# メニーコアプロセッサを用いた大規模な集合間類似結合の高速化

菅野　健太†　　天笠　俊之††　　北川　博之††

† 筑波大学大学院システム情報工学研究科　〒 305–8573 茨城県つくば市天王台 1–1–1
†† 筑波大学計算科学研究センター　〒 305–8573 茨城県つくば市天王台 1–1–1
E-mail: †sugano@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

**あらまし**　集合間類似結合とは，集合をレコードとする二つのレコード集合から閾値以上の類似度を示すレコードのペアを抽出・列挙する処理である．しかし，多くの類似度計算が必要となるため，大規模なレコード集合に対する処理は困難である．そこで，本研究では Intel Xeon Phi を用いた集合間類似結合の高速化手法を提案する．本手法では，計算コストの削減のために MinHash 法による LSH(Locality Sensitive Hashing) を採用し，これを用いた類似結合処理を Intel Xeon Phi に最適化することで高速化を図る．さらに，テキストデータを用いた実験により，提案手法の性能を評価する．

**キーワード**　Intel Xeon Phi，集合間類似結合，LSH，並列計算

## 1. はじめに

集合間類似結合とは，集合をレコードとする二つのレコード集合から閾値以上の類似度を示す集合の対を抽出・列挙する処理である．その応用例としては，文書の実体解決やデータクリーニングなどがある．また，この処理の技術的課題の一つとして，大規模なレコード集合に対する処理の計算コストが大きいことが挙げられる．この問題に対処するために，多くの高速化手法が提案されている．

一方近年，メニーコアプロセッサが高い処理性能を持つ並列計算機として HPC (High Performance Computing) 分野で注目を集めている．Intel 社製のメニーコアプロセッサの Xeon Phi は多くの物理コアと 512-bit の SIMD 演算器を搭載しており，Oakforest-PACS や Tianhe-2 などのスーパーコンピュータの計算ノードとして採用されている．

そこで，本研究では，Intel Xeon Phi を用いた大規模な集合間類似結合の高速化手法を提案する．本手法では，計算コストの削減のために，Li らが提案した b-bit MinHash [8] による LSH (Locality Sensitive Hashing) を用いる．そして，この処理を Intel Xeon Phi によって並列化することで高速化を図った．

本手法は，圧縮処理と結合処理の二つの処理から構成される．まず，圧縮処理では，レコード集合中の各レコードをシグネチャに変換し，シグネチャ集合を構築する．そして，結合処理では，与えられた二つのシグネチャ集合に対して，LSH を用いた類似結合処理を行う．

また，提案手法の性能を評価するために実験を行った．テキストデータを用いた実験により，LSH を用いない場合と比較して，最大で約 39 倍の高速化を達成したことを示した．それに加えて，CPU 上での処理と比較して最大で約 2.4 倍の高速化を達成したことを示した．

まず，2. 章で本研究に関する前提知識について説明する．次に，3. 章で提案手法について説明し，4. 章で評価実験について述べ，5. 章で関連研究について説明する．最後に，6. 章で結論を述べる．

## 2. 前提知識

本節では，本研究の前提知識について説明する．2.1 節で Intel Xeon Phi について説明し，2.2 節で集合間類似結合の説明をする．また，2.3 節で MinHash について説明し，2.4 節で b-bit MinHash について説明する．そして，2.5 節と 2.6 節で b-bit MinHash に基づいた集合間類似結合について説明する．

### 2.1 Intel Xeon Phi

Xeon Phi は Intel 社製のメニーコアプロセッサであり，高い並列処理性能を持つことから，様々なアプリケーションの高速化に貢献している [6, 10]．現在，第二世代の KNL (Knights Landing) が販売されており，この特徴として，最大で 72 個の物理コアによる並列処理が可能であることや，512bit 幅の SIMD (Single Instruction Multiple Data) 命令をサポートすることなどが挙げられる．これらを活用することによって非常に高いパフォーマンスを発揮し，その倍精度ピーク性能は約 3TFLOPS に及ぶ．なお，SIMD 命令とは，一つの命令で複数個のデータに対して同時に演算を適用する処理方式を指す．また，KNL は高バンド幅メモリの MCDRAM (Multi-Channel DRAM) を合計 16GB 搭載する．

### 2.2 集合間類似結合

集合間類似結合は，集合からなるレコード集合 $R, S$，集合間類似度関数 Sim()，閾値 $t$ に対して，

$$R \bowtie_t S \triangleq \{\langle r, s\rangle \in R \times S \mid \mathrm{Sim}(r, s) > t\} \quad (1)$$

を求める処理である．集合間類似度関数としては，一般に Jaccard 係数や Dice 係数，Cosine 類似度などの類似尺度が用いられるが，本研究では，集合間類似度関数として Jaccard 係数を用いる．Jaccard 係数を用いることの利点の一つに，後述する MinHash 法による Jaccard 係数の推定が可能な点が挙げられる．なお，集合 $r, s$ 間の Jaccard 係数は，$\mathrm{Jac}(r, s) = \frac{|r \cap s|}{|r \cup s|}$ として定義される．

Jaccard 係数の計算の技術的課題として，二つの集合の積集合の計算の際に多くの要素ペアの比較が必要となるため，計算コストが大きいという点が挙げられる．それに加えて，集合全体をメモリに格納する必要があるため，Jaccard 係数の計算対象となる集合ペアが多い場合，非常に大きなメモリ領域が必要となるという点が挙げられる．

### 2.3 MinHash

上記の Jaccard 係数の計算の技術的課題に対処するために，Broder らは MinHash [1] を提案した．この手法では，集合をシグネチャと呼ばれるコンパクトなデータ構造に変換し，二つのシグネチャを比較することで Jaccard 係数の推定値を計算する．なお，集合 $r \subseteqq \Omega = \{0, 1, ..., D-1\}$ のシグネチャは，$k$ 個のハッシュ関数 $\pi_1, ..., \pi_k : \Omega \mapsto \Omega$ に対して，

$$\mathrm{Sig}(r) = \left\langle \min_{e \in r} (\pi_1(e)), ..., \min_{e \in r} (\pi_k(e)) \right\rangle \tag{2}$$

として定義される．また，二つの集合 $r, s$ の Jaccard 係数の推定値は，$\hat{\mathrm{Jac}}(r,s) = 1 - \mathrm{Ham}(Sig(r), Sig(s))/k$ で得られる．なお，$\mathrm{Ham}(\cdot,\cdot)$ は二つのシグネチャのハミング距離を示す．

また，Jaccard 係数の推定精度と計算コストはトレードオフの関係にある．ハッシュ関数の数 $k$ の増加に伴って，推定精度は向上するものの，シグネチャのサイズは増加し，推定値計算の処理時間もまた増加する．

### 2.4 b-bit MinHash

上記のトレードオフを改善するために，Li らは b-bit MinHash [8] を提案した．この手法では，各ハッシュ関数について最小ハッシュ値全体を保存するのではなく，それらの下位 $b$-bit のみを保存してシグネチャを構築する．これによって，異なる最小ハッシュ値の衝突が発生するが，ハッシュ関数の数 $k$ を十分に増やすことで推定精度の維持が可能であり，元の MinHash と比較して空間効率が優れている．また，集合 $r, s$ の Jaccard 系数の推定値は，

$$\hat{\mathrm{Jac}}(r,s) = 1 - \frac{\mathrm{Ham}\left(\mathrm{Sig}\left(r\right), \mathrm{Sig}\left(s\right)\right)}{\left(1 - 2^{-b}\right) k} \tag{3}$$

で得られる．

また，b-bit MinHash の重要な特徴の一つとして，$b = 1$ の場合，SIMD 命令によってシグネチャの生成や Jaccard 係数の推定値計算を効率的に処理可能であるという点が挙げられる．

### 2.5 b-bit MinHash に基づく集合間類似結合

b-bit MinHash に基づく素朴な集合間類似結合は，圧縮処理と結合処理から構成される．まず，圧縮処理では，与えられたレコード集合 $R, S$ に対して，各レコードのシグネチャを計算してシグネチャ集合 $R', S'$ を構築する．次に，結合処理では，二つのシグネチャ集合 $R', S'$ に対して，全てのシグネチャペアについてハミング距離を計算し，その値が閾値 $t_h = \lceil k(1 - 2^{-b})(1-t) \rceil$ を下回るペアに対応するレコードペアを出力する．なお，$t_h$ はシグネチャのハミング距離に対する閾値であり，式 3 を変形することで得られる．

ここで，ハミング距離の計算対象となるシグネチャペアの数は $|R| \times |S|$ であるため，大規模なレコード集合に対する処理は困難である．

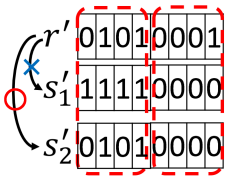


図 1　フィルタリングの例（$b = 1, k = 8, t = 0.5$）

### 2.6 フィルタリングを用いた b-bit MinHash に基づく集合間類似結合

b-bit MinHash に基づく集合間類似結合における結合処理において，フィルタリングにより，明らかに類似していないシグネチャペアをハミング距離の計算対象から除外することで，計算コストの削減が可能である．

b-bit MinHash に基づく集合間類似結合の結合処理は，ハミング距離が閾値 $t_h$ を下回るシグネチャのペアを抽出・列挙する処理である．したがって，シグネチャを互いに素な $t_h$ 個の部分ビット列に分割したとき，全ての対応する部分ビット列のペアが異なるシグネチャペアは，ハミング距離の下限が $t_h$ となるため，ハミング距離の計算対象から除外できる．

図 1 はフィルタリングの様子を示している．$\lceil 8(1 - 2^{-1})(1 - 0.5) \rceil = 2$ より，各シグネチャは 2 個の部分ビット列に分割される．$r'$ と $s'_2$ は一つ目の部分ビット列が等しいためハミング距離の計算対象となるが，$r'$ と $s'_1$ は全ての部分ビット列が異なるため推定値計算対象から除外される．

なお，このフィルタリングは MinHash に基づく LSH (Locality Sensitive Hashing) [5,13] の一例である．LSH とは，類似するレコードを高確率で同一のバケットにハッシュする手法であり，そのハッシュ関数は複数個のハッシュ関数を組み合わせて構成される．

## 3. 提 案 手 法

提案手法は，2.6 節で説明したフィルタリングを用いた b-bit MinHash に基づく集合間類似結合に対して，Intel Xeon Phi によって並列化することで高速化を図る．本手法は，2.5 節及び 2.6 節と同様に，圧縮処理と結合処理の二つの処理から構成される．

### 3.1 レコード集合の表現形式

レコード集合は以下の二つの配列で表されるものとする．

- *tok*: レコード集合中の各レコードの要素を連続した位置に格納する配列．各要素は 4byte で表され，各レコードの先頭の要素の位置は 64byte 境界にアライメントされる．また，i 番目のレコードの末尾の要素と i+1 番目のレコードの先頭の要素の間は，i 番目のレコードの末尾の要素でパディングされる．
- *end*: 各レコードの配列 *tok* 中の末尾の要素の位置を格納する配列．

Xeon Phi は 64byte 境界にアライメントされたデータに対して高速にアクセスすることが可能である．したがって，配列 tok 中の各レコードの先頭要素を 64byte 境界にアライメントすることで，各レコードの SIMD レジスタへのロードが高速化

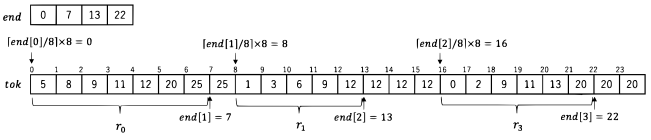


図 2　レコード集合を表す配列 *end,tok* (32byte 境界でアライメントした例)

され，より効率的な処理が可能となる．図 2 にレコード集合の配列表現の例を示す．

### 3.2 圧縮処理

圧縮処理では，2.5 節と同様に，レコード集合中の各レコードに対してシグネチャを計算し，それらを連続したメモリ領域に格納してシグネチャ集合を構築する．なお，2.4 節で述べたように，処理の効率化のため，本手法では $b = 1$ とする．また，512-bit 幅の SIMD 命令を活用するために，使用するハッシュ関数の数 $k$ は 32,64,128,256,512 のいずれかとする．

本手法では，レコード集合を複数個のレコードからなる互いに素な部分レコード集合に分割して各スレッドに割り当て，各スレッドは割り当てられた部分レコード集合中の各レコードを圧縮する．また，レコードの最小ハッシュ値の計算において，SIMD 命令による並列化を行う．Algorithm 1 にレコード集合の圧縮のアルゴリズムを示す．1 行目で，レコード集合を分割してスレッドに割り当てている．また，3 行目で，ハッシュ関数 $h_j$ について，レコード $r_i$ の最小ハッシュ値を計算している．なお，この処理ではレコードの要素を 16 個ずつに分割して複数個のグループを構築し，各グループのハッシュ値計算を SIMD 命令によって同時に実行する．そして，得られた複数個のハッシュ値に対して SIMD 命令を用いた reduce 処理を行うことで，最小ハッシュ値を計算する．

**Algorithm 1** レコード集合の圧縮

**input** : $R = \{r_1, ..., r_n\}, h_1, ..., h_k : \Omega \mapsto \Omega$
**output**: $R' = \{r'_1, ..., r'_n\}$

1: **for** $i = 1$ **to** $n$ **do in parallel** // スレッド並列化
2:  **for** $j = 1$ **to** $k$ **do**
3:  $val \leftarrow \min_{e \in r_i}(h_j(e))$
4:  $val$ の最下位ビットを $r'_i$ の $j$ 番目のビットに格納
5:  **end for**
6: **end for**

### 3.3 結合処理

結合処理では，2.6 節と同様に，フィルタリングを用いた結合処理を行う．本手法では，このフィルタリング処理を転置インデックスを用いて実装する．

Algorithm 2 にフィルタリングのアルゴリズムを示す．まず，2〜7 行目で，$R'$ をもとに転置インデックスを構築する．2 行目で，$R'$ を分割してスレッドに割り当てている．また，3〜6 行目で，$r'_i$ を $t_h$ 個の部分ビット列に分割し，各部分ビット列をキーにして転置インデックスを構築している．次に，8〜14 行目で，$S'$ を用いて転置インデックスを参照し，結合処理を行っている．8 行目で，$S'$ を分割してスレッドに割り当てている．また，10〜11 行目で $s'_i$ の $j$ 番目の部分ビット列をキーとして転置インデックスを参照し，ハミング距離の計算対象となるシグネチャのリストを取得している．そして，12 行目で，リスト中の各シグネチャと $s'_i$ とのハミング距離を計算している．なお，この処理において，リスト中の各シグネチャを複数個のシグネチャからなるグループに分割し，各グループについて $s'_i$ とのハミング距離計算を SIMD 命令によって同時に実行することで，効率的に処理を行っている．また，各グループ中のシグネチャの数はハッシュ関数の数 $k$ に依存する．例えば，$k = 32$ の場合，SIMD レジスタに収まるシグネチャ数は $512/32 = 16$ であるため，各グループのシグネチャ数は 16 となる．

**Algorithm 2** フィルタリング

**input** : $R' = \{r'_1, ..., r'_n\}, S' = \{s'_1, ..., s'_m\}, t_h$

1: $t_h$ 個の転置インデックス $inv_i$ の初期化
2: **for** $i = 1$ **to** $n$ **do in parallel** // スレッド並列化
3:  **for** $j = 1$ **to** $t_h$ **do**
4:  $key \leftarrow r'_i$ の $j$ 番目の部分ビット列
5:  $inv_j[key]$ に $r'_i$ を追加
6:  **end for**
7: **end for**
8: **for** $i = 1$ **to** $m$ **do in parallel** // スレッド並列化
9:  **for** $j = 1$ **to** $t_h$ **do**
10:  $key \leftarrow s'_i$ の $j$ 番目の部分ビット列
11:  $list \leftarrow inv_j[key]$
12:  $list$ 中の各シグネチャと $s'_i$ のハミング距離を計算し，$t_h$ 未満ならば出力
13:  **end for**
14: **end for**

## 4. 評価実験

本章では，提案手法に対する評価実験について述べる．

### 4.1 実験環境

本実験では Xeon Phi 7250 (KNL) を用いて評価を行った．なお，Xeon Phi 7250 は 68 個の物理コアと 384GB のメモリを搭載する．

プログラムの実装は C++で行い，コンパイラとして icpc 17.0.0 を使用した．また，コンパイルオプションとしては，-O3 と-qopenmp を使用し，OpenMP (注1) を用いてマルチスレッディングの実装をした．

本実験では，Enron (注2) と Large という二つのデータセットを使用した．Enron は，Email の本文からなる比較的小規模なデータセットであり，レコード数は 517,431 で，各レコードの要素数の平均は約 133 である．また，Large は人工的に作成したテキストからなる比較的大規模なデータセットであり，レコード数は 10,000,000 で，各レコードは Unix の辞書ファイル (/usr/share/dict/words) 中の 235,886 個の単語からランダム

(注1)：http://www.openmp.org
(注2)：http://www.cs.cmu.edu/ enron/

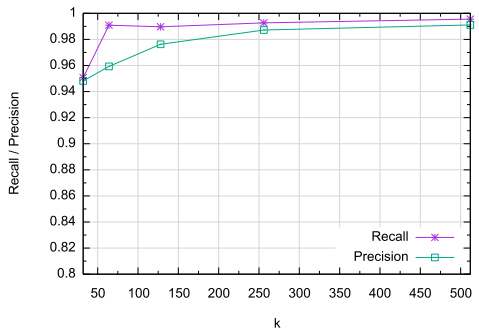


図 3　再現率と適合率

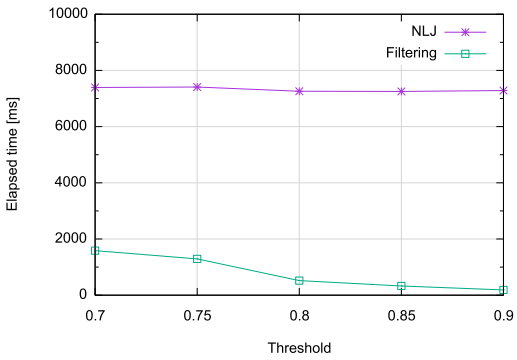


図 4　閾値による実行時間の変化

に複数個選択して生成されている．また，各レコードの要素数は平均が 150 で標準偏差が 40 の正規分布に従っている．

### 4.2　Jaccard 係数の推定精度の評価

本研究では，b-bit MinHash を用いており，その Jaccard 係数の推定精度はパラメータ $k$ に依存する．そこで，適当なパラメータを得るために，推定精度の評価実験を行った．

実験方法としては，Enron からランダムに選択した 10,000 個の集合からなるレコード集合について，閾値 $t = 0.9$ として，パラメータ $k$ を変化させて自己類似結合処理を行い，その再現率と適合率を測定した．実験結果を図 2 に示す．

実験結果より，$k = 32$ のとき，再現率・適合率ともに約 95%に達しており，十分な精度が得られていることがわかる．また，このときのシグネチャ集合のサイズは元のレコード集合の約 0.5%であり，非常に高い圧縮率を達成していることがわかる．

### 4.3　フィルタリングの評価

本節では，フィルタリングの性能の評価実験について述べる．本実験では，データセット Enron に対して，閾値 $t$ を変化させて実行時間を測定するとともに，Nested Loop Join で全ペアについて処理を行った場合の実行時間との比較を行った．実験結果を図 4 に示す．

実験結果より，本手法は閾値 $t$ の上昇に伴って計算コストが大きく削減され，$t = 0.9$ のとき，フィルタリングを行わない場合と比較して，約 38.8 倍の高速化となったことがわかる．

### 4.4　スレッド並列化効率の評価

本節では，マルチスレッドによる並列化の評価実験について

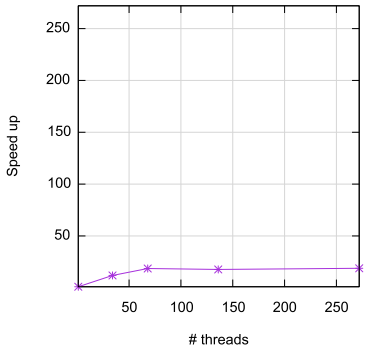


図 5　高 速 化 率

述べる．本実験では，データセット Enron に対して，スレッド数を変化させて高速化率の変化を測定した．実験結果を図 5 に示す．

実験結果より，本手法はスレッド数 68 のとき，高速化率は約 18.6 倍であり，あまり良い並列化効率が得られなかったことがわかる．この原因として，転置インデックスの参照においてキャッシュミスによるレイテンシが発生し，これがボトルネックとなっていることが考えられる．

### 4.5　CPU との比較

本節では，CPU 上での処理との比較実験について述べる．本実験では，データセット Enron と Large に対して，MinHash-KNL と MinHash-CPU と PPJoin-CPU の三つのプログラムについて実行時間を測定した．MinHash-KNL は，本手法を実装したプログラムであり，KNL 上で実行される．MinHash-CPU は，本手法中の 512bit 幅 SIMD 命令を 256bit 幅 SIMD 命令に置換し，CPU 上で実行可能にしたプログラムである．PPJoin-CPU は，CPU 上の集合間類似結合の state-of-the-art 手法の一つである PPJoin [14] をシングルスレッドによって実装したプログラムである．なお，PPJoin では MinHash を用いた手法とは異なり近似処理を行わないため，再現率・適合率は常に 100%である．Enron と Large に対する実験結果をそれぞれ図 6 と図 7 に示す．なお，PPJoin-CPU の Large に対する処理は終了しなかったため，図 7 は本手法と MinHash-CPU についてのみ実行時間を示している．

図 6 より，MinHash-KNL と MinHash-CPU は PPJoin-CPU と比較して非常に高速であり，$|R| = 100,000$ のとき約 11.6 倍の高速化率となっていることがわかる．この理由としては，PPJoin-CPU は厳密な解を求めているのに対して，MinHash-KNL と MinHash-CPU では近似的な解を求めることで計算コストを大きく削減していることが考えられる．

図 7 より，大規模なデータセットに対して，MinHash-KNL は MinHash-CPU と比較して高速であり，$|R| = 10,000,000$ のとき約 2.4 倍の高速化率となっていることがわかる．

## 5.　関 連 研 究

本章では，本研究の関連研究について説明する．集合間類似結合の高速化手法は，フィルタリングによる計算コストの削減

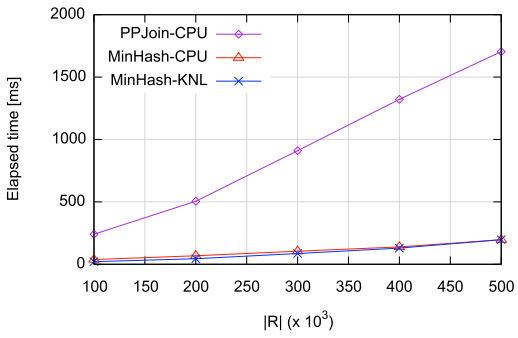


図 6　Enron に対する実行時間の比較

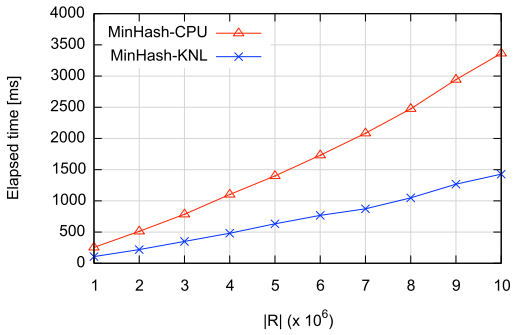


図 7　Large に対する実行時間の比較

と並列計算機を用いた高速化の二つのグループに大別される．

一つ目のグループは，フィルタリングによって明らかに類似していないペアを類似度計算対象から除外することで，計算コストの削減を図っている [2,4,14]．その代表的な手法の一つである PPJoin [14] は，Prefix filtering や Positional filtering などの幾つかのフィルタリング手法を採用しており，多くの計算コストの削減に成功している．

二つ目のグループは，PC クラスタ [11,12] やマルチコアプロセッサ [7] の並列処理性能を活用することで高速化を図っている．Cruz らが提案した手法は，GPU による並列処理による高速化手法 [3] である．この手法では，Li らが提案した One Permutation Hashing [9] を用いてレコード集合を圧縮し，Nested Loop Join によって結合するという処理を，GPU によって並列化している．

## 6. 結　　論

本研究では，b-bit MinHash に基づく集合間類似結合を Intel Xeon Phi で並列化することで処理の高速化を図った．また，評価実験によって，CPU 上での処理と比較して最大で約 2.4 倍の高速化を達成したことを示した．しかし，並列化効率が低く，多数の物理コアを活用しきれていないということがわかった．

現在の転置インデックスを用いたフィルタリングでは，メモリのランダムアクセスに起因するキャッシュミスが多発し，それがボトルネックになっていると考えられる．そこで，今後の課題としては，ソフトウェア・プリフェッチなどを活用することで，キャッシュミスによるレイテンシを削減することが挙げられる．

## 文　　献

[1] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, Vol. 60, No. 3, pp. 630 – 659, 2000.

[2] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[3] Mateus S. H. Cruz, Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Accelerating set similarity joins using gpus. *LNCS*, Vol. 28, pp. 1–22, 2016.

[4] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. An efficient partition based method for exact set similarity joins. *Proc. VLDB*, Vol. 9, No. 4, pp. 360–371, 2015.

[5] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pp. 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[6] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving main memory hash joins on intel xeon phi processors. Vol. 8, pp. 642–653, 02 2015.

[7] Yu Jiang, Dong Deng, Jiannan Wang, Guoliang Li, and Jianhua Feng. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In *EDBT/ICDT Workshop*, 2013.

[8] Ping Li and Arnd Christian König. b-bit minwise hashing. In *WWW, 2010*, pp. 671–680. ACM, 2010.

[9] Ping Li, Art B. Owen, and Cun-Hui Zhang. One permutation hashing. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *NIPS, 2012*, pp. 3122–3130, 2012.

[10] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. S. M. Goh. Mrphi: An optimized mapreduce framework on intel xeon phi coprocessors. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 26, No. 11, pp. 3066–3078, Nov 2015.

[11] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, Vol. 5, No. 8, pp. 704–715, 2012.

[12] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du. Fast and scalable distributed set similarity joins for big data analytics. In *ICDE*, 2017.

[13] Venu Satuluri and Srinivasan Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. Vol. 5, , 10 2011.

[14] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, Vol. 36, No. 3, pp. 15:1–15:41, 2011.