ に対応する部分木 $T_{\tilde{i},\tilde{j}}$ の高さは元の部分木 $T_{i,j}$ の高さより小さい。つまり、$\tilde{j} < j$ であり $T_{\tilde{i},\tilde{j}} \subset T_{i,j}$ である。

したがって、どのハッシュ値に差異があっても、元の $T_{i,j}$ より小さな範囲をカバーする部分木を選択することになる。したがって、finddiff のラウンドが進むにつれ被覆範囲は必ず小さくなり、最終的に高さゼロの部分木、つまり差異を含む葉ノード $b_{\tilde{i}}$ を特定する。 □

差異の特定は、データ列の中で最も古いもののみならず、すべての差異を特定できると予想される。ただし、そのような手法はデータ列の分岐点を検出することが目的の本論文の範囲外である。

定理 3.6 (差異検出の終了性)**.** finddiff($\mathcal{C}_n^A, \mathcal{C}_n^B$) は、$\tilde{i} = n$ のとき最良ケースで *1* 回、$\tilde{i} = 1$ のとき最悪ケースで $\lceil \log_2 n \rceil$ 回のラウンドで終了する。

Proof. 最初のラウンドで交換される層サンプル $\mathcal{C}_{n,\lfloor \log_2 n \rfloor}$ は葉ノード b_n のハッシュ値 $c_{n,0}$ を含んでいるため、$\tilde{i} = n$ のとき 1 回目のラウンドで $\tilde{i} = n$ を特定する。一方、$\mathcal{C}_{n,\lfloor \log_2 n \rfloor}$ は $c_{n,0}$ 以外はすべて中間ノードのハッシュであるため、$c_{n,0}$ 以外で差異を検出した場合は $\tilde{i}$ の特定のために追加のラウンドが必要となる。したがって、最良ケースは $\tilde{i} = n$ のケースで 1 回のラウンドで終了する。

最悪ケースはすべてのラウンドで層サンプル $\mathcal{C}_{i,j}$ の中から最も高い部分木 (最も左の部分木) を選択し続ける。1 回のラウンドで高さが 1 しか減少しないため、葉ノード ($j = 0$) に到達するためには木 T_n の高さである $\lceil \log_2 n \rceil$ 回のラウンドが必要である。この過程では左枝方向に降下して行くため、最終的に $\tilde{i} = 1$ に到達する。したがって、最悪ケースは $\tilde{i} = 1$ のケースで $\lceil \log_2 n \rceil$ 回のラウンドで終了する。 □

3.6 Complexity Analysis

本セクションでは、上に述べた Slate の主要な操作について計算量の理論値を分析する。分析にあたっては、ツリー内に n 個のエントリが存在し、最新のエントリ 1 つをメモリ (一時記憶装置) 上に保持している状況を想定する。

3.6.1 Append Operation

Append 操作は新しいデータからエントリを生成し、直列化形式の末尾に追加する操作である。

- Append: O(1) with cache, O(log n) worst case - Verify: O(log n) - Space: O(log n) for roots, O(n log n) for full tree

3.7 Correctness

- Proposition 1: Append 操作の正しさ - Proposition 2: 一貫性証明の健全性 - 簡潔な証明スケッチ

3.8 構造的アルゴリズムの概念

3.9 層化 (stratification) と追記の形式的定義

3.10 ノード配置とハッシュ計算規則

3.11 Slate の生成過程と更新手順

3.12 計算量と記憶効率の理論的解析

参考：Champine 論文の Algorithm 1, 2 のような明確な pseudocode を提供。Utreexo 論文のような視覚的な図解も効果的です。

3.13 Problem Statement

3.14 Slate Structure

3.15 Properties

4 (Temporal Locality Optimization

4.1 Hamming Distance-based Access Pattern

- 定義：ハミング距離とアクセス効率の関係 - Proposition 3: I/O アクセスパターンの幾何分布 - 証明：数式とともに - 視覚化：アクセス頻度の分布図

4.2 Leveled Gradual Cache Strategy

- 各レベルのキャッシュポリシー - メモリと I/O のトレードオフ分析 - 最適なキャッシュサイズの決定

4.3 Snapshot Isolation

- 並行操作のサポートメカニズム - MVCC-like semantics - 一貫性保証の証明

参考：Utreexo 論文の Section 5 のように、最適化テクニックを実用的な観点から説明。ただし、理論的な裏付けも提供。

5 Experimental Results

5.1 AWS Deployment

5.2 Performance Evaluation

5.3 Implementation

- 実装環境（AWS EC2, Ubuntu, etc.）- ベンチマーク設定 - データセット（real-world workloads）

5.4 Performance Evaluation

- Append throughput * キャッシュあり/なしの比較 * スループット vs メモリ使用量 - Verification performance - Memory consumption

5.5 Comparative Analysis

- vs History Tree - vs Merkle Search Trees - vs Prolly Trees - 表形式での比較（Table: Comparative Performance）

5.6 workload-Specific Results

- WAL scenario - Blockchain scenario - Version control scenario

参考：Crosby 論文の Table 2 のような明確な性能比較表。Utreexo 論文の Figure 4 のようなグラフで視覚化。

6 Discussion

展開の流れ：- 結果の解釈：なぜ Slate が優れているか - 適用可能なユースケース：どんなシステムに向いているか - トレードオフ：- メモリ vs I/O - 追記性能 vs 検証性能 - シンプルさ vs 最適性 - 限界と制約：- ランダムアクセスには不向き - 削除操作の複雑さ（もし実装していないなら）- 実装の考慮事項：- プロダクション環境での注意点 - チューニングパラメータ

6.1 Related Work

- Merkle Tree Variants - Binary Numeral Trees - Authenticated Data Structures - Log-Structured Storage

7 Conclusion

- 貢献の再確認（3 つの key contributions を再度）- 理論的意義：時間的局所性の形式化と証明 - 実用的意義：実世界のワークロードでの性能改善 - 今後の方向性：- 削除操作の最適化 - より強力な暗号プリミティブ - 他の分散システムへの適用 - 実装の公開（GitHub URL）

参考文献

[1] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, no. 2011. Athens, Greece, 2011, pp. 1–7.

[2] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.

[3] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. USA: USENIX Association, 2009, p. 317â 334.

[4] B. Laurie, A. Langley, E. Kasper, E. Messeri, and R. Stradling, "Certificate transparency version 2.0," *Internet Requests for Comments, RFC Editor, RFC*, vol. 9162, 2021.

[5] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," in *Proceedings of the eighteenth annual ACM symposium on Theory of Computing*, 1986, pp. 109–121.

[6] V. Almeida, A. Bestavros, M. Crovella, and A. De Oliveira, "Characterizing reference locality in the www," in *Fourth International Conference on Parallel and Distributed Information Systems*. IEEE, 1996, pp. 92–103.

[7] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1998, pp. 151–160.

[8] D. Sornette and Y. Zhang, "Transaction flows and holding time scaling laws of bitcoin," *Physica A: Statistical Mechanics and its Applications*, vol. 658, p. 130294, 2025.

付録 A　ビット演算と CPU 命令への変換

Slate の構造は二分木に基づいており、その数学的な表現において 2 のべき乗や余剰の演算が頻繁に現われる。これらは汎用プログラミング言語の多くがサポートする単純な四則演算、ビット演算、および CPU 命令を用いて高速に算出することができる。数式表記との対応表を Table 1 に示す。

数式表記	ビット演算または CPU 命令
2^j	`1 << j`
$i \bmod 2^j$	`i & ((1 << j) - 1)`
$i - (i \bmod 2^j)$	`i & ~((1 << j) - 1) = (i >> j) << j`
$\left\lfloor \frac{i}{2^j} \right\rfloor$	`i >> j`
$\left\lceil \frac{i}{2^j} \right\rceil$	`(1 + (1 << j) - 1) >> j`
$\lfloor \log_2 x \rfloor$	`ilog2(x)`: x の 2 進数表現で最も左にある 1 の位置。
$\lceil \log_2 x \rceil$	`ilog2(x - 1)`: $x-1$ の 2 進数表現で最も左にある 1 の位置。
$\mathrm{popcount}(x)$	`popcnt(x)`: x のビット内に存在する 1 の個数。
$\mathrm{ctz}(x)$	`ctz(x)`: x の最右ビットから連続する 0 の個数。count trailing zeros.
$\mathrm{clz} = \begin{cases} n-1-\lfloor \log_2 x \rfloor & \text{if } x > 0 \\ n & \text{if } x = 0 \end{cases}$	`clz(x)`: x の最左ビットから連続する 0 の個数。count leading zeros.
$\mathrm{clo}(x)$	`clz(~x)`: x の最左ビットから連続する 1 の個数。count leading ones.

表1: 数式表記からビット演算と CPU 命令への変換。

実際の演算においては、例えば i を 64-bit 整数と仮定した場合に j は $0 \leq j \leq 64$ の範囲を取ることができるため、$j = 64$ のケースでオーバーフローを起こす可能性がある。この境界値チェックのために `if(j < 64)` のような条件分岐を追加する必要があるが、現実的な使用範囲では `j < 64` は常に `true` となり、CPU の分岐予測と投機実行に高い予測成功率を期待できるため、追加の境界チェックのオーバーヘッドはほぼ無視できると考えられる。