DEIM Forum 2011 C7-4

編集距離制約下におけるトライを用いた高速並列類似結合

成田　和世† 　中台　慎二†

† 日本電気株式会社 サービスプラットフォーム研究所　〒 211–8666 神奈川県川崎市中原区下沼部 1753
E-mail: †k-narita@ct.jp.nec.com, ††s-nakadai@az.jp.nec.com

あらまし　2 つの文字列データの中から類似する文字列ペアを全て列挙する**文字列類似結合** (*string similarity join*) は幅広い応用分野を持つ重要な技術である．潜在的に全ての文字列ペアに対して距離計算を伴う**突き合わせ** (*matching*) を行う必要があるため，高コストであることで知られている．この分野に関する多くの研究は逐次処理における議論に留まっている．デジタルデータの大規模化や分散化が進む中，文字列類似結合の並列化は重要な課題である．そこで本稿では，距離尺度が**編集距離** (*edit distance*) である制約の下，平均文字列長が比較的短い文字列データを対象とし，文字列類似結合をより高速に行う並列処理フレームワークを提案する．このフレームワークでは，2 つの絞り込みの工夫を導入することで高速化を図る．まず文字列の接頭辞に基づき，不要な突き合わせが発生しないように入力文字列データを分割する．次に，残りの接尾辞の情報を用いることで，不要な文字列ペアの編集距離計算をより早く打ち切りながら並列に突き合わせを実行する．更に，多数の突き合わせ処理をより高速に処理するために，逐次処理における最新の関連研究 Trie-Join [11] を拡張した新たなアルゴリズムを提案する．実データを用いた性能評価実験では，Trie-Join と比較することで提案手法が 2 つの絞り込みにより高速に動作することを示す．また，スケーラビリティの検証を行う．

キーワード　文字列，類似結合，編集距離，並列処理

Parallel String Similarity Joins using a Trie with Edit Distance Constraints

Kazuyo NARITA† and Shinji NAKADAI†

† NEC Service Platform Laboratory　1753 Shimonumabe, Nakahara, Kawasaki, Kanagawa, 211–8666 Japan
E-mail: †k-narita@ct.jp.nec.com, ††s-nakadai@az.jp.nec.com

Key words　string, similarity join, edit distance, parallel processing

1. はじめに

2 つの文字列データの中から類似文字列ペアを全て列挙する**文字列類似結合** (*string similarity join*) は，幅広い応用分野を持つ重要な技術である．考えうる産業利用として，例えば，マーケティング分析のためのデータ統合やデータクリーニング，情報検索における類似クエリの検出，バイオインフォマティクス，医療カルテの統合などが挙げられる．

文字列間の距離 (類似度) を測る尺度は，編集距離や Jaccard 係数など数多く存在する．その中で，編集距離は文字に対する挿入，削除，置換に着目した，オペレーションベースの尺度である．トークンベースの尺度と異なり言語の特性やトークンへの重みなどを考慮することなく算出出来る利点がある．本稿では編集距離に焦点を置き議論する．

文字列類似結合では，潜在的に全ての文字列ペアに対して距離計算を伴う**突き合わせ** (*matching*) を行う必要があるため，入力文字列データに含まれる文字列数や，距離の閾値 τ の増加に伴い処理コストが増大する．この問題に対し，これまで様々な高速化手法が提案されてきた [1], [2], [4], [6], [10]〜[13]．編集距離制約下における多くの既存手法は，*filter-and-refine* アプローチに基づく．このアプローチでは，まず各文字列からシグネチャを生成し，それを用いて類似文字列ペアの候補集合を得る (filter フェーズ)．そして，候補集合の中から真の類似文字列ペアを発見する (refine フェーズ) [1], [2], [6], [12], [13]．しかしながら，filter-and-refine アプローチは概して，平均文字列長の小さな文字列データを扱う際には候補集合を絞り込む適当なシグニチャを生成することが難しく，処理性能が低下する傾向にある．これに対して 2010 年に J. Wang らが提案した Trie-Join [11] は，平均文字列長が比較的短い文字列データを対象としており，**トライ** (*trie*) を用いることで候補集合を生成す

ることなく高速に類似文字列ペアを列挙する．J. Wang らは実験で，平均文字列長が高々30 程度の文字列データを対象としたとき，閾値 $\tau \leqq 3$ に対して，複数の最新の filter-and-refine アプローチより Trie-Join が高速であることを示した．しかし，メモリ溢れの問題や，編集距離の閾値 τ の増加に伴い処理速度が急激に低下していく問題は依然として存在する．平均文字列長が比較的短い文字列は，氏名や住所，電話番号，商品の名前，商品番号など様々考えられ，このような文字列を有するデータに対する高速な類似結合への期待は今後ますます高まると予想される．

ところで，処理高速化を図る手段の一つとして，並列コンピューティングが知られている．文字列類似結合の分野では未だ多くの研究が逐次処理における議論に留まっているものの，近年 Hadoop などの台頭により大規模データに対する並列処理技術への関心が再び高まってきていることから，徐々に注目が集まりつつある [10]．

そこで，本稿では平均文字列長が比較的短い文字列データを対象とし，編集距離制約下で文字列類似結合を高速に行うための並列処理フレームワークを提案する．なお，本稿で想定する並列環境は shared-nothing 方式で，Σ を文字列データにおける文字のドメイン，N を並列数とするとき，$N \leqq |\Sigma|$ とする．

本稿の貢献は次の通りである．

文字列類似結合の並列処理フレームワークの提案

本フレームワークは次に述べる 3 つの大きな特徴を備えている．

a） **不要な突き合わせを除去するデータ分割手法**

本稿で提案するデータ分割手法はハッシュ結合アプローチに則る．ハッシュ結合は，古くから研究されている等価結合 (equi-join) 技術において，中でも高速なアプローチとして知られる．これは，入力データを一度スキャンし，結合キーとなる属性値にハッシュ関数を適用することでタプルをあるバケットに分配した後，バケット内のタプルペアに関してのみ突き合わせ処理を行うものである．一致しそうもない属性値を持つタプル同士は異なるバケットに分配され，突き合わせが発生しない．そこで我々はこの考え方に基づき，入力データを一度スキャンし，類似しそうもない文字列同士を異なるバケットに分配する．これにより，後で発生する突き合わせの回数をブルートフォース手法と比べて削減することが出来る．等価結合と異なり，類似結合では結合キーに対応する文字列そのものをハッシュ関数に適用することは不可能であるため，文字列を分配する手段として，本手法は filter-and-refine アプローチで候補集合を絞り込むためにしばしば利用される prefix pruning の原理を応用している．

b） **不要な突き合わせをより早く打ち切る類似性評価**

バケットに振り分けることで，ある程度不要な突き合わせ処理を刈り取った後も，なお突き合わせ回数は多い．そこで本稿では，バケットごとに並列に突き合わせ処理を実行する際に，不要な突き合わせをより早く打ち切る工夫を導入する．具体的には，並列処理において，局所的に本来与えられた閾値より小さな閾値を導出する．そして文字列同士の編集距離を計算する代わりに各々の接尾辞同士の編集距離を計算し，局所的な閾値で評価することで，類似文字列ペアか否かを判定する．これにより不要な突き合わせ処理をより早く打ち切ることが期待出来る．真の編集距離の値は接尾辞同士の編集距離と局所的な閾値から容易に特定することが出来る．本手法は与えられた閾値に対する全ての類似文字列ペアを過不足なく列挙可能である．

c） **トライベースの突き合わせアルゴリズム**

上記 2 つの絞り込みを踏まえた突き合わせ処理を，より高速に動作させるため，平均文字列長の小さいデータに対して高速性を発揮する関連研究 Trie-Join [11] を拡張したアルゴリズムを提案する．

実データを用いた性能評価

実データを用いた実験で，処理速度およびスケーラビリティの評価結果を示す．Trie-Join と比較した処理速度評価で，提案する 2 つの絞り込みの工夫が有効であることを示す．更に，提案手法が実験データにおいて，並列数 1 に対して並列数 2 で約 1.8 倍，並列数 4 で約 3.1 倍，高速化したことを示す．

以下，本稿の構成は次の通りである．2. で本稿で用いる用語や記述の定義，および Trie-Join の概要を述べる．3. で提案手法について概説する．4. で実験結果を示す．5. で本稿に関連する研究について言及する．6. でまとめと今後の課題を述べる．

2. 準　　備

ここでは，本稿で用いる用語や記述の定義，編集距離の一般的な性質，および Trie-Join [11] について述べる．

2.1 記　　法

文字列データ X を文字列の集合であるとする．有限かつユニークな文字の集合を Σ とすると，文字列 $x \in X$ は Σ の要素文字からなる有限なシークエンスである．集合濃度およびシークエンス長を $|\cdot|$ で表す．即ち，文字列データ X に含まれる文字列数は $|X|$ である．文字列 x の文字列長は $|x|$ である．$0 \leqq k \leqq |x|+1$ に対して，$x[k]$ は x における k 番目の文字を表す．$x[0]$ は空文字 (‘’)，$x[|x|+1]$ は終端文字 (‘$’) をそれぞれ示し，$|\Sigma|$ および $|x|$ にはカウントされないとする．例えば $x =$“abbccd” とすると，$|x| = 6$，$x[0] =$‘’，$x[1] =$‘a’，$x[7] =$‘$’ である．更に $|$“”$| = 0$，$|$“$”$| = 0$ である．x の k 番目の文字までの接頭辞を x_k^p と表す．x の k 番目の文字から始まる接尾辞を x_k^s と表す．例えば $x =$“abbccd” であるとき，$x_3^p =$“abb”，$x_4^s =$“ccd” である．タプル (組) は定義域をそれぞれ持つ属性の集合に対するインスタンスである．記号 $\langle\cdot\rangle$ で表す．例えば，文字列ペアは 2 つの文字列属性の集合に対するタプルであり，各々の属性値を x, y とすると，$\langle x, y\rangle$ と表記される．

2.2 編 集 距 離

編集距離とは，ある文字列を別の文字列に変形するのに必要なオペレーションの最小回数である．ここでオペレーションとは，文字の挿入，削除，および置換の 3 つの処理を意味する．以降，2 つの文字列 x, y の編集距離を $ed(x, y)$ と記述する．一般に，$ed(x, y) = ed(y, x)$ は常に成り立つ．編集距離 $ed(x, y)$ は次式で得られる接頭辞の編集距離 $ed(x_i^p, y_j^p)$ を，動的計画法

		0	1	2	3	4	5
		_	n	o	r	t	h
0	_	0	1	2	3	4	5
1	a	1	1	2	3	4	5
2	r	**2**	**2**	**2**	**2**	**3**	**4**
3	t	3	3	3	3	2	3
4	h	4	4	4	4	3	2

図 1　編集距離計算の例

によって順次計算していくことで求められる [8].

$$ed(x_i^p, y_j^p) =$$

$$\begin{cases} \max(|x_i^p|, |y_j^p|) & (|x_i^p| = 0 \vee |y_j^p| = 0) \\ \min \begin{pmatrix} ed(x_{i-1}^p, y_j^p) + 1, \\ ed(x_{i-1}^p, y_{j-1}^p) + \theta, \\ ed(x_i^p, y_{j-1}^p) + 1 \end{pmatrix} & (\text{otherwise}) \end{cases}$$

ここで，θ は $x[i] = y[j]$ のとき 0，それ以外のとき 1 であるとする．本稿では，$ed(x, y)$ を省略して $ed_{x,y}$ と表すこともある．図 1 は，ある文字列ペアの編集距離を計算したときに得られる接頭辞同士の編集距離を，行列表現したものである．(i, j) 要素の値が $ed(x_i^p, y_j^p)$ に対応している．

［定義 1］ **文字列類似結合**　文字列データ X, Y および編集距離の閾値 τ が与えられたとき，文字列類似結合とは，閾値条件 $ed_{x,y} \leqq \tau$ $(x \in X, y \in Y)$ を満足するような文字列ペアとその編集距離とのペア $\langle \langle x, y \rangle, ed_{x,y} \rangle$ を，全て列挙する処理である．

与えられた条件を満足するか否かを判定することを，**突き合わせ** (*matching*) と呼ぶ．閾値条件を満足する文字列ペアを**類似文字列ペア**と呼ぶ．$X = Y$ のとき，そのような結合を *self-join* と呼ぶ．

編集距離計算は高コストであることで知られているが，$i = \tau + 1$ かつ任意の j に対して $ed(x_i^p, y_j^p) > \tau$ ならば，計算するまでもなく $ed(x, y) > \tau$ は自明であるため，計算を途中で打ち切ってよい．例えば図 1 では，閾値が 1 のとき，3 行目以降は計算する必要がない．このような打ち切り手法は *prefix pruning* と呼ばれ，逐次処理の枠組みの中で数多くの filter-and-refine アプローチに利用されている [4], [11], [13].

2.3　トライと Trie-Join

トライ (*trie*) は探索木の一つである．例えば，図 2 は表 1 の文字列データから構築されるトライである．各ノードに付された番号は便宜上の識別番号であるとする (ノード生成順)．今，識別番号 k であるノードを nd_k と表す．ノードは文字 $\sigma \in \Sigma$ をラベルとして所持し，それぞれ挿入された文字列の接頭辞に対応している．根ノードのラベルは空文字 (‘’) である．例えば，nd_9 は表 1 における sid_2 および sid_3 の接頭辞 “ran” に対応する．本稿ではトライ上のノード nd と対応する接頭辞 x_i^p とは相互に置き換え可能とする．即ち，図 2 と表 1 において，nd_9 は接頭辞 “ran” であり，接頭辞 “ran” は nd_9 である．同様に，$ed(nd_{11}, nd_{18}) = 1$ は $ed(\text{“ranna”}, \text{“ronna”}) = 1$ と同義である．nd_6 や nd_{11} などのように，与えられた文字列データの要素文字列そのものに対応するノードを，終端ノードと呼ぶ．本稿では，トライの終端ノードは元文字列へのポインタを保持するものとする．

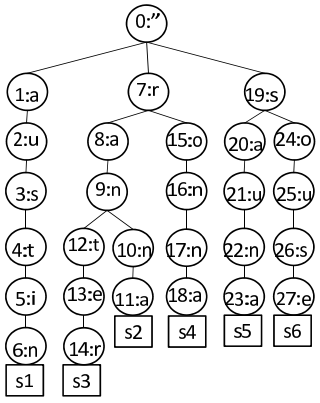

図 2　トライの例

表 1　文字列データの例

sid	**string**
sid_1	austin
sid_2	ranna
sid_3	ranter
sid_4	ronna
sid_5	sauna
sid_6	souse

今，本稿で提案するアルゴリズムの基本となる Trie-Join [11] について概説する．説明の簡略化のため，ここでは self-join の場合を考える．即ち，入力文字列データは 1 つである．Trie-Join では，入力文字列データ X から構築したトライを辿ることによって，ある 1 つの文字列 $x \in X$ に対して，各文字列 $y \in X$ との間で生じる動的計画法による接頭辞同士の編集距離計算を，幅優先で実行する．つまり，ある文字列 x に焦点を置き，x の接頭辞と各 y の可能な全ての接頭辞との編集距離を全て計算した後に，次に大きな x の接頭辞と，各 y の可能な全ての接頭辞との編集距離を全て計算する．これにより，共通の接頭辞 y_j^p を持つ文字列 y が複数存在するとき，$ed(x_i^p, y_j^p)$ の計算は一度しか発生しないため，効率がよい．具体的には次のような動作で実現する．まず，文字列データ X と閾値 τ とを入力とし，X からトライを構築する．次に，トライの根ノードから前置順に，各ノードのアクティブリストを生成していく．ノード nd のアクティブリストとは次の手順で生成されるノードの集合である．根ノード *root* のアクティブリストは，根ノード自身と，根ノードとの編集距離が閾値条件を満たす全ての子ノードから成る．根ノード以外の任意のノード nd のアクティブリスト A_n は，nd の親ノード *parent* が所持するアクティブリスト A_p から算出される．具体的には，まず各ノード $an \in A_p$ と，an の全ての子ノード ac に対し，nd との編集距離を算出する．そして閾値条件を満たすノード an (または ac) を A_n の要素とする．最後に Trie-Join は，トライの終端ノードと，そのアクティブリストに含まれる終端ノードとから可能な全ての文字列ペアを作り，類似文字列ペアとして出力する．ここで，より厳密には，ノード nd のアクティブリストの要素は，あるノード an と対応する編集距離 $ed_{nd,an}$ とのペア $\langle an, ed_{nd,an} \rangle$ であるとする．図 2 を例に取る．$\tau = 2$ とすると，根ノード nd_0 のアクティブリスト A_0 は，$\{ \langle nd_0, 0 \rangle, \langle nd_1, 1 \rangle, \langle nd_7, 1 \rangle, \langle nd_{19}, 1 \rangle \}$ である．nd_7 のアクティブリストは，親ノード nd_0 のアクティブリスト A_0 を用い，$\{ \langle nd_0, 1 \rangle, \langle nd_1, 1 \rangle, \langle nd_7, 0 \rangle, \langle nd_{19}, 1 \rangle, \langle nd_2, 2 \rangle, \langle nd_8, 1 \rangle, \langle nd_{15}, 1 \rangle, \langle nd_{20}, 2 \rangle, \langle nd_{24}, 2 \rangle \}$ となる．1 つのノードに対して生成されるアクティブリストのサイズ (即ち，リストに含まれる要素数) は $O(c_1^\tau)$ (c_1 は定数) となり，閾値 τ が大きいほど急激に増大することが分かる．一方，トライの全ノー

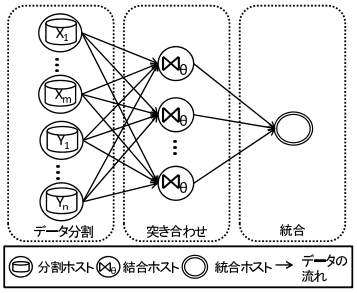

図 3　提案フレームワーク

ド数は $O(|X|)$ である．Trie-Join の時間計算量はアクティブリストの平均サイズとノード数の積に依存する．なお，入力とする文字列データが 2 つの場合でも，Trie-Join は問題なく動作する．この場合，2 つの入力文字列データを 1 つのトライに挿入し，同じデータに属するノード同士は，突き合わせ処理の対象としない．

3. 提 案 手 法

まず，提案するフレームワークの全体像を俯瞰する．図 3 が示す通り，提案フレームワークは，(a) データ分割フェーズ，(b) 突き合わせフェーズ，(c) 統合フェーズの 3 つのフェーズから成る．突き合わせフェーズが並列処理部分である．説明の便宜上，各フェーズを実行するホストをそれぞれ分割ホスト，結合ホスト，統合ホストと呼ぶ．実装上は，一つのマシンが複数種類のホストの役割を演じても構わない．なお，図 3 のように，入力文字列データ (文字列集合) X は，実際には複数の部分文字列集合 $X_1, ..., X_m$ に分かれ，複数の分割ホスト間で所持されていても良い．

本稿では文字列類似結合において多数発生する突き合わせを，データ分割と突き合わせのフェーズでそれぞれ絞り込むことで全体処理の高速化を図る．即ち，まず，等価結合におけるハッシュ結合アプローチに則り，入力文字列データを一度スキャンし，類似しそうもない文字列同士を異なるバケットに分配する．次に，結合ホストがバケットごとに文字列ペアの突き合わせを行う際，不要な突き合わせをより早く打ち切る．

以下，各々の絞り込みについて詳しく説明した後，それらを取り入れた並列処理フレームワークの具体的な動作を改めて述べる．

3.1 データ分割における絞り込み

等価結合と異なり，類似結合では結合キーに対応する文字列そのものをハッシュ関数に適用することは不可能である．そのため，本稿では prefix pruning を応用して入力データを分割する．前章で述べたように，prefix pruning はある文字列ペアに対して，接頭辞同士の比較を行い，明らかに類似しそうもない場合は，編集距離を最後まで計算することなく途中で打ち切るテクニックである．以下の原理に基づいている．

［定義 2］ **prefix pruning の原理**　閾値 τ に対し，2 つの文字列 x, y の各々の接頭辞 $x^p_{\tau+1}, y^p_{\tau+1}$ に共通文字が 1 つも存在しないならば，必ず $ed(x, y) > \tau$ である．

これは言い換えれば，文字列長が $\tau+1$ である接頭辞 $x^p_{\tau+1}, y^p_{\tau+1}$ に共通文字が 1 つも存在しなければ，文字列 x, y は閾値条件を満たす見込みがないため，突き合わせを行う必要がないことを意味する．

本研究では，この原理を並列処理におけるデータ分割に応用する．具体的には，入力文字列データ X と閾値 τ を与えられたとき，空文字と終端文字を除く任意の文字 $\sigma_i \in \Sigma$ をラベルに持つ $|\Sigma|$ 個のバケット $b_{\sigma_1}, ..., b_{\sigma_{|\Sigma|}}$ を用意し，接頭辞 $x^p_{\tau+1}$ が σ_i を有する文字列 $x \in X$ を，対応するバケット b_{σ_i} に振り分ける．1 つの文字列 $x \in X$ が，最大 $\tau+1$ 個のバケットに分配されうる．生成された各バケット b_{σ_i} は，所持ラベル σ_i を引数とする同一のハッシュ関数によって，N 個の分割データのいずれか一つに振り分けられ，並列に突き合わせ処理に掛けられる．即ち，本研究で生成される分割データは，上述のバケットを要素とするバケット集合である．接頭辞に共通文字が一つも存在しない文字列同士は同一のバケットに分配されることがないため，無駄な突き合わせ処理の発生を未然に防げる．

ここで，self-join を考えることにより，提案手法で発生し得る突き合わせ回数を簡単に観察する．ネステッドループを用いて任意の文字列ペアの突き合わせを行うブルートフォース手法の場合，バケットの大きさは元データ X に対して $O(\frac{\tau}{|\Sigma|}|X|)$ であることから，ある分割データに対して発生する突き合わせ回数は $O(\frac{|\Sigma|}{N}(\frac{\tau}{|\Sigma|}|X|)^2)$ である．並列数 N が大きくなるほど突き合わせ回数は小さくなると期待出来る．なお，本稿では平均文字列長の小さな文字列データを対象としており，閾値 τ もまた小さいと仮定している．後述するように，本稿では Trie-Join に基づくアルゴリズムを提案することで，ブルートフォース手法より高速な突き合わせ処理を行う．

3.2 突き合わせにおける絞り込み

prefix pruning の原理から，ラベル σ のバケットで発生する突き合わせでは，文字列ペア x, y は $1 \leqq i \leqq \tau+1 \wedge 1 \leqq j \leqq \tau+1$ において $x[i] = y[j] = \sigma$ であるために類似している可能性のみを考慮すれば良いことが分かる．今，編集距離 $ed(x, y)$ に対して 次のような局所的な上限値を導出する．

［定義 3］ **局所上限値**　文字列データ X, Y の任意の要素文字列 $x \in X, y \in Y$ と $1 \leqq i \leqq |x|+1, 1 \leqq j \leqq |y|+1$ に対して，次式で与えられる値 $u_{i,j}(x, y)$ を，編集距離 $ed(x, y)$ の**局所上限値**と呼ぶ．

$$u_{i,j}(x, y) = \max(i-1, j-1) + ed(x^s_i, y^s_j)$$

以降，$u_{i,j}(x, y)$ に対して，編集距離 $ed(x, y)$ を**真の編集距離**，$ed(x^s_i, y^s_j)$ を**接尾辞編集距離**と呼ぶ．上式が表す通り，局所上限値は，接尾辞編集距離と，対応する接頭辞 x^s_{i-1}, y^s_{j-1} の文字列長の大きいほうとの和である．文脈が明らかなとき，$u_{i,j}(x, y)$ を省略して u または $u(x, y)$ と表すことがある．局所上限値から，次のような性質が導ける．

［性質 1］ $1 \leqq i \leqq |x|+1, 1 \leqq j \leqq |y|+1$ に対して，常に $ed(x, y) \leqq u_{i,j}(x, y)$ が成り立つ．

［証明 1］ 基本的な性質 $ed(x, y) \leqq ed(x^p_{i-1}, y^p_{j-1}) + ed(x^s_i, y^s_j)$ および $ed(x, y) \leqq \max(|x|, |y|)$ から自明である．　□

［性質 2］ 任意の文字列 $x \in X, y \in Y$ と閾値 τ とが与えられたとき，$1 \leqq i \leqq \tau+1 \wedge 1 \leqq j \leqq \tau+1 \wedge x[i] = y[j]$ を満たす

図 4　バケットの例

b_σ	x^s_{i+1}	i	ptr_x
$b_{\text{`a'}}$	austin	0	sid_1
	anna	1	sid_2
	anter	1	sid_3
	auna	1	sid_5

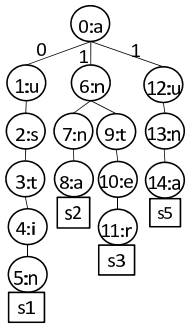

図 5　重みつきトライの例

i, j に対して，以下の条件が成り立つとき，$\langle x, y\rangle$ は類似文字列ペアである．

$$ed(x^s_i, y^s_j) \leqq \tau - \max(i-1, j-1) \quad (1)$$

［証明 2］ 性質 1 から自明である．　□

以降，$\tau - \max(i-1, j-1)$ を**局所閾値**と呼ぶ．

［性質 3］ 任意の文字列ペア $\langle x, y\rangle$ が類似文字列ペアであるとき，集合 $U^\tau_{x,y} = \{u_{i,j}(x,y) | 1 \leqq i \leqq \tau+1 \wedge 1 \leqq j \leqq \tau+1 \wedge x[i] = y[j]\}$ は空でなく，かつ，必ず $\min_{u \in U^\tau_{x,y}}(u) = ed(x,y)$ である．

［証明 3］ prefix pruning の原理より，$ed(x,y) \leqq \tau$ のとき必ず $x[i] = y[j] \wedge 1 \leqq i \leqq \tau+1 \wedge 1 \leqq j \leqq \tau+1$ を満たす i, j が 1 つ以上存在する．編集距離 $ed(x,y)$ を地点 $(0,0)$ から地点 $(|x|, |y|)$ への最短経路コストと見なせば，最短経路はいずれかの部分経路 $(i-1, j-1) \to (i, j)$ を必ず通る．一般性を失うことなく，$x[k] = y[h]$ である異なる地点 (k, h) $(k \neq 0,$ $h \neq 0)$ が地点 $(i-1, j-1)$ までのあらゆる経路上に存在しない場合のみを考えることが出来る．このとき，部分経路 $(i-1, j-1) \to (i, j)$ を通り，地点 $(|x|, |y|)$ までいく最小コストは $\max(i-1, j-1) + ed(x^s_i, y^s_j) = u_{i,j}(x,y)$ となる．最も小さい $u_{i,j}(x,y)$ が最短経路コストであり，これは即ち $ed(x,y)$ である．　□

以上の性質から，バケットで発生する突き合わせで，真の編集距離を計算する代わりに接頭辞編集距離を計算し，条件式 1 を評価することで，類似文字列ペアを過不足なく発見出来ることが分かる．これは従来手法と比べて次の 2 つの点で有効であると考えられる．第 1 に，接尾辞編集距離の計算コストは真の編集距離のそれよりも小さい．第 2 に，元の閾値 τ よりも小さい局所閾値を用いることで，不要な接尾辞編集距離の計算をより早く打ち切ることが出来る．

なお，性質 3 から，類似文字列ペアの真の編集距離は局所上限値から容易に特定出来る．

3.3　並列処理フレームワーク

ここでは，前節で述べた 2 つの絞り込みを取り入れた並列処理フレームワークについて，各フェーズごとに詳述する．

3.3.1　データ分割フェーズ

データの分割および分配がどのように行われるかを述べる．今，分割ホストが所持する入力文字列データ X から生成される N 個の分割データ $dX_1, ..., dX_N$ を，次のように定義する．

［定義 4］ **SIP タプル**　文字列 $x \in X$ と整数 $1 \leqq i \leqq \tau+1$ に対して，文字列属性値 (接尾辞) x^s_i，整数属性値 (接頭辞長) $i-1$，および x へのポインタ属性値 ptr_x の三つ組 $\langle x^s_i, i-1, ptr_x\rangle$ を，x の i 番目の **SIP タプル**，または単に SIP タプルと呼ぶ．

［定義 5］ **バケット**　文字列データ X に対して，ある文字 $\sigma \in \Sigma$ をラベルとして所持するバケット b_σ は，SIP タプルの集合 $b_\sigma = \{\langle x^s_i, i-1, ptr_x\rangle | x \in X \wedge 1 \leqq i \leqq \tau+1 \wedge x[i] = \sigma\}$ である．

［定義 6］ **分割データ**　文字列データ X とハッシュ関数 $h(\sigma)$ とから生成される X のある分割データ dX_n は，バケットの集合 $dX_n = \{b_\sigma | h(\sigma) = n\}$ である．ただし，$1 \leqq n \leqq N$ である．

表 4 は，表 1 から閾値 $\tau = 2$ のときに生成されるラベル ‘a’ のバケットを示している．文字列属性 x^s_{i+1} の最初の文字が ‘a’ である SIP タプルが，バケット $b_{\text{`a'}}$ に含まれている．余白の都合上省略するが，この例の場合，$b_{\text{`a'}}$ の他に $b_{\text{`n'}}$，$b_{\text{`o'}}$，$b_{\text{`r'}}$，$b_{\text{`s'}}$ および $b_{\text{`r'}}$ の，計 6 つのバケットが生成される．$N = 6$ とすれば，これらのバケットの内の 1 つを要素として持つような 6 つの分割データが，6 つの結合ホストへそれぞれ分配される．1 つのバケットに同じポインタ (ptr_x) を所持する SIP タプルが複数存在することもある．

単純化のため，文字列データ X に出現する文字の頻度が一様であると仮定すれば，生成される 1 つの分割データの大きさは $O(\frac{\tau}{N}|X|)$ である．並列数 N が大きくなるほど，個々の結合ホストでかかる処理コストは小さくなることが期待出来る．

3.3.2　突き合わせフェーズ

突き合わせフェーズでは，データ分割フェーズで生成された分割データを各々の結合ホストが受け取って，バケットごとに SIP タプルのペアの突き合わせを行う．考えられる最も単純な計算手法 (ブルートフォース手法) は，ネステッドループを用い，バケット内のあらゆる SIP タプルペアを条件式 1 で評価していくことである．一方で，本稿と同じく平均文字列長の比較的短いデータを対象とする Trie-Join は，トライを利用することでブルートフォース手法より高速な突き合わせ処理を行うことが出来る．そこで本稿では，Trie-Join を拡張し，条件式 1 による突き合わせを効率よく行うアルゴリズムを，新たに提案する．以降，説明の簡略化のため，self-join の場合を考える．

提案アルゴリズムは，分割データ dX と閾値 τ とを入力として受け取ると，各バケット $b_\sigma \in dX$ ごとに，そこに属する SIP タプルをパトリシアトライに似たトライ構造で表現する．本稿が提案するトライ構造をここでは**重みつきトライ**と呼ぶ．重みつきトライの例を図 5 に示す．これは表 4 におけるバケット $b_{\text{`a'}}$ に含まれる SIP タプルを表現した木構造である．図 2 に示された従来のトライと大きく異なる点は，根ノードから子ノードへ伸びるエッジに**重み**が存在していることである．重みつきトライでは，各ノードが SIP タプルの文字列属性値 (接尾辞) の各接頭辞に対応する．また，重みは SIP タプルの整数属性値に対応する．根ノードだけは，重みに応じて複数の文字列へ対応し得る．例えば図 5 では，重みが 0，1 の 2 種類存在することから，根ノードは文字列長が 0 である接頭辞 “” を持つ文字列 “a” と，文字列長が 1 である接頭辞 “*” を有する文字列 “*a” の 2 つの文字列に対応している．ここで，‘*’ はなんらかの文字を表す特殊文字とする．以降，根ノード $root$ が重み i によっ

表 2　重みつきトライ構築アルゴリズム

入力: バケット b_σ
出力: 重みつきトライ T_σ

1. 根 pnd のラベルを σ とし，T_σ を初期化;
2. **foreach** $\langle x^s, i, ptr_x \rangle \in b_\sigma$
3. 　重み i とラベル $x^s[2]$ を持つ根の子ノード nd を探索;
4. 　**if** nd が存在しない **then**
5. 　　重み i とラベル $x^s[2]$ を持つ根の子ノード nd を生成;
6. 　$pnd = nd$;
7. 　**foreach** $k = 3, ..., |x^s|$
8. 　　**if** pnd にラベル $x^s[k]$ を持つ子ノード nd が存在しない
9. 　　**then** ラベル $x^s[k]$ を持つ nd を生成し，pnd の子とする;
10. 　　$pnd = nd$;
11. 　終端ノード pnd に ptr_x を登録;

表 3　重みつきトライ探索アルゴリズム

入力: 重みつきトライ T_σ，閾値 τ
出力: 局所上限値 $u(x,y) \leqq \tau$ を満足するポインタペアと $u(x,y)$ とのペア集合 $dS = \{\langle \langle ptr_x, ptr_y \rangle, u(x,y) \rangle\}$

1. **foreach** 根の重み i のついた子ノード nd:i
2. 　$root$ のアクティブリスト $A_{root:i}$ を初期化;
3. 　**call** make_root_active_list(i) ;
4. 　**call** make_active_list($root$:i, nd:i) ;
5. **foreach** T_σ の終端ノード nd:i
6. 　**foreach** $\langle an$:$j, ed \rangle \in A_{nd:i}$
7. 　　**if** an:j が終端ノード
8. 　　**then** nd:i と an:j がそれぞれ保持するポインタのあらゆるペア $\langle ptr_{nd:i}, ptr_{an:j} \rangle$ を生成し，$\langle \langle ptr_{nd:i}, ptr_{an:j} \rangle, \max(i,j) + ed \rangle$ を dS へ追加;

Function make_root_active_list(重み i)

9. **foreach** 根の重みつき子ノードの重み i'
10. 　$A_{root:i}$ に $\langle root$:$i', 0 \rangle$ を追加;
11. 　**foreach** 重み i' のついた重みつき子ノード cn:i'
12. 　　**if** $1 \leqq \tau - \max(i, i')$ **then** $A_{root:i}$ に $\langle cn$:$i', 1 \rangle$ を追加;

て区別されるべきとき，$root$:i と記述することがある．根ノード $root$:i の子孫ノード nd を nd:i と記述することがある．例えば図 5 では，根ノード nd_0:1 の子孫ノード nd_{11} は nd_{11}:1 と記述される．重みつきトライを構築するアルゴリズムを表 2 に示す．表 2 によれば，各 SIP タプル $\langle x^s, i, ptr_x \rangle$ に対して，最初に 5 行目で重みと共に根の子ノードを登録し，以降は従来のトライ構築のプロセスと同様に，ノードを生成しながら文字列属性 x^s をトライに挿入していく．

提案アルゴリズムは，重みつきトライを根ノードから前置順に探索しながら，訪問した各ノード nd_1:i に関して，必要な他ノード nd_2:j との編集距離 $ed(nd_1$:i, nd_2:$j)$ (即ち，接尾辞編集距離) を計算し，条件式 1 で評価する．表 3 は重みつきトライ探索アルゴリズムを示している．表 3 によれば，提案アルゴリズムはまず，ある重み i で根ノード $root$ を区別し，$root$:i のアクティブリスト $A_{root:i}$ を make_root_active_list 関数を呼び出すことで生成する (3 行目)．次に 4 行目において，make_active_list 関数を再帰的に呼び出すことで前置順に全てのノードを訪問し，親ノードのアクティブリストから自分のアクティブリストを生成していく．ノード nd:i のアクティブリストの要素は，局所閾値 $\tau - \max(i,j)$ を満足するノード an:j と接尾辞編集距離 $ed(nd$:i, an:$j)$ とのペアである．再帰処理を終えると，重みつきトライの終端ノード nd:i と，そのアクティブリスト $A_{nd:i}$ に含まれる全ての終端ノード an:j とを用い，各々が所持する SIP タプルのポインタ属性値で可能な全てのペア $\langle ptr_{nd}, ptr_{an} \rangle$ を作り，そのペアと対応する局所上限値とのペア $\langle \langle ptr_{nd}, ptr_{an} \rangle, max(i,j) + ed \rangle$ を，集合 dS へ追加していく (8 行目)．最後に，重みつきトライ T_σ から得られる局所的な類似結合結果 (**局所結合結果**) として dS を出力する．dS_σ は統合ホストへ送られる．

make_root_active_list 関数は，引数として重み i を与えられる．まず，重み i で区別された根ノード $root$:i と，任意の重み i' で区別された根ノード $root$:i' との接尾辞編集距離を計算する．このとき必ず 0 であるため，局所閾値との比較を行うことなく $root$:i のアクティブリスト $A_{root:i}$ へノード $root$:i' と 0 を追加する (10 行目)．次に make_root_active_list 関数は，$root$:i' とその子ノード cn:i' との接尾辞編集距離を得る．編集距離の意味を考えれば，$ed(root$:i', cn:$i') = 1$ は自明である．12 行目で条件式 1 を評価し，真ならば cn:i' と 1 を $A_{root:i}$ へ追加する．make_active_list 関数のアルゴリズムの記載は余白の都合上省略する．

Trie-Join と同様，提案アルゴリズムの時間計算量はアクティブリストの平均サイズとトライのノード数 (即ち，アクティブリストの生成回数) との積に依存する．本稿では突き合わせ処理を条件式 1 で評価することから，Trie-Join では $O(c_1^\tau)$ であるアクティブリストの平均サイズが，$O(E_{\max}(c_2^{\tau - \max(i,j)}))$ となる．ここで，c_2 は c_1 とは異なる定数，$E_{\max}(\cdot)$ は $\max(i,j)$ に関する期待値である．ただし，c_1 と c_2 との間に大きな差はないものとする．一方，ある結合ホストでバケットごとに構築される重みつきトライのノード数の総和は $O(\frac{\tau}{N}|X|)$ となるため，提案手法は並列数 N が大きくなるほど高速化が見込まれる．ところで，厳密には，c_1 および c_2 は $|\Sigma|$ に依存する値である．本稿の実験では $|\Sigma|$ が与える影響への評価は今後の課題とし，最も処理時間に影響を与える τ に着目した評価のみを行っている．

3.3.3　統合フェーズ

統合ホストは，N 個の結合ホストから局所結合結果 dS を受け取ると，各要素 $\langle \langle ptr_x, ptr_y \rangle, u_i \rangle \in dS$ を大域的な結合結果 S へ追加する．このとき，同じポインタペアを有する要素 $\langle \langle ptr_x, ptr_y \rangle, u_j \rangle$ が既に S に存在していた場合，$u_i < u_j$ であれば $\langle \langle ptr_x, ptr_y \rangle, u_j \rangle$ を S から削除して代わりに $\langle \langle ptr_x, ptr_y \rangle, u_i \rangle$ を追加する．最終的に得られた S が，閾値 τ を与えたときの文字列類似結合の正しい出力結果となる．

なお，以上は self-join における議論であるが，入力とする文字列データが 2 つの場合でも，提案手法は問題なく動作する．

4. 実　　験

本稿では，提案手法の処理速度とスケーラビリティを評価す

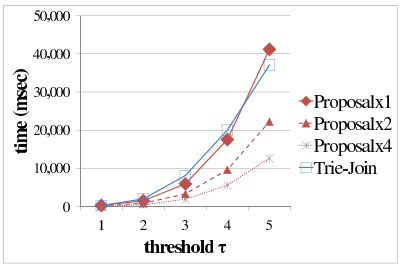

図 6　処 理 時 間

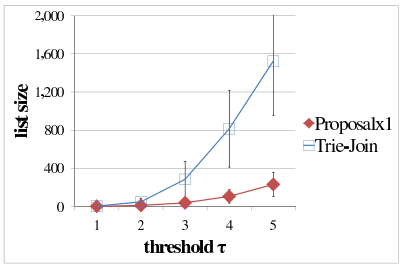

図 7　アクティブリストの平均サイズ

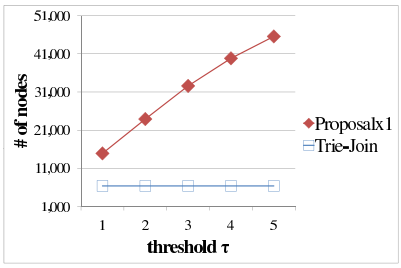

図 8　木のノード数

る．実験環境は，Fedora 5，PentiumD 3.4GHz　CPU，2GB RAM である．実装は C++で行い，-O3 オプションを与え，g++ 4.1.1 によりコンパイルした．与えたパラメタは，それぞれ $1 \leqq \tau \leqq 5$，並列数 $N \in \{1, 2, 4\}$ である．実験データとして，実データである英語の辞書データ Dict を用いた．Dict の統計量は，|Dict|=1K，|Σ|=26．更に，平均文字列長，最大文字列長および最小文字列長が，それぞれ 8.7，14 および 2 である．

4.1　処 理 速 度

ここでは各々の並列数 N で得られる処理速度と，提案した 2 つの絞り込みの工夫が処理速度に与える影響とを確認する．逐次処理である Trie-Join を比較対象とした．以降，N に対する提案手法を，それぞれ Proposalx1，Proposalx2，Proposalx4 と記す．本稿では，並列処理に掛かる処理時間を擬似的に得た．即ち，データ分割フェーズにおいて，Dict から N 個の分割データを生成した後，同一のマシンで逐次的に，各々の分割データを入力とする提案アルゴリゴリズムを実施していき，最も処理時間の大きいものを，Proposalx$N$ における処理時間とした．通信時間は今回の実験では加味しない．Trie-Join に対しては，同一のマシンに Dict そのものを入力として与え，処理時間を計測した．

図 6 は閾値 τ に対する処理時間を示している．どのプロット線も，τ の増加に伴い処理時間が増大しているが，提案手法では並列数 N が大きくなるほど τ の増加に伴う速度低下の傾向が緩やかになっていく様子が確認出来た．今，並列数 N における高速化率を，Proposalx$N$ の閾値 τ における処理時間 $time_N^\tau$ を用いて $\frac{time_1^\tau}{time_N^\tau}$ の平均値で得ると，Proposalx1 に対して，Proposalx2 は約 1.8 倍，Proposalx4 は約 3.1 倍の高速化が認められた．入力データの大きさは，ProposalxN が $O(\frac{\tau}{N}|Dict|)$，Trie-Join が $O(|Dict|)$ であり，N が小さく τ が大きいほど提案手法が処理するデータは Trie-Join に比べて大きくなっていくが，提案手法は $N = 1$ においても $\tau \leqq 5$ の範囲で Trie-Join と同等の速度性能を示しており，提案した 2 つの絞り込みの効果が認められた．なお，$\tau > 5$ において，Proposalx1 の処理速度は Trie-Join と比較して急速に増大した．

2 つの絞り込みの効果をより詳しく分析するため，提案手法の処理速度に影響を与える因子である，アクティブリストの平均サイズとトライのノード数とを測定した．図 7 は，Dict における Proposalx1 と Trie-Join のアクティブリストのサイズの平均サイズを示している．エラーバーは標準偏差を表している．提案手法で生成されるアクティブリストの平均サイズは並列数 N に依存しないため，$N > 1$ に対する Proposal の実験結果は，ここではプロットしない．Trie-Join と比べて，提案アルゴリズムは τ の増加に伴うアクティブリストの成長を大幅に抑制していることが分かる．図 8 は閾値 τ に対する，Trie-Join で構築される 1 つのトライのノード数と，提案手法で分割データに含まれるバケットから構築される全てのトライのノード数の総和とを表している．Trie-Join のノード数は τ に対して一定である一方，提案手法では比例的に増加する傾向が確認出来た．理論上，並列数 N が大きくなるほど，提案手法におけるノード数は小さく抑えることが出来る．

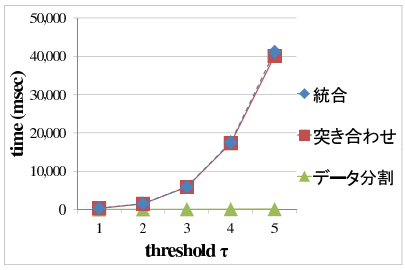

図 9　Proposalx1 における処理時間の内訳

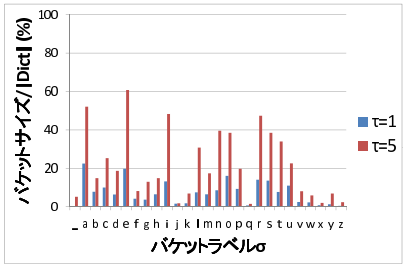

図 10　|Dict| に対するバケットサイズの比

4.2　スケーラビリティ

全体時間の中で突き合わせに掛かる時間が占める割合がどの程度であるかを検証することで，スケーラビリティの簡単な評価を行う．本来，全体時間には処理時間の他に通信時間も含まれるが，今回は通信時間を考慮に入れないものとする．理論的には，元データ X に対して，分割ホストから結合ホストへ流れる総通信量は $\Omega(\tau|X|)$，類似文字列ペアの完全集合を S とすると，結合ホストから統合ホストへ流れる総通信量は $\Omega(\tau|S|)$ となるため，通信時間はこれらに準じた大きさとなることが予測出来る．

図 9 は，図 6 における Proposalx1 の実行時間を，データ分割フェーズ (緑)，突き合わせフェーズ (赤)，統合フェーズ (青) ごとに内訳したものである．データ分割フェーズの処理時間には 1 回のデータスキャンに掛かった時間も含まれている．突き

合わせフェーズに掛かる時間が全体に対して大きな割合を占めており，このフェーズを並列化することの有効性が確認出来る．

ところで，selft-join の場合，個々のバケット b_σ のサイズ (要素数) は単純には元データ X に対して $O(\frac{\tau}{|\Sigma|}|X|)$ であるが，実際には $\sigma \in \Sigma$ の出現頻度に依存する．図 10 は，Dict データを閾値 τ で分割した際の，|Dict| に対するバケットサイズの比を表している．τ は 1 および 5 を与えた．どちらの閾値の場合でも，バケットの大きさに偏りが見られ，突き合わせ処理に掛かる時間が，結合ホストの間でばらつくことが分かる．

より大きな実験データを用いた検証や，バケットサイズの均一化によるスケーラビリティの向上は，今後の課題とする．

5. 関 連 研 究

文字列類似結合手法として，数多くの filter-and-refine アプローチが提案されている [1], [4], [6], [13]．例えば L. Gravano らは，類似する文字列同士が多くの q-gram を共通して含むことに着目し，文字列ペア間の共起 q-gram 数の下限を導出し，それを用いた filtering 手法を提案した [6]．これに対して，C. Xiao らは，文字列ペア間で共起しない q-gram の数から，距離尺度の下限を導き出した [13]．これらの研究が q-gram をシグネチャとして用いている一方，元の文字列から k 個の文字を削除することで作成される variants をシグネチャとする研究もある [3], [12]．彼らは固有表現抽出の問題として文字列類似結合を扱っている．一方，S. Ji らはキーワード検索 [7] に関する研究で，トライを用いた編集距離計算手法を提案した．J. Wang らは平均文字列長が高々30 程度の文字列データを対象とした Trie-Join アルゴリズムを提案した [11].

その他，文字列に対する類似問合せ処理として，閾値による検索や，top-k による検索および結合に関する研究も数多くなされている [2], [14]

等価結合においては，並列処理の研究は 80 年代から盛んに研究されてきた．例えば M. Kitsuregawa らの並列 GRACE ハッシュ結合などが有名である [9], [15]．等価以外の条件を指定する θ 結合に関しては，数値属性を結合キーとする結合処理のためのデータ分割手法などが 90 年代から研究されてきた [5]．しかしながら，文字列に対する類似結合の並列処理に関する研究は今なお少ない．2010 年，R. Vernica らは Jaccard 係数やコサイン係数などのトークンに着目した距離尺度に対する類似結合を MapReduce の枠組みの中で議論した [10].

6. お わ り に

本稿では，編集距離制約下で，平均文字列長が比較的短い文字列データを対象に，より高速な文字列類似結合を行うための並列処理フレームワークを提案した．提案手法は，不要な突き合わせが発生しないように入力データを分割し，並列処理において，不要な突き合わせをより早く打ち切る 2 つの工夫を導入している．また，より高速な突き合わせ処理を行うために，関連研究である Trie-Join [11] を拡張した新しいアルゴリズムを提案した．実データを用いた実験では，提案手法の処理速度とスケーラビリティの評価を行った．Trie-Join と比較した処理速度評価では，フレームワークに取り入れた 2 つの絞り込みの工夫が有効であることを確認した．更に実験データにおいて，提案手法が並列数 1 に対して並列数 2 で約 1.8 倍，並列数 4 で約 3.1 倍の高速性を得られたことを示した．今後の課題として，並列数 $N > |\Sigma|$ での文字列類似結合の並列処理手法や，データサイズが結合ホスト間で均等になるようなデータ分割手法の検討，および多様な実データを用いたより詳細な実験などが挙げられる．

謝辞 本研究の一部は，経産省エネルギーイノベーションプログラムの一環として独立行政法人新エネルギー・産業技術総合開発機構 (ＮＥＤＯ) から委託を受け実施している「グリーン IT プロジェクト」の成果である．

文　　献

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[3] Thomas Bocek, Ela Hunt, and Burkhard Stiller. Fast similarity search in large dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007.

[4] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[5] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, pages 443–452, 1991.

[6] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[7] Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[8] Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. In *CPM*, pages 60–68, 2000.

[9] Masaru Kitsuregawa, Shin ichiro Tsudaka, and Miyuki Nakano. Parallel grace hash join on shared-everything multiprocessor: Implementation and performance evaluation on symmetry s81. In Forouzan Golshani, editor, *Proceedings of the Eighth International Conference on Data Engineering, February 3-7, 1992, Tempe, Arizona*, pages 256–264. IEEE Computer Society, 1992.

[10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD Conference*, pages 495–506, 2010.

[11] Jiannan Wang, Guoliang Li, and Jianhua Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

[12] Wei Wang, Chuan Xiao, Xuemin Lin, and Chengqi Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, pages 759–770, 2009.

[13] Chuan Xiao, Wei Wang 0011, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[14] Chuan Xiao, Wei Wang 0011, Xuemin Lin, and Haichuan Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[15] 喜連川優, 津高新一郎, and 中野美由紀. 共有メモリ型マルチプロセッサによる並列ハッシュ結合演算処理とその評価. **情報処理学会論文誌**, 34(5):1019–1030, 5 月 1993.