Practical Minimal Perfect Hash Function

Susumu Yata

参考文献

- Simple and Space-Efficient Minimal Perfect Hash Functions
  - Hypergraph と Rank/Select を利用した完全ハッシュ
  - Fabiano C. Botelho, Rasmus Pagh and Nivio Ziviani
  - 10th International Workshop on Algorithms and Data Structures (WADS07)
  - 139-150, August 2007

- Bep: Associative Arrays for Very Large Collections
  - 大規模コレクション向けの連想配列
  - Daisuke Okanohara
  - http://www-tsujii.is.s.u-tokyo.ac.jp/~hillbig/bep-j.htm

説明の順序

- 概要
  - 提案手法の特徴と応用
- 基礎知識
  - ハッシュと完全ハッシュ
- 提案手法
  - コンセプト
  - ハッシュ関数 → 完全ハッシュ
  - 完全ハッシュ → 最小完全ハッシュ
- 評価
  - 時間と空間
- まとめ

提案手法の概要と成果物

- 実用的な最小完全ハッシュ関数
  - 規模　　大規模なキー集合（1000 万件でも大丈夫）
  - 時間　　構築時間は O(n), 検索時間は O(1)
  - 領域　　理論的下限値の 2 倍程度（3 bits/key）

- 公開ライブラリとライセンス
  - http://cmph.sourceforge.net/
    - LGPL
  - http://www-tsujii.is.s.u-tokyo.ac.jp/~hillbig/bep-j.htm
    - 修正 BSD ライセンス
  - http://homepage1.nifty.com/herumi/diary/0711.html
    - 特に指定なし

最大の特徴

- 3 bits/key は伊達じゃない

キー数	サイズ	補足
10 万	37.5 KB	キャッシュ
100 万	375 KB	キャッシュ
1000 万	3.75 MB	キャッシュ & メモリ
1 億	37.5 MB	メモリ
10 億	375 MB	メモリ

- 例えば，ダブル配列…
  - 理論的に 8 bytes/key 以上
  - 1 億 → 800 MB 以上 … むりぽ

その他の特徴

- マッチング
  - 完全一致のみ　　→ ダブル配列（トライ）とは別用途
- 未登録キーの扱い
  - 登録／未登録を判断できない
  - 何かが出てくる　　→ あらためて確認が必要
- 時間
  - ハッシュ値を 2, 3 用意する
  - Rank 関数も使う　　→ こいつらがネック

その特徴を生かすには？

- 記憶装置の特性を考慮
  - 速い　キャッシュ ＞ メモリ ＞ HDD　遅い
- ネックが緩和されるのは…
  - 低速な記憶装置にデータを置く状況
- インデックスとして利用
  - Case 1:　ハッシュ関数 on キャッシュ　→ インデックス
    キー本体 on メモリ　→ データ
  - Case 2:　ハッシュ関数 on メモリ　→ インデックス
    キー本体 on HDD　→ データ

提案手法の応用先

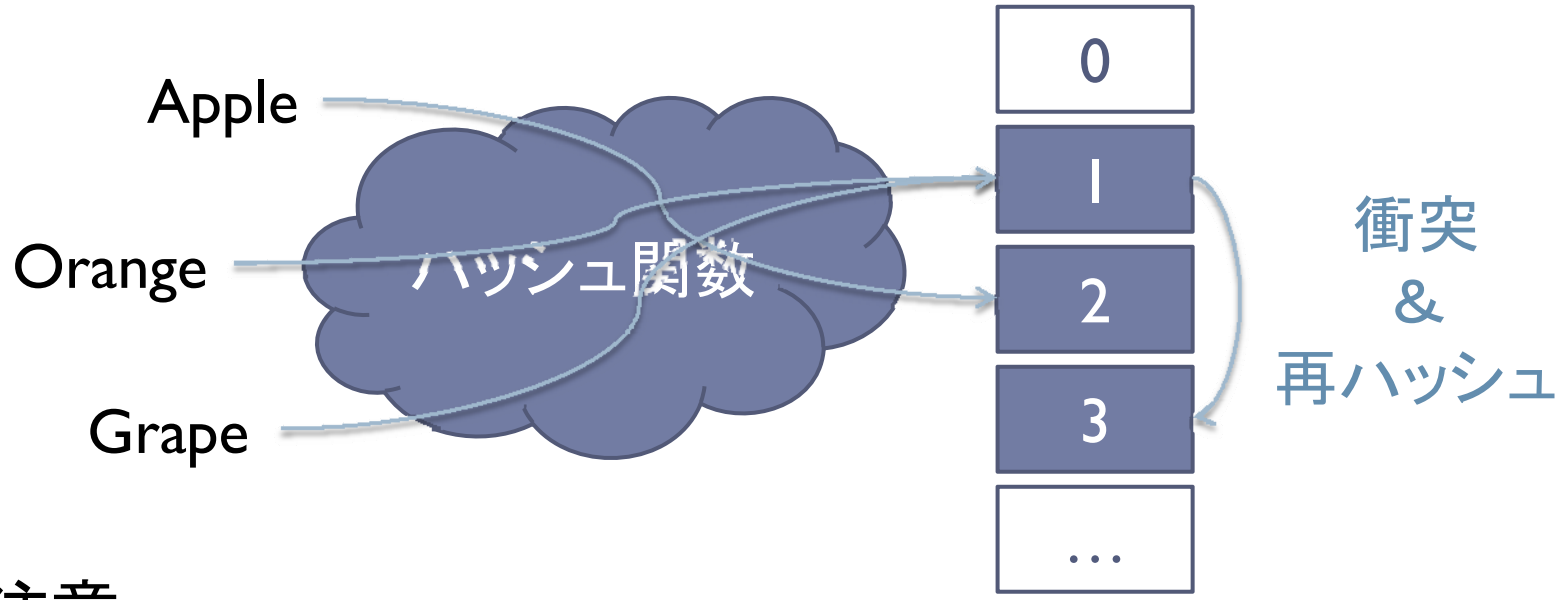
- Web アーカイブ
  - 入力　URI　平均 100 Bytes
  - 出力　DATA　平均 10 KB
  - 規模　1 億ページ
    - URI リスト　10 GB　on HDD or SSD
    - DATA リスト　1 TB　on HDD
- 参照方法
  - 完全ハッシュ（37.5 MB）→ URI リスト → DATA リスト ☺
- 課題
  - 構築時の作業領域については未検討 😐

基礎知識

ハッシュ関数から最小完全ハッシュ関数まで

ハッシュ関数

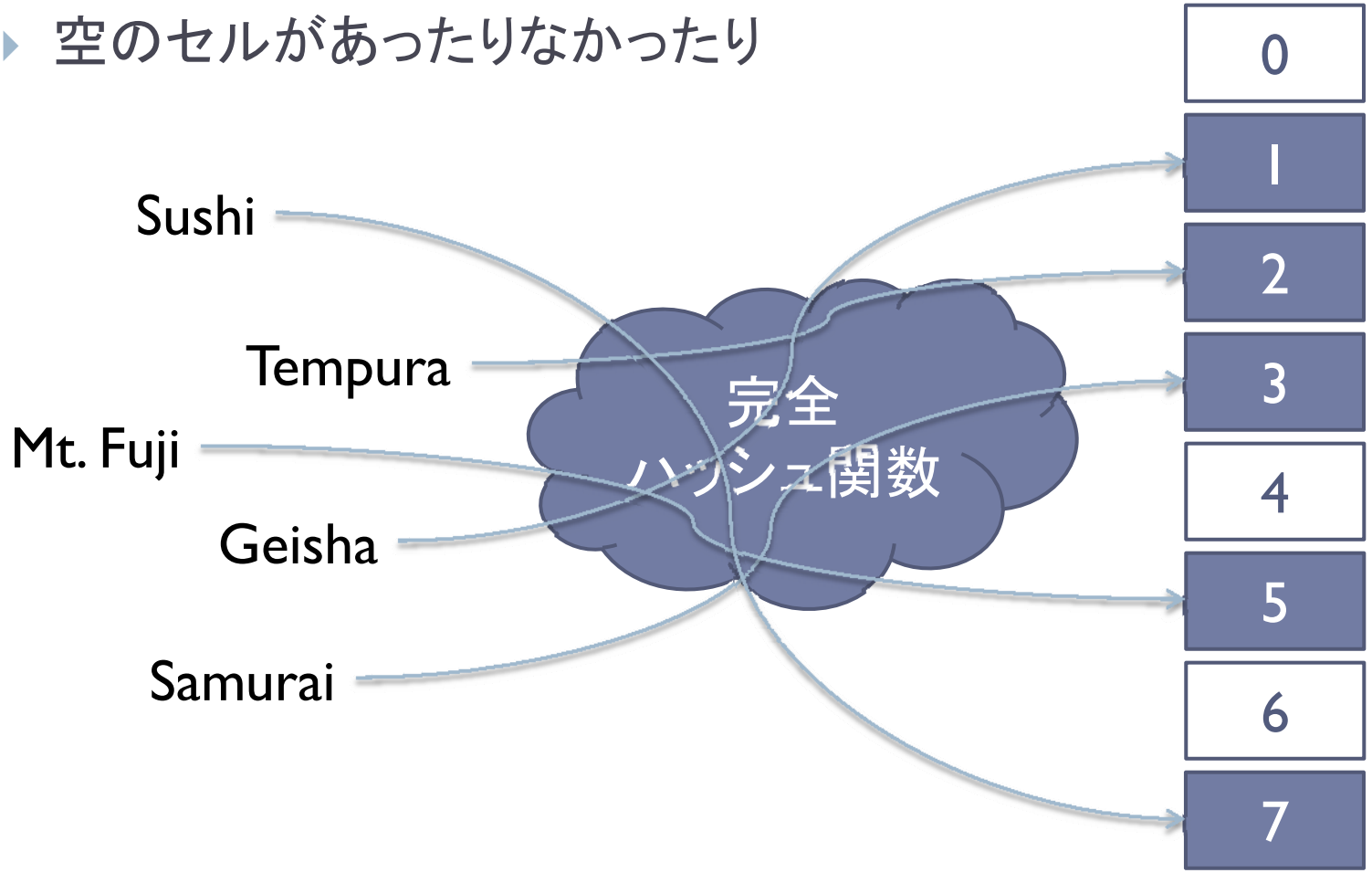
- ハッシュ表にて利用
  - キーをインデックスに変換　　O(1)
  - 衝突したときは再ハッシュ　　少ないほど良い



- 注意
  - MD5, SHA-1　　暗号理論にて頻出のハッシュ関数

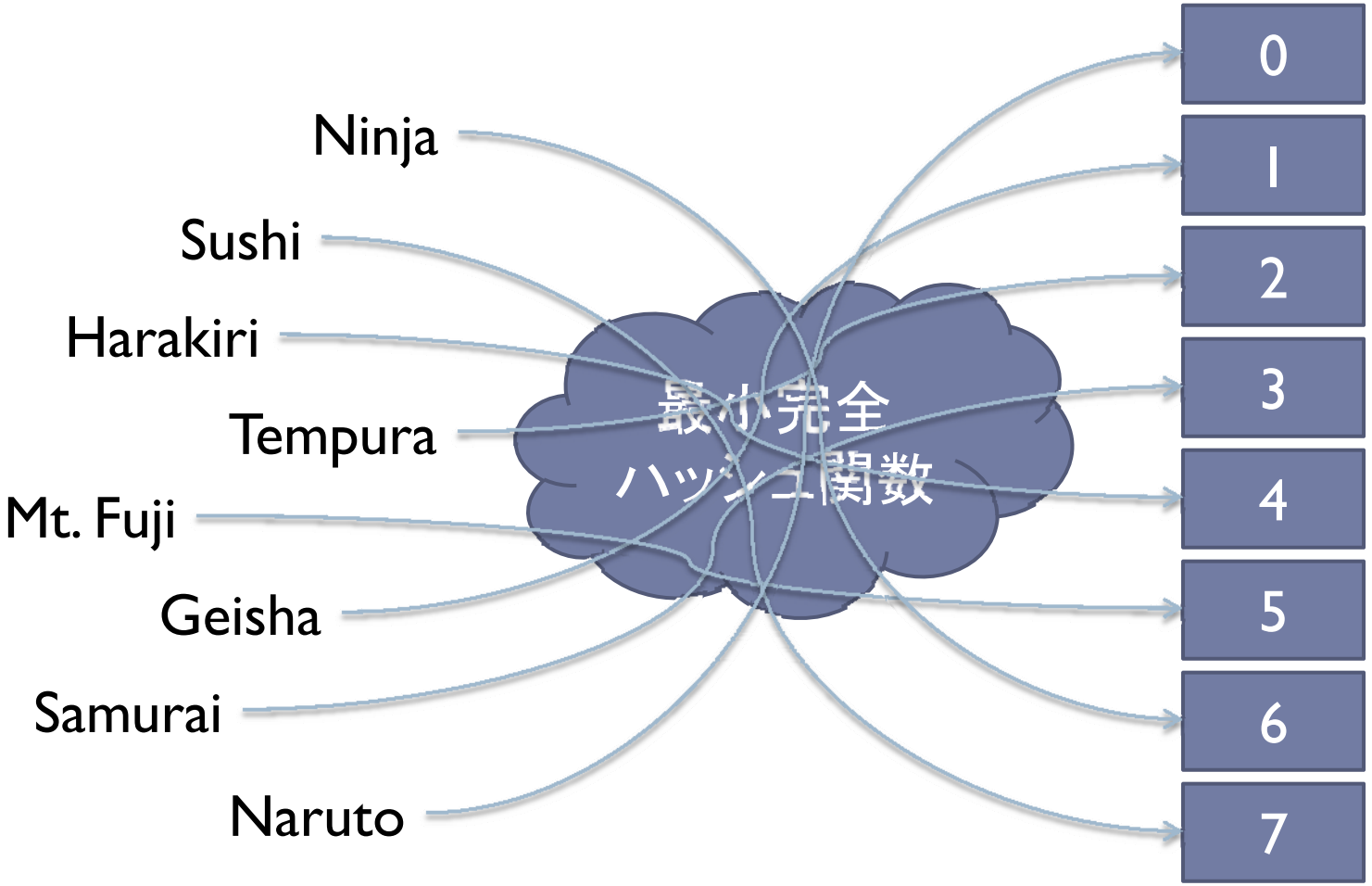
完全ハッシュ関数

- 衝突のないハッシュ関数
  - 空のセルがあったりなかったり



最小完全ハッシュ関数

- 衝突も空きもないハッシュ関数



提案手法のコンセプト

イメージをつかむ

提案手法のコンセプト

1. 完全ハッシュ関数の生成

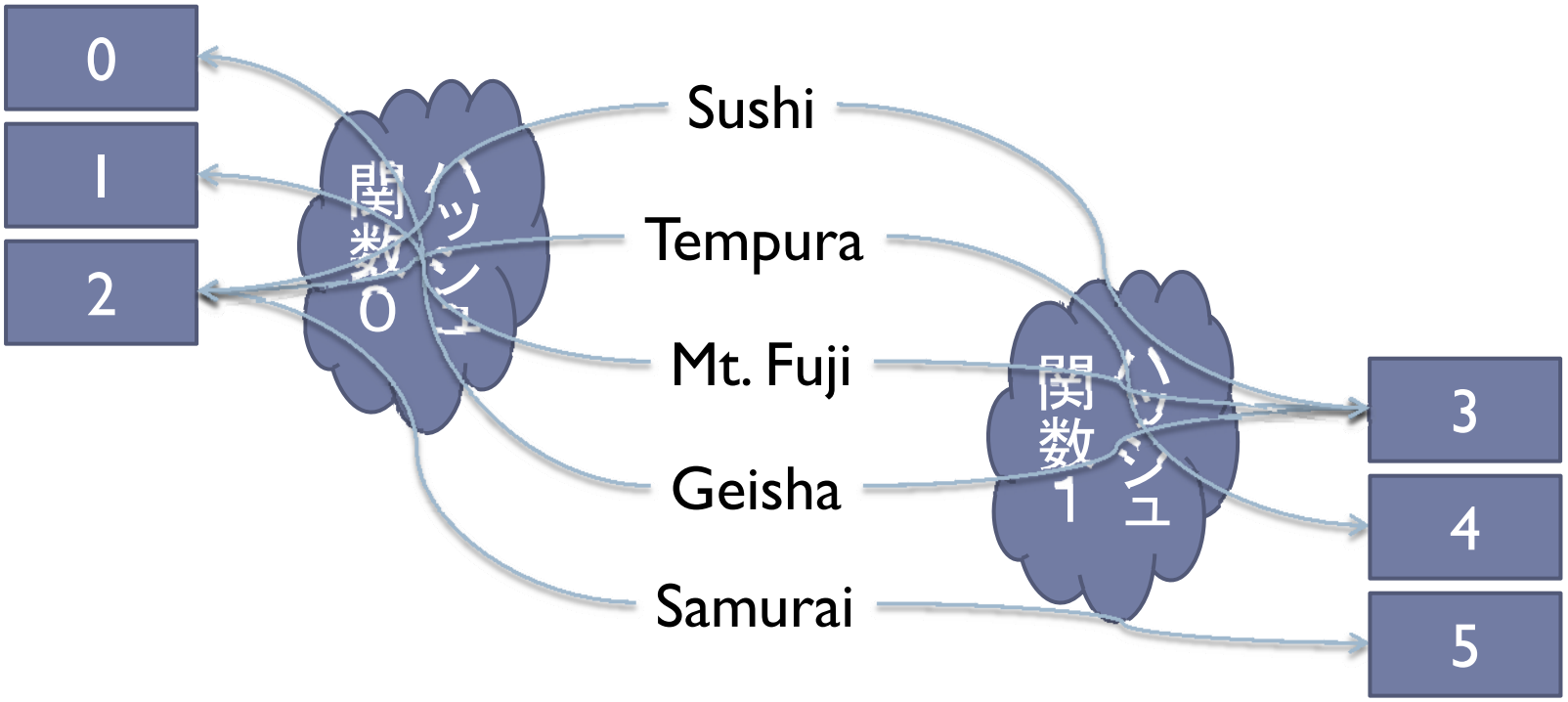
   - ハッシュ関数を 2, 3 用意
   - 合成して完全ハッシュ関数になるか検証
   - ダメなら再チャレンジ

2. 最小完全ハッシュ関数への変換

   - Rank 関数による穴埋め
     - Practical Entropy-Compressed Rank/Select Dictionary
       - Okanohara, Daisuke and Kunihiko Sadakane
       - In the Proceedings of ALENEX 2007

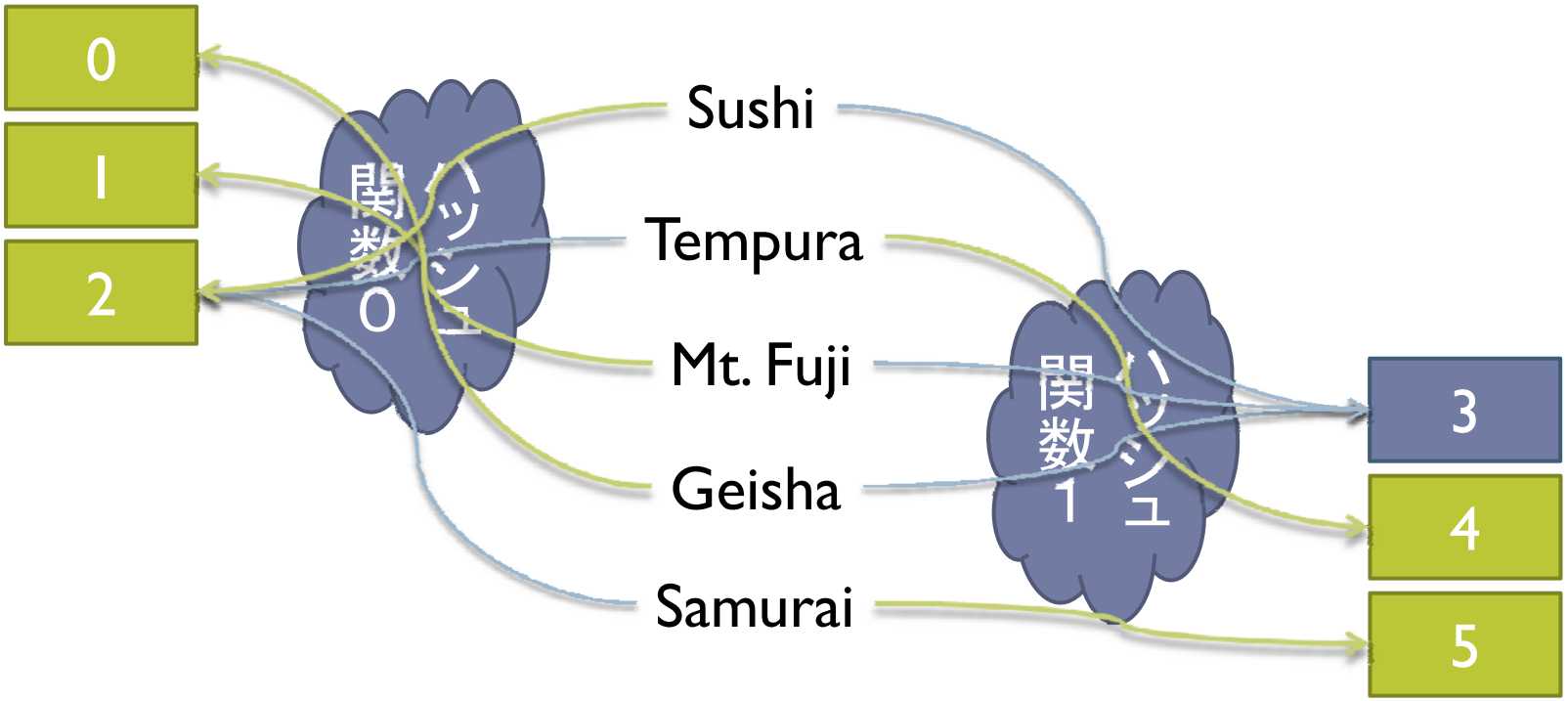
基になるハッシュ関数の用意

- 汎用的なハッシュ関数を 2, 3 用意
  - ハッシュ表も 2, 3 用意
  - 大きさには少し余裕を持たせる



ハッシュ関数の合成

- 衝突しないようにハッシュ関数を
  - に　み合　せる
  - うまくいかないときはハッシュ関数を用意しな　す

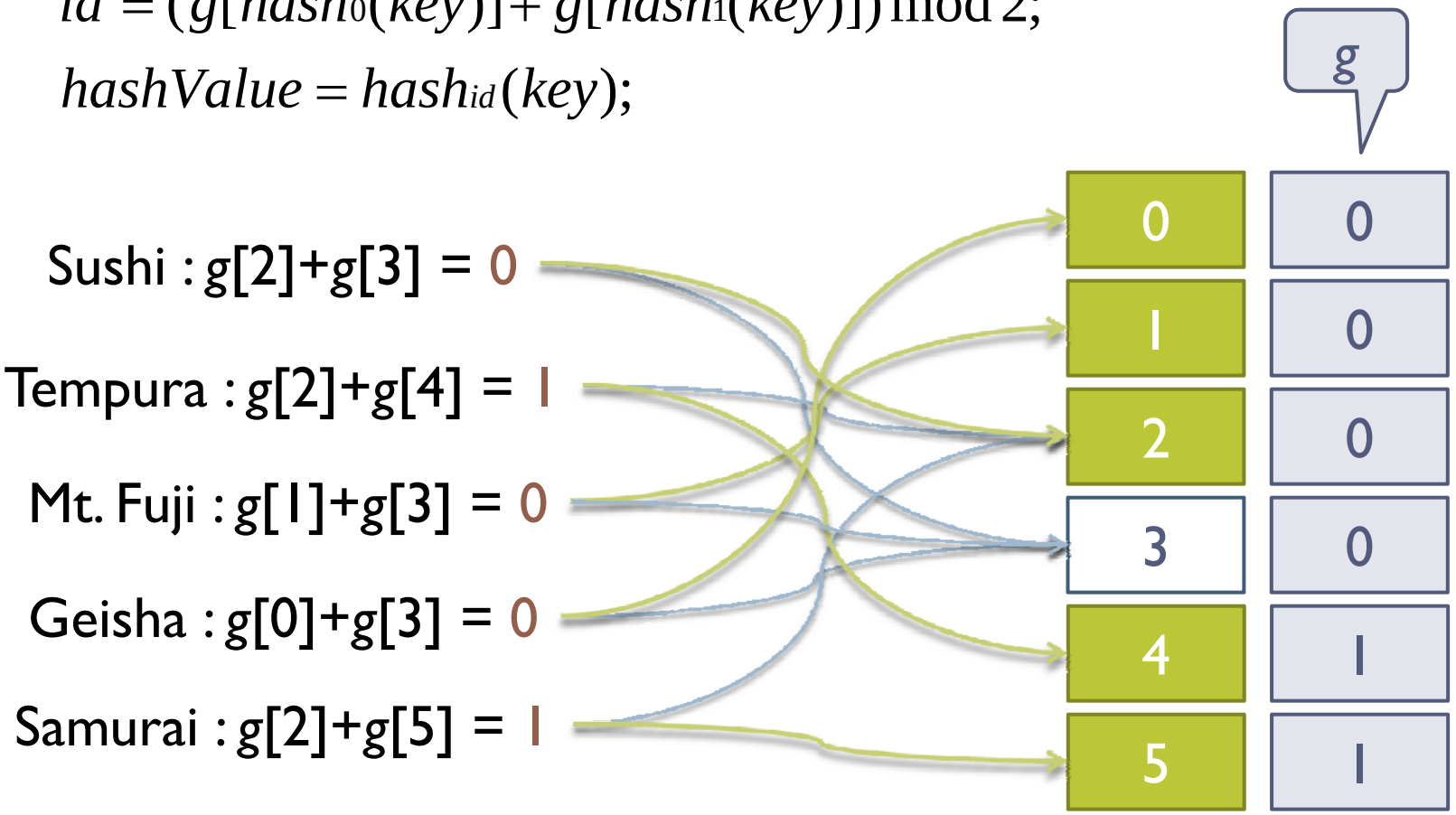


ハッシュ関数の組み合わせを数値化

- ーブル g を用意

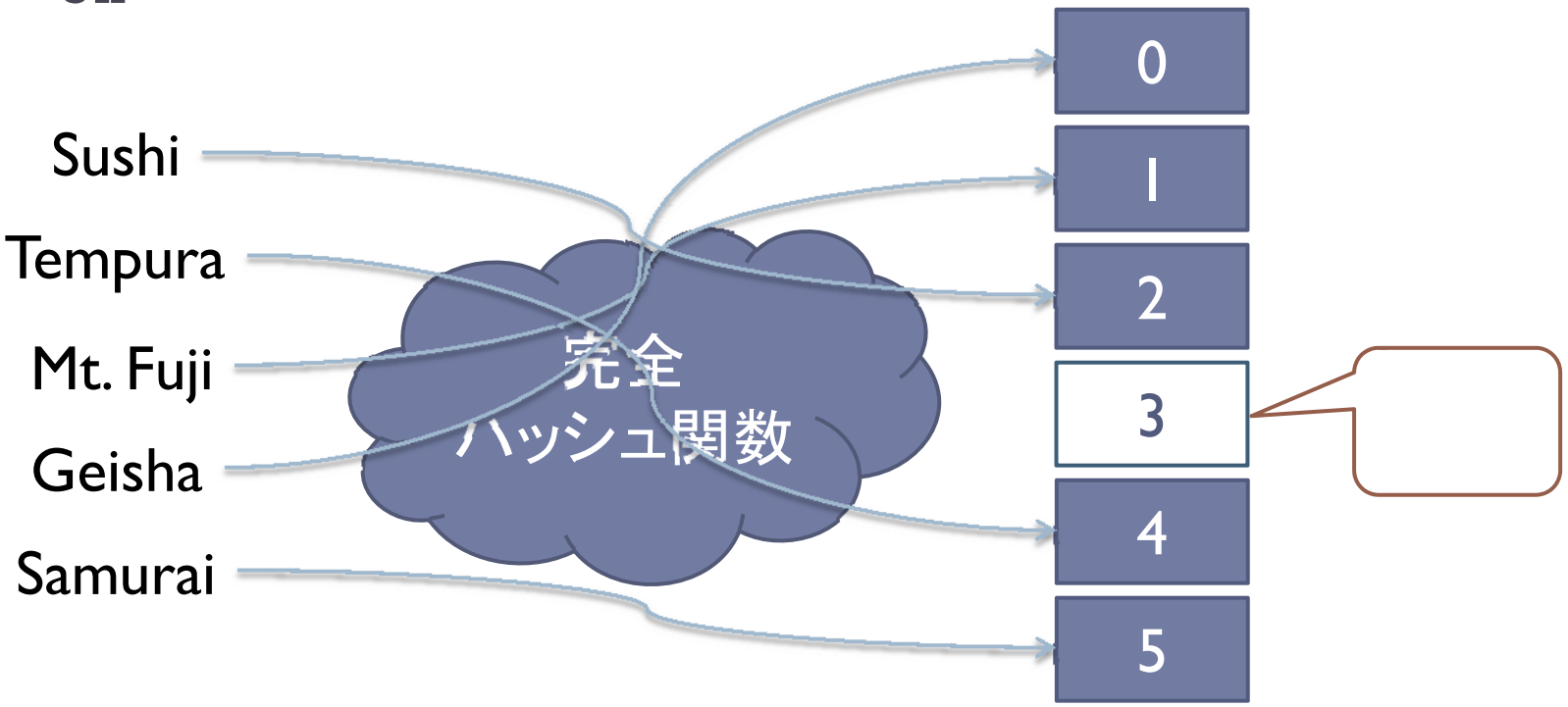
$$id = (g[hash_0(key)] + g[hash_1(key)]) \bmod 2;$$

$$hashValue = hash_{id}(key);$$



完全ハッシュ関数ができた

- 構成要
  - *hash0()*　　ハッシュ関数　の
  - *hash1()*　　ハッシュ関数　の
  - *g[]*　　ーブル



そして最小完全ハッシュ関数へ

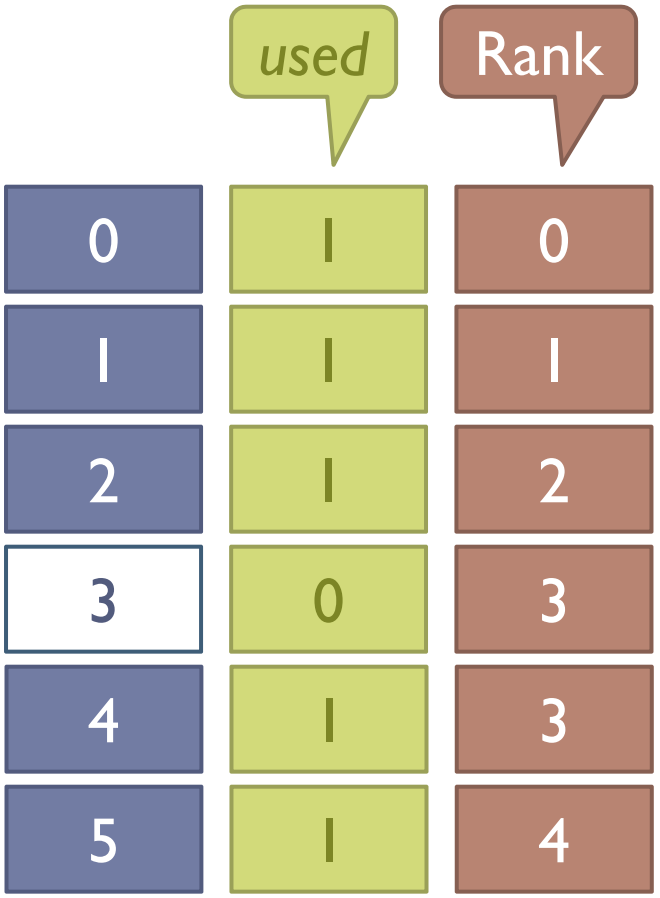
- 下
  - *used*[] : 使用(1)／未使用(0)の　ット列
  - Rank 用のインデックス

- Rank 関数
  - 　ット列 *used*[] に　して

$$\mathrm{Rank}(used, x) = \sum_{i=0}^{x-1} used[i]$$

  - 時間　　O(1)

- 最小完全ハッシュ関数
  - Rank の値を　すのみ



提案手法の実装

ハッシュ関数の用意

基になるハッシュ関数の工夫

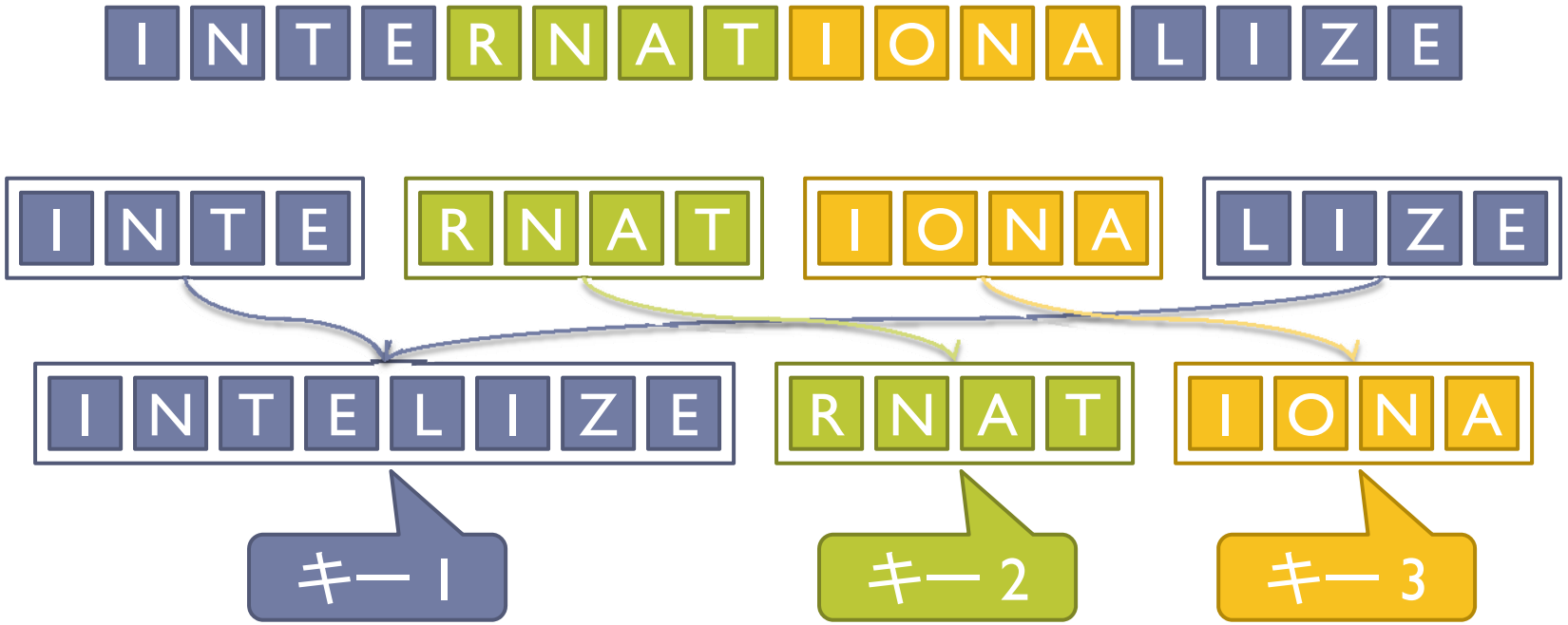
- 汎用的なハッシュ関数（bep にて　用）
  - http://www.burtleburtle.net/bob/hash/evahash.html
  - 速で均一（CPU の知識が必要）
  - 定の変　による　り　えが

- ハッシュ関数を 3 つ　←　的に良い　数
  - 1　用意するのが
  - 2　に時間がかかる

- 案
  - キーを 3

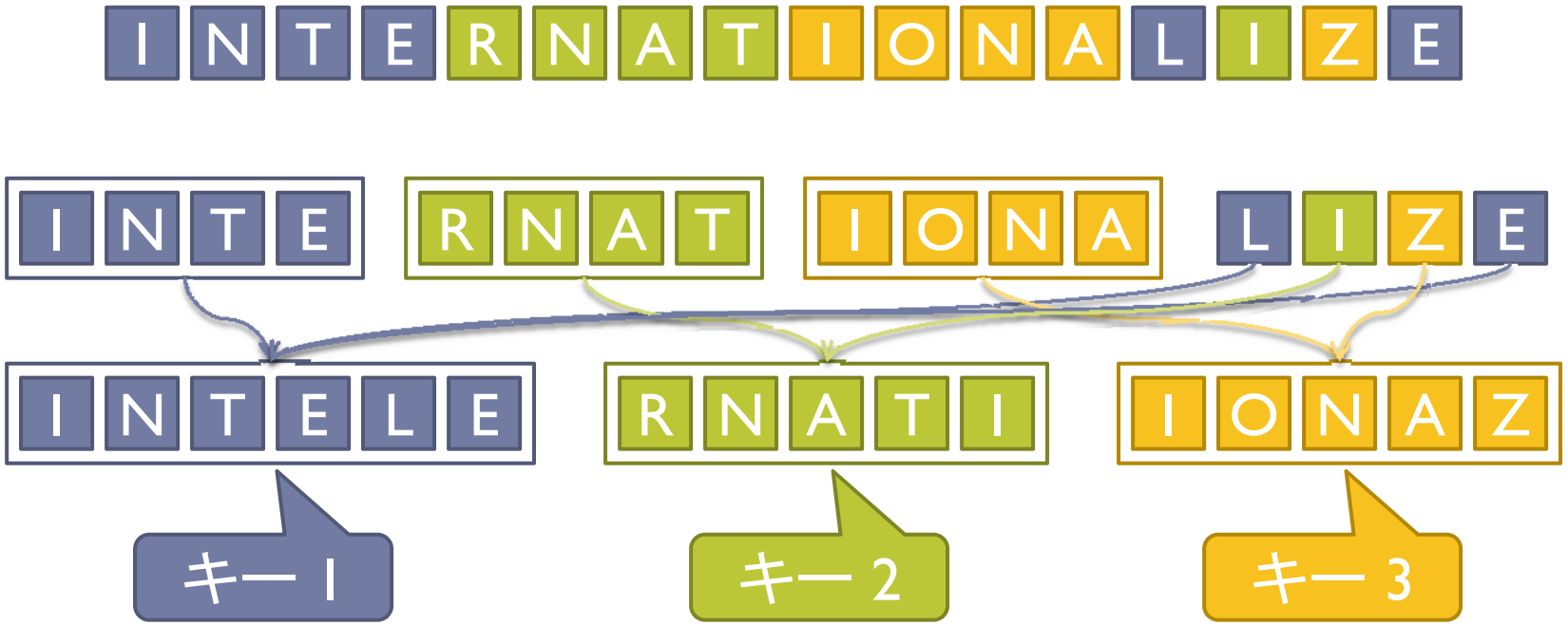
キーを 3 分割

- 　速　の　イント
  - 32 or 64　ット　つ　理（　ッキング）
  - ハッシュ関数を I　　け使う
  - 例　INTERNATIONALIZE → INTELIZE + RNAT + IONA



キーが短いときは

- キーを　　できないので
  - 最　ブ　ックを　ッキングしない　　← 未検証
  - 例　INTERNATIONALIZE → INTELE + RNATI + IONAZ
  - 例　HASH → HH + A + S



ハッシュ値の算出

- ハッシュ値の　りうる値
  - キー数 n に　して　0, 1.23n）
    - ハッシュ関数が均一と　定して
    - 1000　の　　でほ　100% 成　する大きさ
  - 　　キーに　り　てる
    - ブ　ックの大きさ m: 1.23n / 3 = 0.41n
    - キー 1　0, m）, キー 2　m, 2m）, キー 3　2m, 3m）

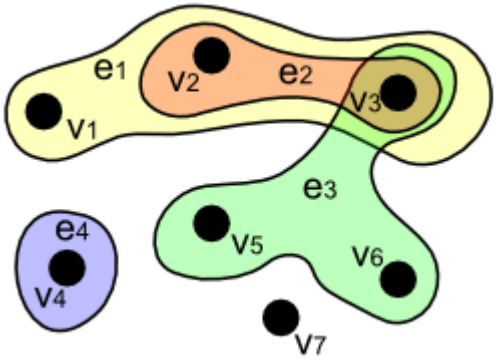
- ハッシュ値の　め方　　← 実　には一度の　　出し
  - ハッシュ値 1　$hash(key_1)$ % m
  - ハッシュ値 2　m + ($hash(key_2)$ % m)
  - ハッシュ値 3　2m + ($hash(key_3)$ % m)

提案手法の実装

完全ハッシュ関数の実

Hypergraph（ハイパーグラフ）

- ハイ　ーなグラ　のこと　←　からなくても大丈夫
  - X　　の集合　　の　数　　ッジの数
  - E　　ッジの集合　　ッジの　度　　の数
    - ッジは X の　　集合（空集合を　く）



$$X = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$E = \{e_1, e_2, e_3, e_4\} = \{\{v_1, v_2, v_3\}, \{v_2, v_3\}, \{v_3, v_5, v_6\}, \{v_4\}\}$$

ハイ　ーグラ　の例（ハイ　ーグラ　–Wikipedia）

特殊なハイパーグラフ

- *k*-uniform hypergraph
  - ッジの　度が *k*
    - ッジが持つ　　の数が *k*　という意
    - 一　的なグラ　は 2-uniform hypergraph

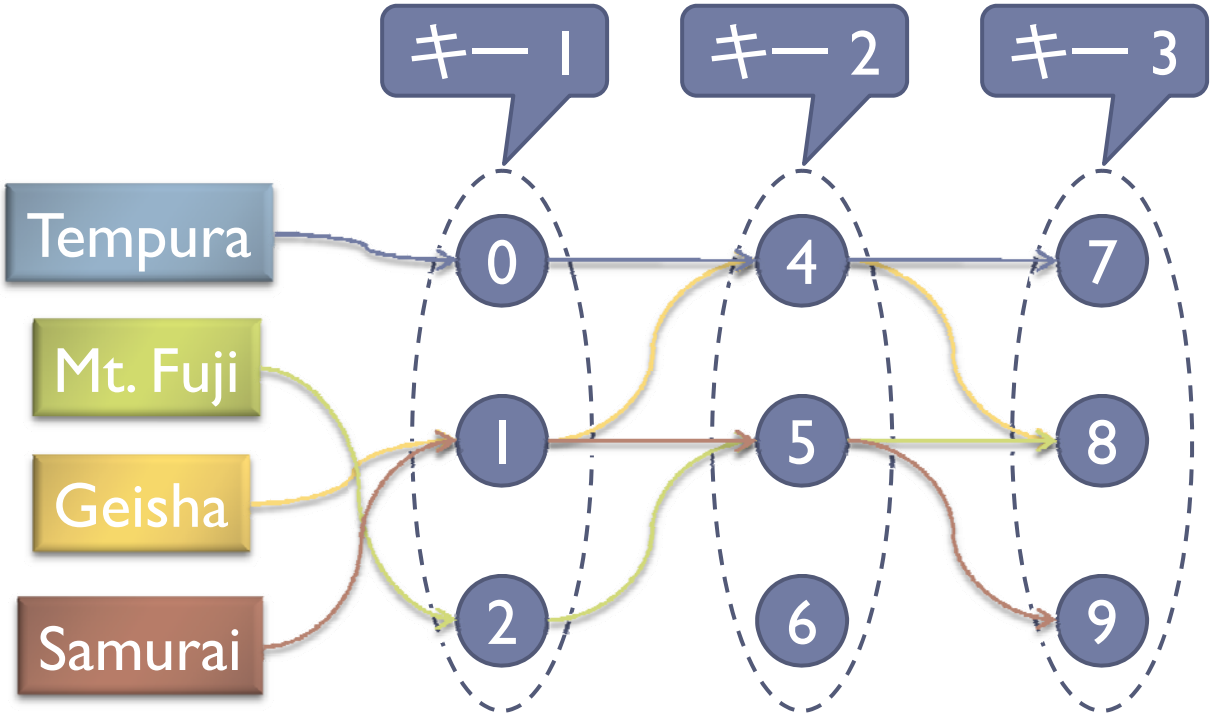
参考 ハイ　ーグラ　–Wikipedia

- *k*-partite hypergraph
  - *k*　に　　できるハイ　ーグラ
    - の例は 3-partite hypergraph
      - v_1, v_2, v_3, v_5, v_6
      - v_4
      - v_7

いたいこ　な　じ

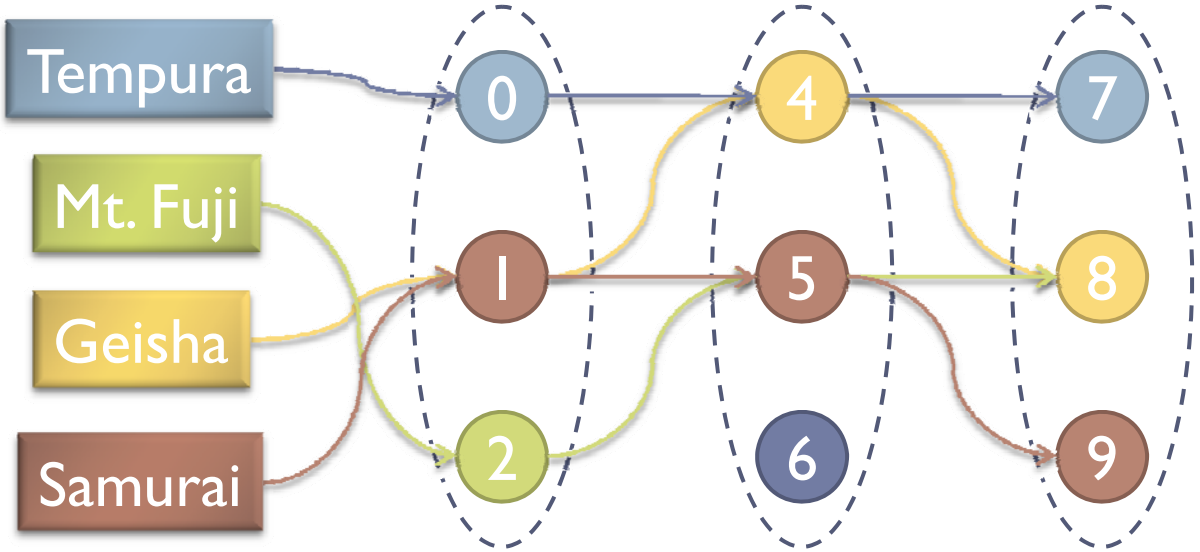
ハッシュ値からハイパーグラフを生成

- ハイ　ーグラ　の構成
  - 3-partite uniform hypergraph
  - キー数 n に　して1.23n　の　　（$m = 0.41n$　つ）
  - 　キーのハッシュ値 3 つからなる n　の　ッジ



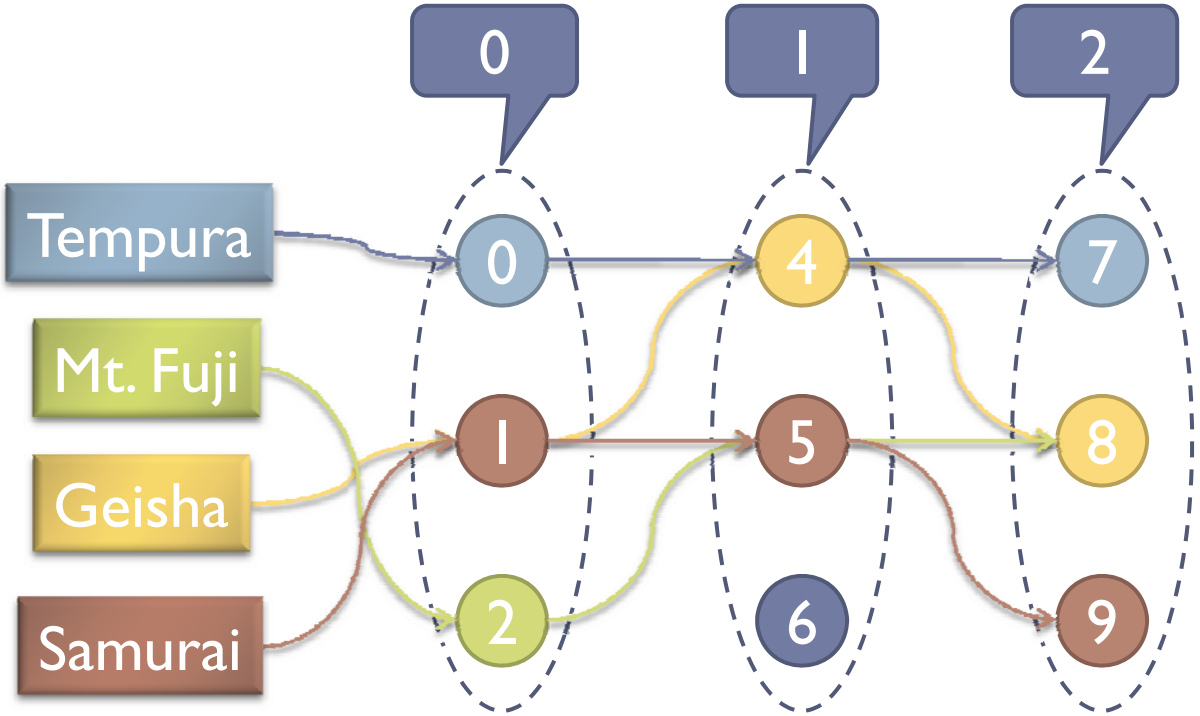
Acyclic（非循環）かどうかを検証

- Acyclic とは　　← この論　での定
  - 以下の手　によりす　ての　ッジを　　できるかどうか
  1. 　　数 I の　　を　む　ッジを　す
     1. 最　は Tempura, Mt. Fuji, Samurai のどれか
  2. 　　つからなければ　　する
  3. 　　ッジを　り　いて最　に　る



Acyclic だとどうなるの？

- 完全ハッシュ関数ができる
  - り　いた　ッジ(キー)の　　を利用する
  - 　　にハッシュ値を　り　てる
  - 時に　　ーブルを生成する

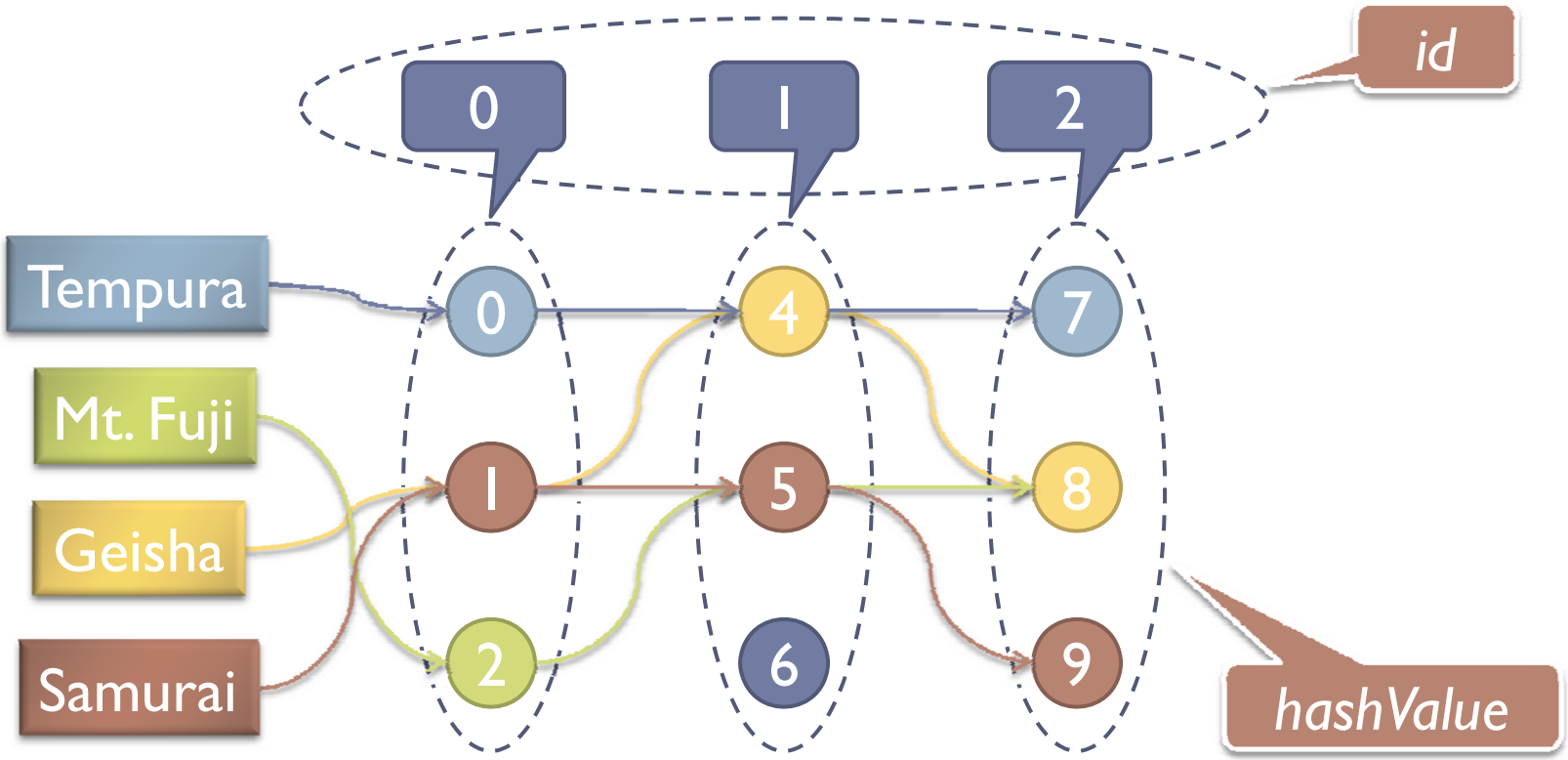


選択テーブルを生成

- どのハッシュ値を　用するのか　める

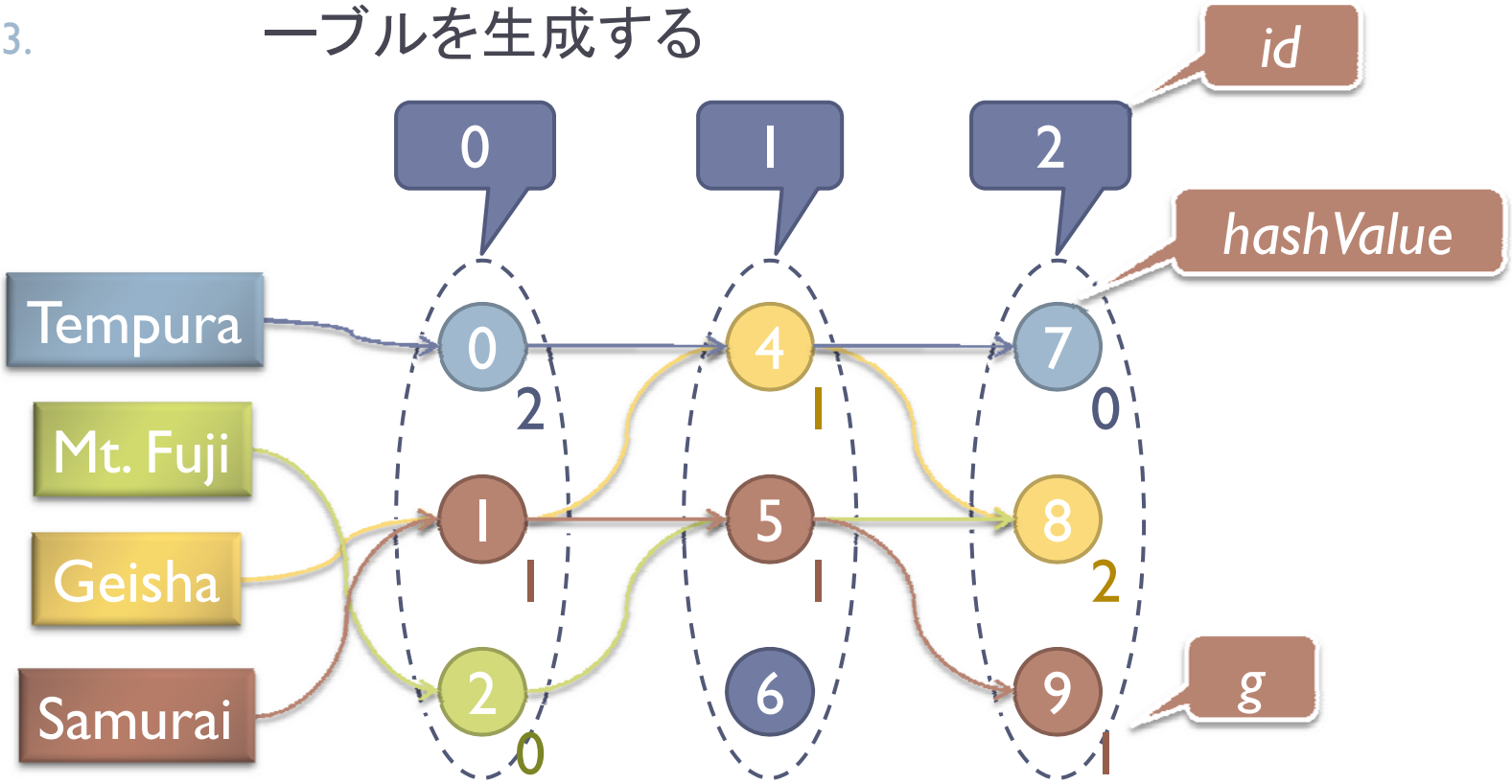
$$id = (g[hash_0(key)] + g[hash_1(key)] + g[hash_2(key)]) \bmod 3;$$

$$hashValue = hash_{id}(key);$$



アンコール（Encore）

- もうー
  1. 　　キーのハッシュ値から　ッジを生成する
  2. 完全ハッシュ関数になるかどうか判定する
  3. 　　ーブルを生成する



提案手法の実装

最小完全ハッシュ関数の実

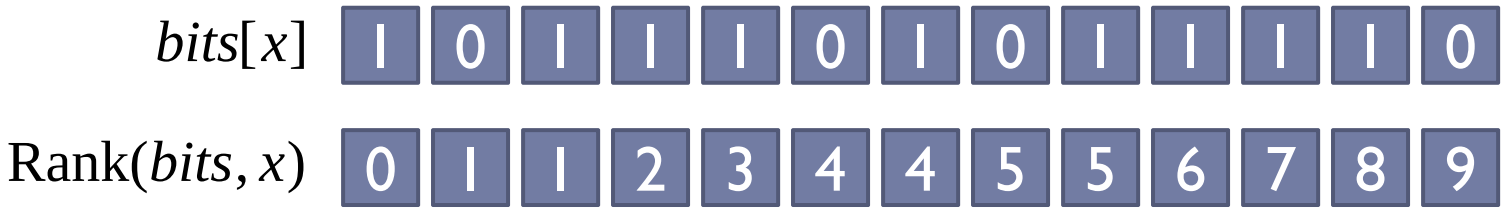
Rank 関数

- Rank 関数の定
  - ット列 *bits*[] に　して

$$\mathrm{Rank}(bits, x) = \sum_{i=0}^{x-1} bits[i]$$

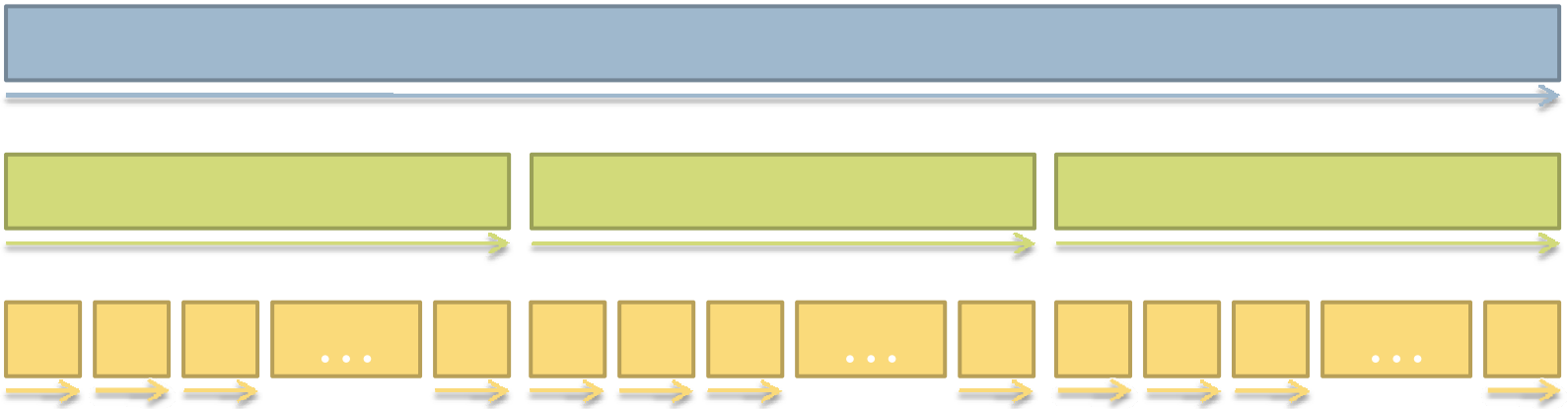
  - *bits*[0] から *bits*[*x*-1] までに　まれる　ット 1 の数

- Rank 関数の時間
  - インデックスなし　O(n)
  - インデックスあり　O(1)



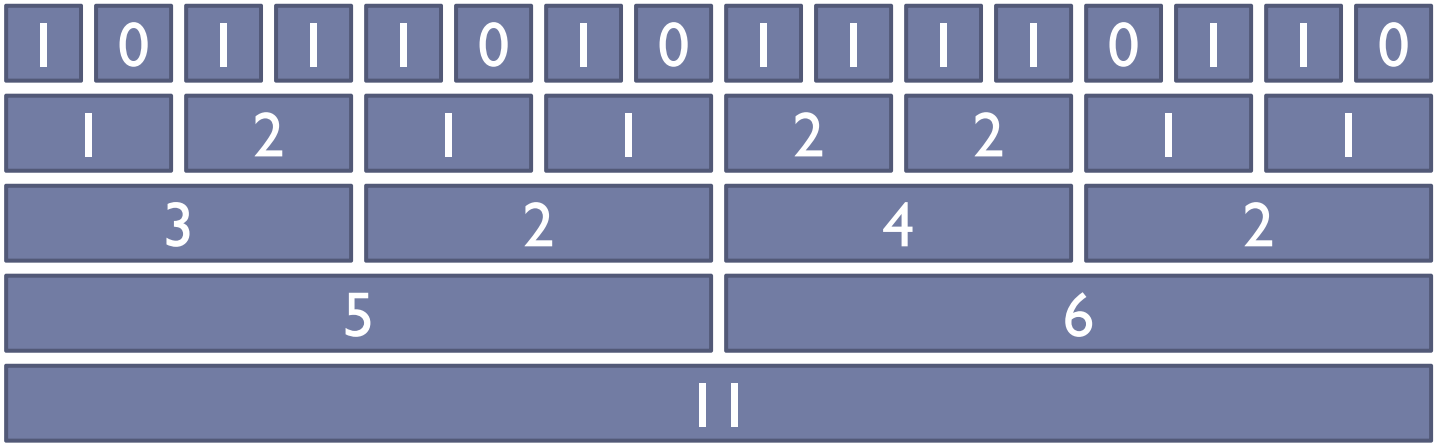
Rank 辞書

- Rank 関数の　速
  - 　ット列を 2　　のブ　ックに　　する
    - 大きめのブ　ック 256　ット
      - 　ブ　ックの　　に　する Rank() を　← か　うじて O(1)
      - 32 bit / large block　← 0.125 bit / bit
    - 小さめのブ　ック 32　ット
      - 大きめのブ　ック　での Rank() を　← まさしく O(1)
      - 8 bit / small block　← 0.25 bit / bit



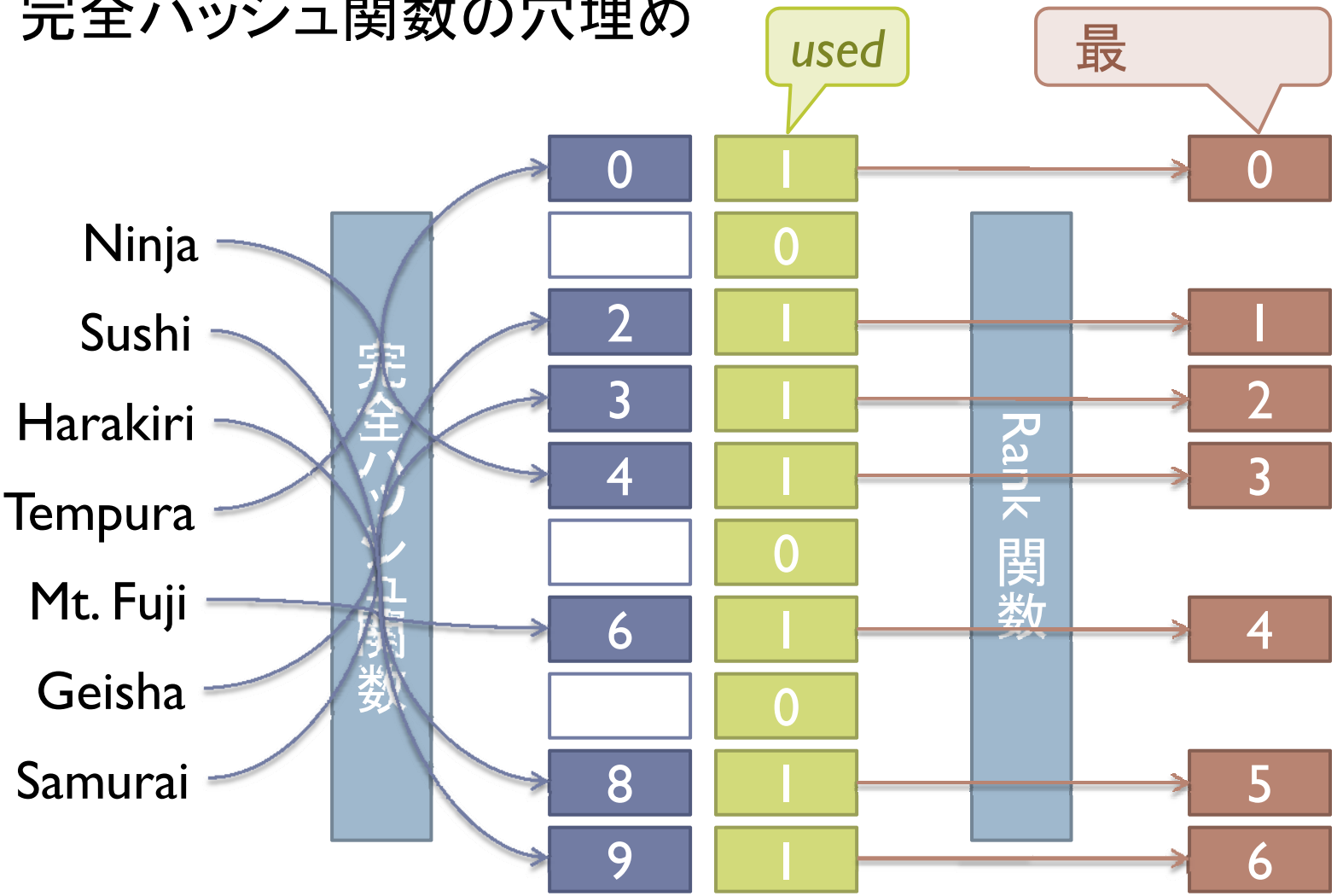
小さめブロック内での計算

- は 3 つくらい
  - SSE4 POPCNT
  - 夫なし 32 のループ
  - 夫あり ット を使ってマージ
    - 参考 1 “PopCount” で Google 生に る
    - 参考 2 “Hacker’s Delight”, Addison-Wesley, 2002
    - 参考 3 “ハッカーのたのしみ”, スアイ ーアクセス, 2004



Rank 関数で何をするんだっけ？

- 完全ハッシュ関数の穴埋め



評価とまとめ

り

理論的な評価

- 空間
  - ーブル　1.23*n* x 2 = 2.46*n* bits
  - Rank　1.23*n* x 0.375 = 0.46*n* bits
  - 合　2.92 bits/key

- 時間
  - 参照　O(1)
    - ハッシュ関数 O(1) + Rank 関数 O(1)
  - 構築　O(*n*)
    - ハッシュ関数 O(*n*) + ハイ　ーグラ　生成 O(*n*) + Acyclic 検証 O(*n*) + ーブル生成 O(*n*) + Rank 関数 O(*n*)
    - 1000　の　でほ　100%　まる想定

まとめ

- まとめ
  - ハッシュ関数の合成による最小完全ハッシュ関数の実
    - 3-partite uniform hypergraph と Rank 関数を利用
  - 　　の最小完全ハッシュ関数とは　って実用的
    - O(*n*) で構築して，O(1) で参照できる
    - サイズは理論的下限の 2, 3 倍くらい
- 　　の課題
  - 構築時の作業領域を　　したい
    - デ　スク上でな　とかならないか
  - 応用したい
    - 応用例（Web アーカイブ）みたいなもの