クラウドソーシング環境における実行プラン処理方式の提案

水澤　健† 田島　敬史†† 森嶋　厚行†††

† 筑波大学情報学群情報メディア創成学類　〒 305-8550 茨城県つくば市春日 1-2
†† 京都大学大学院情報学研究科　〒 606-8051 京都市左京区吉田本町 36-1
††† 筑波大学 知的コミュニティ基盤研究センター　〒 305-8550 茨城県つくば市春日 1-2
E-mail: †ken.mizusawa.2016b@mlab.info, ††tajima@i.kyoto-u.ac.jp, †††mori@slis.tsukuba.ac.jp

あらまし　本稿では，クラウドソーシング環境におけるデータ処理を記述した実行プランを，効率よく処理する方法について提案する．単純に実行プランをボトムアップに処理すると，場合によっては無駄なタスク処理を大量にクラウドソースすることが必要になる．一方，完全にトップダウンに処理すると，最小限のタスクのクラウドソースで済む可能性があるが，クラウドソーシングの特徴である大量の人を投入した並列処理が困難になる．また，商用のクラウドソーシングサービスを利用した処理を行いたい場合には，各サービス毎に，最小限のタスク数等，利用の際に満たすべき制約がある．本稿では，これらの問題を考慮した実行プラン処理方式を提案する．

キーワード　クラウドソーシング, 問合せ処理

1. はじめに

近年，ネットワーク技術の発達と普及に伴い，ネットワーク上の人々に問題解決のための一部の作業を割り振って，人々の知を利用することが可能となった．このような背景のもと，クラウドソーシングが注目を集めている [1]．クラウドソーシングとは，ネットワーク上にいる不特定多数の人 (以下，ワーカ) に作業を委託することをいう．このクラウドソーシングにおいて，ワーカに委託する作業の単位を一般に「タスク」と呼ぶ．また，作業をクラウドソーシングを利用して依頼する場合、一般的にはクラウドソーシングプラットフォームを通して行う。依頼者は，プラットフォームに用意されたタスクテンプレートに作業内容を入力，あるいは，対応するプログラムを記述することで，タスクをプラットフォーム上に掲載し，ワーカに依頼する．

本稿では，宣言型クラウドソーシングにおける実行プランの処理時のタスク発行制御の問題を取り扱う．宣言型クラウドソーシングとは，DBMS による宣言型問合せ処理と同様に，クラウドソーシングの宣言的な記述を元にタスク発行などの制御を行うものである [2] [3]．通常の DBMS の問合せ処理と同様，実行プランを用意する．図 1 は，簡単な実行プランの例である．R と S にそれぞれ含まれる 100 タプルのデータからクラウドソーシングを通して，目標となる Z のタプルを 10 タプル集める．図 1 において，実行プランのオペレータが四角で囲まれている作業を，マイクロタスクとして発行し，クラウドソーシング処理を行う事とする．例えば，図 1 の選択演算のタスクでは，提示されたデータが，選択条件を満たしているか否かを判定するタスクを発行する．R と S の結合演算タスクでは，提示された R と S のタプルのペアが，結合条件を満たしているか否かを判定するタスクを発行する．

このとき，どのようにタスクを発行するかは，様々な手法が存在する．発行するタスク数を最小化するには，1 つずつタスクを発行し，必要な結果を得られた時点で終了すればよい．例えば，まず結合判定タスクを一つ発行し，もし結合条件を満たせば，その結果に対して選択判定タスクを一つ発行する，と言うことを順次繰り返す．しかし，この方法では，タスク処理の並列性が落ちるため，クラウドソーシングの強みである人海戦術を利用できない．一方，実行プランの下位ノードから全てボトムアップに実行 (図 1 では，$R \bowtie S$ のタスクを全て実行) していくと，タスクの並列性は最大限活用できる．これは，一般的なクラウドソーシングサービスを利用するときによく行われている手順ではあるが，余計なタスク発行を行う可能性が高くなる．

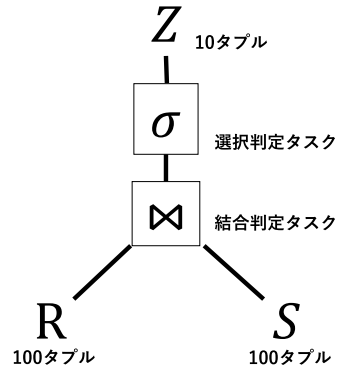

図 1　簡単な実行プランの例

また，一般的に用いられている商用のクラウドソーシングプラットフォームを利用したい場合には，プラットフォームごとに満たすべき制約がある．例えば“ タスクを依頼する場合には，最低 50 タスクは一度に発行し，まとめて掲載する必要がある ”“ プラットフォーム上のタスクの掲載期間は 2 週間以内 ” などの制約が挙げられる．したがって，必ずしも任意の並列処理が可能なわけではない．さらに，クラウドソーシングで得られる結果は必ずしも予想ができないため，どれだけのタスクを発行すると，十分な結果が得られるかが不明である．

本稿では，以上の問題を考慮したクラウドソーシング実行プ

ラン処理方式を提案する．具体的には，与えられた実行プランの元で，並列で処理するタスクの数を制御し，余計なタスクを行わないようにしながら，できるだけ並列性を高める．具体的には，演算子のイテレータの実装において，非同期プル (Asynchronous Pull) [4] の発行個数を制御する事によって，この目的を実現しようとするものである．

提案手法では，最初に作業目標が与えられた宣言から，目標達成のために必要な最低タスク発行数を逆算する．次に，発行したタスクから得られた結果を元に，データの選択率等のパラメータを見積もる．続いて，これらのパラメータから目標達成に必要な最低タスク数を再計算し，次のタスク発行数を動的に変化させる．この必要な最低タスク発行数，および，各パラメータを計算し，次のタスクの発行数を変化させるということを繰り返すことで，目標達成のために必要十分な数のタスクの発行でクラウドソーシングを実行することを目指す．

本研究の貢献は次のとおりである．(1) クラウドソーシングの実行プランにおけるタスク発行数の制御を，クラウドソーシングの特徴である並列性に着目しながら制御する手法を提案する．(2) シミュレーションにより，提案手法が，無駄になるタスク発行の増加を抑えつつ，並列性を増大させる事ができる可能性があることを示す．

本稿の構成は以下のとおりである．第 2. 節では関連研究について述べる．第 3. 節では，DBMS の問合せ処理で用いるイテレータモデルについて述べる．第 4. 節では，クラウドソーシング実行プランの表現のために，イテレータモデルを拡張したモデルについて述べる．第 5. 節では，プラットフォーム制約について述べる．第 6. 節では，クラウドソーシング実行を効率的に行うための提案手法について述べる．第 7. 節ではシミュレーションを行い，提案手法について評価をする．第 8. 節ではまとめと今後の課題について述べる．

2. 関 連 研 究

人と計算機を組合わせた処理を効率よく実行するための様々な問題は以前から知られており [5]，これまでいくつかの研究が行われてきた．CrowdDB [6]，CrowdOp [7] では宣言的なクラウドソーシングについて研究がなされている．どちらも Amazon Mechanical Turk [8](以下，MTurk) で，プログラム記述を通してタスクの発行を行っている．Qurk [9] はタスク表現とタスク処理を扱えるクエリ文を用いて，Mturk 上での処理方式を構築し問合せ処理を行うことで，タスクを発行する．

Deco [3] は同じく宣言的なクラウドソーシングであり，独自のルールの記述によりクラウドソーシング実行が可能である．Deco では，非同期プル [4] を利用したタスクの発行制御を行っている．ただし，Deco で採用されている制御は単純なものであり，最初に必要最低数のタスクを発行した後，得られた結果が十分でない間，一つずつタスクを発行する [10]．また，MTurk を通じて独自のプラットフォームを対応させる事を前提としている．本研究では，結果が十分でない場合，タスクの発行数を変化させることでクラウドソーシングの特徴である複数のワーカによる作業の並列性を確保する．

Eddies [11] は，データベースにおけるクエリ問合せ処理の最適化を，処理を行いながら実行プランを変化させることで同時に実行している．本研究では実行プランは変更せず，タスクの発行数を制御することで処理の最適化を図ることを目標としている．

3. イテレータモデル

本節では，DBMS の問合せ処理で広く知られているイテレータモデル [12] を説明する．イテレータモデルでは，各オペレータに，Open(), GetNext(), Close() という 3 つの実行プラン中のメソッドを用意する．ルートノードから処理を開始し，親のオペレータが子のオペレータの Open() をコールし，GetNext() を結果が得られるまでコールし，最後に Close() をコールする，という処理を再帰的に行う．

3.1 3 つのメソッド

イテレータモデルにおける．各オペレータに用意する 3 つのメソッドについて説明する．

- Open(): 最初にコールするメソッド．初期設定をする．例えば，オペレータが選択演算の場合には，結果を格納する出力バッファをオープンし，子オペレータの Open() メソッドをコールする，といった操作が考えられる．
- GetNext(): 次のタプルを読み出すメソッドである．例えば，オペレータがリレーション選択演算の場合には，自分より下の GetNext() を，選択条件を満たすタプルが現れるまで繰り返しコールし，結果を出力バッファに格納するといった事が考えられる．
- Close(): 全てのタプルを読み終えたら呼び出すメソッド．終了処理をする．例えば，オペレータが選択演算の場合には，出力バッファをクローズし，自分の子の Close() を呼び出すといった操作が考えられる．

イテレータモデルの利点は，様々な実行方式を一つの枠組みで表現可能である事である．例えば，オペレータが結合演算の場合には，Open() メソッドで，2 つの子オペレータ (結合対象) をオープンし，GetNext() を可能なだけ呼び出して結合演算を行い，全ての結果をバッファに全て格納し，クローズするという方法が考えられる．このとき，GetNext() メソッドは，単にバッファにある結果を順に上位に渡すだけの操作になる．これは，完全なボトムアップ処理を実現する事になる．逆に，GetNext() を最低数だけ呼び出して結果の処理を行うということ再帰的に繰り返す実行方式も表現することが可能である．

4. クラウドソーシング実行プランとイテレータモデル

本節では，通常のリレーショナル演算子におけるイテレータモデルを拡張する．拡張する点は，タスクの表現を許すこと，演算子の追加，および，イテレータモデルのメソッドの追加である．

4.1 タスクの表現

実行プランにおいてタスクを表現するための記法を導入する．具体的には，オペレータを四角で囲うことにより，そのオ

ペレータの処理をクラウドソーシングのタスクとして発行する事とする．オペレータが選択演算の場合には，子オペレータから得られた各タプル毎にタスクを発行し，そのタプルが，選択条件を満たすかどうかを判定することを依頼する．図 2(i) は，R からタプルを得て，選択条件を満たすかどうか判定するタスクを表している．結合演算の場合には，二つの子オペレータから得られたタプルの組合せ毎に，結合条件を満たす解かを判定する事を依頼するタスクを発行する．図 2(ii) は，R，S からタプルを得て，結合条件を満たす解かを判定するタスクを表している．

4.2 演算子の追加

クラウドソーシングのタスクとしてデータ入力が一般的であることを考え，データ入力演算子 $\delta_{attrs}(T)$ を導入する．これは，T の各タプルの $attrs$ 属性の内容を入力させるタスクである．T は省略可能であり，その場合は，任意個のタプルを入力できる事になる．これは，タスク指定 (四角で囲む) することを前提とした演算子である．図 2(iii) は，R の $attrs$ 属性を入力するデータ入力演算子 $\delta_{attrs}(R)$ を表している．

4.3 非同期プルの追加

GetNext() を行う際に，返り値を待たずに複数を発行できるように拡張する．これは非同期プルと呼ばれ，論文 [4] でその考えが提案されたものである．非同期プルでは，GetNext() はすぐに結果を返さずに，非同期で上位のオペレータに結果を返すことになる．本研究ではこの非同期プルの発行数を制御することで，効率的なクラウドソーシングを行うことを目標としている

5. プラットフォーム制約

クラウドソーシングプラットフォームには，利用する際に満たさなければならない制約が存在する場合がある．例えば，タスクの発行数に関しての制約が決められている場合がある．本稿では，同時にワーカが作業が行えるように，発行されたタスクをまとめてプラットフォームに掲載する行為を Run と呼ぶ．この 1 度の Run で掲載できるタスクの数には上限と下限が存在し得る．したがって，タスク発行数について制約が存在する場合には，この指定された範囲に収まるように発行数を決定しなければならない．

掲載期間にも制約が存在する場合がある．発行したタスクを長期間プラットフォーム上に掲載したままにはできず，掲載してから 2 週間が経過すると，掲載が終了される．この場合には，同じタスクを再び掲載するといった処理が必要になると考えられる．

6. 提 案 手 法

提案手法を説明するために，図 3 の記号と説明を用いる．まず，実行プランが与えられた時，そこに含まれるタスクノードの集合を $T = \{t_1, t_2, \ldots, t_n\}$ とする．実行プラン処理の過程のある時点において，これまでに行った Run の回数を r とし，r 回目依頼したタスクを処理した結果，これまで t_i が入力として得たタプルの集合を $In_i = \{in_{i1}, in_{i2}, \ldots, in_{ij}\}$，これまで

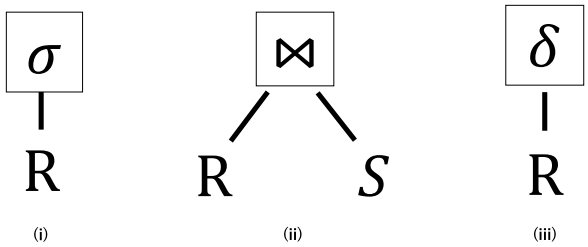

図 2 タスク表現の例

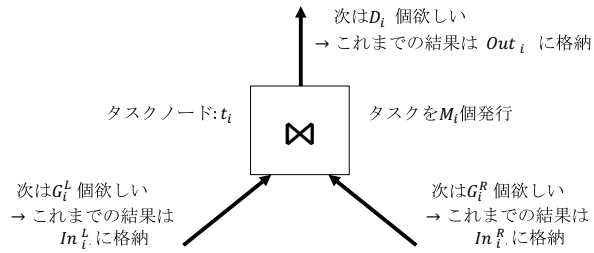

図 3 処理の流れの例

出力したタプルの集合を $Out_i = \{out_{i1}, out_{i2}, \ldots, out_{ik}\}$ とする．(二項演算子の場合には，右上に L,R を付けて区別する). また，各 t_i 毎における r 回目の依頼時の，タスクの結果として欲しいタプルの数を D_i, それを得るために同時に投げるタスク数を M_i，そのタスク数を結果として欲しい個数発行するために，子ノードに対して呼び出す GetNext() の数を G_i とする (これも同様に二項演算子の場合には，右上に L,R を付けて区別する).

次に発行数の初期化・更新を行うための変数について説明し，提案手法の流れを確認する．

6.1 変　　数

タスクの発行数の初期化や更新には次の変数を用いる．

【目標タプル】クラウドソーシングによる処理で最終的に求める目標タプルを z とする，実行プランのルートノードに相当する．

【増加タプル】Out_i において，前回の Run の時点での結果 Out'_i から増加したタプル集合を ΔOut_i $(= Out_i - Out'_i)$ とする．

【総タスク発行数】D_i を得るためにこれまでタスク t_i で発行したタスク数の総数を M_{total_i} とおく．

【選択率 : s_i 】選択条件，及び，結合条件におけるこれまでの選択率である s_i は, これまで発行したタスクが，どれだけ欲しいタプルを得られたかどうかを示す値である．したがって，タスク t_i の選択率 s_i について，

$$s_i = \frac{|Out_i|}{M_{total\_i}}$$

$$(0 \leqq s_i \leqq 1)$$

が成り立つ．この，選択率を求める処理を

$$s_i \leftarrow Set\_s(Out_i,\ M_{total\_i})$$

と表現することにする．

【寄与率：c_i 】結合演算における結果への寄与率である c は，親ノードである結合演算タスクの結果を生み出したタプルに，各子ノードがどれだけ寄与したタプルをもっていたかどうかを示す値である．したがって，タスク t_i の寄与率 c_i ついて

$$c_i^{(L\ or\ R)} = \frac{|\pi(Out_i)\ \cap\ |In_i^{(L\ or\ R)}|}{|(In_i^{(L\ or\ R)})|}$$
$$(0\ \leqq c_i^{(L\ or\ R)}\ \leqq 1)$$

が成り立つ．例えば，図 4 において，表 1 に表されるように $t_i(In_{i1}^L, In_{i1}^R)$，$t_i(In_{i2}^L, In_{i2}^R)$，$t_i(In_{i3}^L, In_{i1}^R)$ が Out_i に含まれたとすれば In_i^L，In_i^R の寄与率はそれぞれ $(2/4 = 0.5,\ 2/3 = 0.67)$ となる．

この，寄与率を求める処理を

$$c_i\ \leftarrow\ Set\_c(Out_i,\ In_i)$$

と表現することにする．

このとき，本提案手法では，次の手順で処理を行う．

（1） 初期化： 全ての t_i について，D_i, M_i, G_i を計算し，In_i および Out_i を空集合にする．

（2） ルートノードから GetNext() メソッドを G_i 個呼び出し，タスクノードに関しては下記の処理を行う．

（a） 子ノードからの GetNext() の結果が来るたびに，In_i に追加する．

（b） 子ノードからの結果がそろったら，タスクを M_i 個発行する．

（c） タスクの結果が来る毎に，結果を Out_i に追加する．

（d） 発行したタスクの結果が全てそろったら，In_i, Out_i の情報を元に，D_i, M_i, G_i を更新する．

以上の手続きで，問題となるのは，D_i, M_i, G_i の初期化とタスクの選択率 s_i，寄与率 c_i を用いた更新である．本提案手法では，次のようにタスク発行数を決定する．

- 最初に最も楽観的な仮定による最低限のタスク発行数を初期化する．
- 更新の際には，結果が全く得られなかった場合，タスク発行数を 2 倍にする．得られた場合には，これまでの結果をもとに選択率，寄与率を見積もり，欲しいタプル数 D_i，タスク発行数 M_i, GetNext() の発行数 G_i を決定する．

6.2 タスク発行数の初期化

タスク発行数の初期化は，最も楽観的な仮定で行う．この楽観的な仮定とは，発行したタスクを処理した結果，全てが欲しい結果に値するタプルを取得できるという仮定である．

6.2.1 選 択 演 算

選択演算では，選択条件の選択率 s を 1 としてタスクの発行数を決定する．上位のノードで必要なタプル数を N とすると，$D_i = M_i = G_i = N$ とする．例えば，図 4(a) の選択ノード t_1 の場合，$N = 10$ ならば $D_1 = M_1 = G_1 = 10$ である．

6.2.2 結 合 演 算

L と R の結合演算では，直積に関する結合条件の選択率 s を

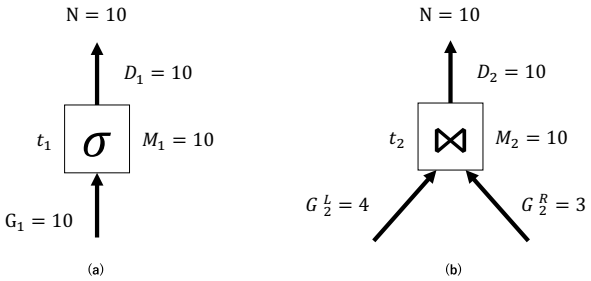

図 4 タスク発行数の初期化例

表 1 寄与率の具体例

$t_{ij}^L \setminus t_{ij}^R$	1	2	3
1	○	-	-
2	-	○	-
3	○	-	-
4	-	-	-

1 として値を決定する．また，結果への L の寄与率 c^L と R の寄与率 c^R が等しく 1 と仮定する．すなわち，上位のノードで必要なタプル数を N とすると，下記の様になる．

- $D_i = M_i = N$
- $(G_i^L, C_i^R) = (x, y)$ ここで x, y は $xy \geqq N$ となり，$x + y$ が最小となる組合せである．

例えば，図 4(b) の選択ノード t_2 の場合，$N = 10$ ならば $D_2 = M_2 = 10$ となり $(G_i^L, G_i^R) = (4, 3)$ である．$((G_i^L, G_i^R) = (3, 4)$ でも構わない)．

6.3 タスク発行数の更新

生成したタスクの処理が終了した後，次のタスクの発行数の更新を行う．

6.3.1 選 択 演 算

3 つのケースに分けることが出来る．

【ケース 1】 $|Out_i|\ =\ D_i$

タスクを発行した結果，欲しいタプルが全て集まった場合である．このとき各種値について 0 にするということを行う．

したがって更新は

$$D_i,\ M_i,\ G_i\ \leftarrow 0$$

と行う．

【ケース 2】 $|Out_i| \leqq D_i$ **であり，かつ，** $|Out_i|/M_{total\_i} = 0$

タスクを発行したものの，欲しいタプルが 1 つも集まっていない場合である．このとき，更新は D_i を変えず，M_i, G_i を倍増させるということを行う．

したがって更新は，

$$M_i \leftarrow 2M_i$$
$$G_i \leftarrow 2G_i$$

と行う．

【ケース 3】 $|Out_i| \leqq D_i$ **であり，かつ** $|Out_i|/M_{total\_i} \neq 0$

タスクを発行した結果，いくつか欲しいタプルが集まったが，まだ完全には欲しい数が集まっていないという場合である．このとき更新は，集めたタプルの数だけ D_i を削減，選択率 s_i を

計算し，それを元に次に投げるタスク数となる M_i を計算する．したがって更新は．

$$D_i \leftarrow D_i - |\Delta Out_i|$$
$$s_i = Set\_s(Out_i,\ M_i)$$
$$M_i \leftarrow D_i \times \frac{2}{1+s_i}$$
$$G_i \leftarrow M_i$$

と行う．

6.3.2 結 合 演 算

選択演算の時と同様に，3 つのケースに分けることが出来る．

【ケース 1】 $|Out_i| = D_i$

タスクを発行した結果，欲しいタプルが全て集まった場合である．これは選択演算の時と同様に

$$D_i,\ M_i,\ G_i\ \leftarrow 0$$

と更新を行う．

【ケース 2】 $|Out_i| \leqq D_i$，**かつ，** $|Out_i|/M_{total\_i} = 0$

タスクを発行したものの，欲しいタプルが 1 つも集まっていない場合である．これも選択演算の時と同様に

$$M_i \leftarrow 2M_i$$
$$G_i \leftarrow 2G_i$$

と更新を行う．

【ケース 3】 $|Out_i| \leqq D_i$ **であり，かつ** $|Out_i|/M_{total\_i} \neq 0$

タスクを発行した結果，いくつか欲しいタプルが集まったが，まだ完全には欲しい数が集まっていないという場合である．この場合，選択演算と異なり，結合演算では，In_i^L, In_i^R のどちらがほしいタプルを集めることに貢献しているかを示す寄与率 c_i の計算をする必要がある点である．したがって更新は，

$$D_i \leftarrow D_i - |\Delta Out_i|$$
$$s_i = Set\_s(Out_i,\ M_i)$$
$$c_i = Set\_c(Out_i,\ In_i)$$
$$M_i^L \leftarrow D_i \times \frac{2}{1+s_i} \times \frac{2}{1+c(t_i^L)}$$
$$M_i^R \leftarrow D_i \times \frac{2}{1+s_i} \times \frac{2}{1+c(t_i^R)}$$
$$C_i^L \leftarrow M_i^L$$
$$C_i^R \leftarrow M_i^R$$

と行う．

6.4 プラットフォーム制約の確認

実行プラン処理の中でプラットフォーム制約が存在する場合には，それを考慮した変更を行う．

【発行数制約： $p\_max,\ p\_min$**】**

タスクの発行数 M を決める際に，プラットフォーム制約により値が制限される場合，制約に基づいて M の値を変更する．このプラットフォームによるタスクの発行数の上限と下限をそれぞれ p_max, p_min とする．タスクの発行数を更新した際に，

Algorithm 1 提案手法の流れ

Input: タスクノード集合 : T, 目標タプル数 : d
Output: 最終目標タプル取得結果 : Out_d

```
1: for i = 1, ..., T.length do
2:    Initialize(D_i, M_i, G_i)
3:    Out_i ← φ, In_i ← φ
4: end for
5: while d > Out.root_node do
6:    root_node.Open()
7:    for i = 1, ..., T.length do
8:       child_node.GetNext() ← t_i.GetNext()
9:       if R_i == In_i then
10:         Run(R_i)
11:         Out_i ← R_i.result
12:      end if
13:   end for
14:   root_Node.Close()
15:   for i = 1...., T.length do
16:      Updatate(D_i, M_i, G_i)
17:   end for
18: end while
19: return Out_d
```

この範囲内に当てはまるように変更する．したがって，

$$M_i \leftarrow \begin{cases} p\_max & (p\_max < M_i) \\ M_i & (p\_min \leqq M_i \leqq p\_max) \\ p\_min & (M_i < p\_min) \end{cases}$$

と更新を行う．プラットフォーム制約がある場合には各ケースにおいて M の値について算出したあと，制約に当てはまる処理をおこなった後に，G の発行数について算出する．

【掲載期間制約： $duration$ **】**

依頼したタスクがワーカによって処理されず，プラットフォーム上に残り続ける場合が存在する．このとき，プラットフォーム上にタスクを残しておける期間に制限がされる場合，再び同じタスクを発行することで対応する．

6.5 提案手法の流れ

各タスクノードで用意されたイテレータモデルのメソッドを Algorithm1 に示す．

1 行目から 4 行目までは Open() の処理である．Open() では，タスク発行に使うためのタプルや，結果として得られたタプルを格納するための In_i, Out_i を，タプルを保存するためにオープンしておく．また，子ノードに対してタプルの要求ができるように，子ノードの Open() をコールする．

5 行目から 13 行目までは GetNext() の処理である．5 行目から 10 行目までは，親ノードから GetNext() がコールされた場合の処理で，返すタプルがあれば適応したタプルを返し，なければ自身も GetNext() を子ノードにコールすることでタプルを要求する．

11 行目から 13 行目までは，子ノードからタスクの発行のために必要なタプルを要求した結果が得られた場合，タスクに必

要なタプルを In_i に格納する．タスク結果が返されたら，結果を Out_i に格納する．14 行目から 20 行目までは Close() の処理である．In_i, Out_i をクローズし，各タスクノードにおける処理を終了する．

このとき提案する処理方式ではまず，各パラメータを初期化し，ルートノードの Open() をコールすることから開始する．ルートノードから必要分の GetNext() がコールされ，必要なタプルがあれば親ノードに返し，なければ子ノードに GetNext() を再帰的にコールする．必要なタプルが発行され，すべてのタスク処理が終了したら結果を保存し，一度 Close() する．得られた結果をもとに各パラメータについて更新を行い，再び GetNext() をコールする．という処理を目標タプルが取得できるまで繰り返す．

また，処理の中でプラットフォームの発行数制約，掲載期間制約に触れた場合，手法に基づき各変数について変更を加える．

7. シミュレーション

実行プラン処理方式における，一般的な処理方式と提案手法を用いた処理方式の違いを比較し，提案手法について評価する．シミュレーションでは実際のクラウドソーシング作業を想定し，人間の顔画像分類作業を取り上げる．画像の自動分類は機械学習などの分野で研究が進められており，その場合に扱う画像の教師用データセットの構築・評価などにクラウドソーシングを用いることがある [13]．本稿におけるシミュレーションでは，クラウドソーシングを通して，大量の顔画像から特定の条件に当てはまるものを取得すること想定し，図 5 の実行プランを用いて「金髪かつ同一人物のペア画像データを取得する」ということを行う．

7.1 設　　定

想定作業を行えるようなデータとタスクの設定を用いてシミュレーションを行う．顔画像は 100 人の人間の顔画像がそれぞれ 5 枚ずつ計 500 枚集められているものが，2 組存在するということを想定する．

7.1.1 デ ー タ

顔画像のデータは次のように定義する．まず，顔画像データを保持するリレーション R, S を設定し，500 タプルずつ用意する．各タプルは顔画像一枚についてのデータを表し次の要素を持つ．

- id：各画像を識別するための番号
- hair：顔画像の髪色を表す
- person：顔画像の人間を表す

設定からリレーション R，S に存在するタプルについて，id は 1 から 500 までが割り振られているものとする．hair は髪色を表し，この作業では金髪であるかないかがタスクにより挿入される．person には 1 から 100 までの人間を識別するための値がタスクにより挿入される．ここで，各セットに含まれる人間は同じ人間 100 人分とし，各セットから重複しない同一人物のペアが 100 組できることが約束されているものとする．

7.1.2 タ ス ク

このシミュレーションにおいて使用するタスクは，図 5 に示すように，(1) リレーション R あるいは S から取得できるタプル 1 つに対して，金髪であるかどうかを判定する**選択タスク:HairTask**．(2)R, S から金髪だと判断されたタプルをそれぞれ 1 つ用いて，それらが同じ人間の画像であるかどうかを判定する**結合タスク:PersonTask** の 2 つを用意する．これらのタスクを行った結果から異なる人間について選択し，目標である 10 組分に相当するタプルを取得することを行う．

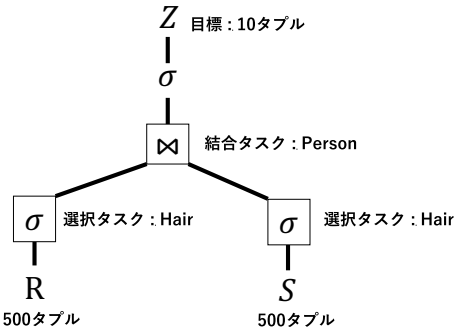

図 5　シミュレーションにおける実行プランの概要図

表 2　途中処理結果を用いた実行プランにおける相違点

比較対象	*Bottom up*	*Top down*	*Optimistic*	*Dynamic*
タスク数	最大限	最小数	必要最低限	必要最低限
優先度	子ノード	親ノード	子ノード	子ノード
発行数制御	行わない	行わない	常に必要最低限	動的に変化

7.2 使用した実行プラン処理方式

本シミュレーションにおいて使用した実行プラン処理方式は

- 発行可能な最大数で発行する方式:*Bottom up*
- 発行可能な最小数で発行する方式:*Top down*
- 最低目標取得数を元にする方式:*Optimistic*
- 本研究の提案手法を用いる方式:*Dynamic*

の 4 つであり，その違いについて比較したものを表 2 に示す．

【*Bottom up* と *Top down*】

Bottom up では，タスクを発行可能な最大数で発行する．例えば，HairTask の発行に関して，各リレーションには 500 タプルずつ存在するので，最大数としてそれぞれ 500 タスクを一度に発行する．これに対し，*Top down* では，タスクを発行可能な最小数で発行する．したがって，HairTask の発行に関して，500 タプルが存在していても，発行は 1 タスクずつとなる．また，発行するタスクの優先度が *Bottom up* では子ノードが優先，*Top down* では親ノードが優先される．親ノード優先の *Top down* では HairTask により，PersonTask の発行が可能になり次第，次に発行するタスクとして PersonTask を優先する．

これより *Bottom up* では大量のタスク発行により，無駄なタスク発行を行う可能性があるが，人間の作業の並列度が高くなる．逆に *Top down* では少量でのタスク発行により，一度に行えるタスクの数が減るために作業の並列度は低くなるが，タスクの発行数を削減できる．

【*Optimistic* と *Dynamic*】

Optimistic と *Dynamic* の 2 つは，前に記述した 2 つの処理方式が常に最大数，最小数での発行を行うのに対し，タスクの

発行数が処理の中で変更される．このとき，発行数の変更に関して提案手法で説明した，必要な最低タスク発行数を基準にする．*Optimistic* では常に必要最低数で次のタスクを発行するのに対し，提案手法を用いた処理方式である *Dynamic* では，結果を用いて次のタスクの発行数を動的に更新する．

7.3 概　　要

４つの処理方式について，次の２点で比較・評価を行う．まず，目標を達成するまでに全てのタスクノードで発行したタスク発行数の総数を「総タスク発行数 (以下，*Issue*)」とする．*Issue* を比較することで，タスクをどれほど削減できているかどうかを評価する．同じ *Run* 回目において依頼されたタスクは，全て並列作業で処理されることで，同時に結果が得られるとここでは仮定する．各処理方式によって一度に発行するタスク数は異なるため，*Issue* が同じでも *Run* が異なることが発生する．このとき，一度に発行されるタスクが多いほど人間が同時に作業が行えると考えられ，*Run* を比較することで作業の並列度をどれほど上げることができているかどうかを評価する．

シミュレーションでは各設定に基づいた違うデータを用いて複数回施行する．このとき，用意したデータに存在する金髪の人間の数により，目標タプルを取得する難易度は変化する．例えば，金髪の人間が 10 人ならば，最終的に必要な数が 10 人分であるので，このデータセットから取得できるペア画像データについて全種類取得しなければならない．25 人ならば，そのうちの 10 人分で足りるので，選択率として 10 人の場合よりも高くなることが考えられる．このシミュレーションでは用意された人間 100 人のうち，金髪の人間が (1)10 人の場合 (2)25 人の場合の 2 つの場合について 50 回施行する．またワーカの回答精度を 100%とし，タスクを行った結果，金髪であるのに，金髪でないといった回答は発生しないものとする．

7.4 結　　果

シミュレーションの結果について，各ケースにおいて 50 回施行したときの，*Issue* と *Run* の平均値の表と散布図を用いて説明する．

【ケース 1：金髪の人間が 10 人の場合】金髪の人間が 10 人の場合についてのシミュレーション結果についてまとめる (表 3 は 50 回施行の平均値，図 6 は *Issue* と *Run* についての散布図を示す)．*Bottom up* では *Issue* と *Run* について，設定から $Issue = (500 \times 2) + (50 \times 50) = 3500$，$Run = 1 + 1 = 2$ が得られる．したがって，提案手法 *Dynamic* は *Issue* について約 *Bottom up* の 34%の値で作業を終了できたことが分かる．

図 6 から *Top down* , *Optimistic* , *Dynamic* について，*Issue* について平均値としては *Top down* が最も少なく．*Run* は *Dynamic* が最も少ない結果が得られた．また全体として，*Issue* の値がが多くなるほど，*Run* について差が開き，*Dynamic* が *Top down* や *Optimistic* に比べて並列性を確保していることが明らかになった．

【ケース２ ：金髪の人間が 25 人の場合】

金髪の人間が 25 人の場合についてのシミュレーション結果についてまとめる (表 4 は 50 回施行の平均値，図 7 は *Issue* と *Run* についての散布図を示す)．10 人の場合であるケース 1 に比べて全体的に *Issue*，*Run* の値は少なくなっている．結果の処理に応じてタスクを発行するので，必要なタプルを取得できる確率が高いほど，最終的な値が小さくなる結果が得られた．

表 3　ケース 1：発行数と依頼回数の平均値

	Bottom up	*Top down*	*Optimistic*	*Dynamic*
Issue	3500	1143.6	1177.8	1222.1
Run	2	1016.5	595.7	263.2

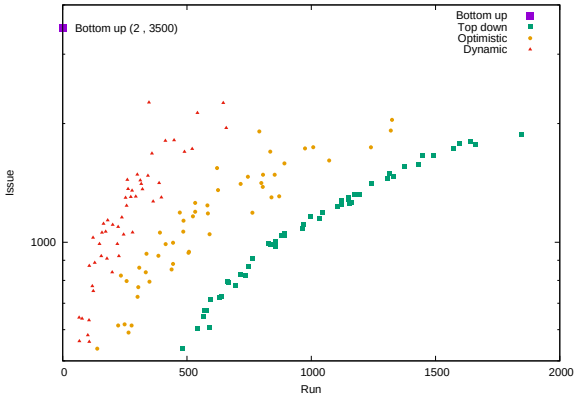

図 6　ケース 1：金髪 10 人の場合のタスクの発行数と依頼回数

表 4　ケース 2：発行数と依頼回数の平均値

	Bottom up	*Top down*	*Optimistic*	*Dynamic*
Issue	16625	671.3	685.3	689.6
Run	2	618.9	238.1	131.1

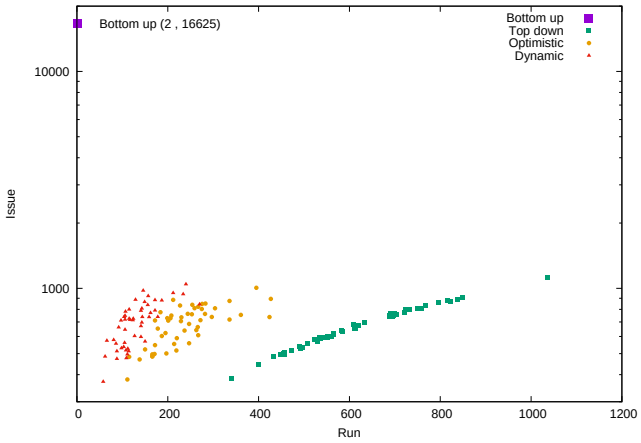

図 7　ケース 2：金髪 25 人の場合のタスクの発行数と依頼回数

図 7 から *Top down* , *Optimistic* , *Dynamic* について，金髪の人間が 10 人の場合と同様に，*Issue* について平均値としては *Top down* が最も少なく．*Run* については *Dynamic* が最も少ない結果が得られた．ただし全体として，*Optimistic* と *Dynamic* の結果がケース 1 に比べて近いものが得られた．取得目標であるタプルがケース 1 よりも取得しやすいために，発行数の更新を行う際の増加数の変化が少なくなったことが原因

だと考えられる．

7.5 考　　察

今回のシミュレーションではタスクの総発行数を示す *Issue* と依頼回数を示す *Run* の値について，提案手法を用いた処理方式である *Dynamic* と他の処理方式を比較した．結果として *Dynamic* が，*Bottom up* よりも *Issue* を削減し，*Top down* に比べて *Run* の値を抑え，作業の並列性を確保できていることが確認できた．また，動的にタスクの発行数を変化させる *Dynamic* のほうが，常に最低数で発行する *Optimistic* よりも，発行数をさほど増加させることなく，並列性を上げていることも確認できた．

しかし，今回のシミュレーションは単純なクラウドソーシングを想定しているためいくつかの点でまだ考慮すべき点がある．第一に，タスクの処理について，実際のクラウドソーシング上ではワーカの回答精度は 100%ではなく，依頼者が必要としない回答が多く発生する可能性があるため，選択率がシミュレーションよりも低下する可能性がある．第二に，依頼回数である *Run* について，実際のクラウドソーシングプラットフォーム上で，人間の手作業により依頼する場合の回数としては多いという点がある．シミュレーションでは *Run* の値が 100 を超えており，仮に 1 日に 1 度，手作業で依頼したとすれば，手間がかかるだけでなく，作業が終了するまでに 100 日以上を要する．

今後はこれらの考慮点を踏まえた，実際のクラウドソーシングで行う場合のタスク発行数を動的に決定する際の基準値について検討する必要がある．例えば，*Run* の値について上限を設定し，基準値を大きくすることで作業の並列性を高め，上限を超えない範囲で作業を終了できるような処理を目指すというようなことが考えられる．

8. まとめと今後の課題

本稿ではクラウドソーシング環境における処理方式実行プランについて提案を行った．シミュレーションにより，目標に必要な最低タスク数と処理結果から導けるデータの選択率など各パラメータを用いて，タスクの発行数を動的に変化させる手法を用いた処理方式が有効である可能性を示した．

今後の課題として (1) 実際に使われているクラウドソーシングプラットフォーム上での評価実験を行うこと．(2) 提案手法における *Run* の回数について上限の制約がある場合の，処理の最適化について検討することなどがあげられる．

謝　　辞

本研究の一部は，JST CREST および JSPS 科研費（#25240012）の支援による

文　　献

[1] Jeff Howe. The rise of crowdsourcing. *Wired Magazine*, Vol. 14, No. 06, pp. 1–5, 2006.

[2] Atsuyuki Morishima, N. Shinagawa, and T. Mitsuishi. Cylog/crowd4u: a declarative platform for complex data-centric crowdsourcing. *Proceedings of the VLDB Endow.*, Vol. 5, No. 12, pp. 1918–1921, 2012.

[3] Aditya Ganesh Parameswaran, Hyunjung Park, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. Deco: declarative crowdsourcing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pp. 1203–1212. ACM, 2012.

[4] Roy Goldman and Jennifer Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and theWeb. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data - SIGMOD '00*, Vol. 29, pp. 285–296, 2000.

[5] Aditya Parameswaran and Neoklis Polyzotis. Answering Queries using Humans, Algorithms and Databases. *Proceedings of the 2011 Conference on Inovative Data Systems Research*, pp. 160–166, 2011.

[6] Michael J Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. Crowddb: answering queries with crowdsourcing. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 61–72, 2011.

[7] Ju Fan, Meihui Zhang, Stanley Kok, Meiyu Lu, and Beng Chin Ooi. Crowdop: Query optimization for declarative crowdsourcing systems. *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*, Vol. 27, No. 8, pp. 1546–1547, 2016.

[8] Amazon mechanical turk. `https://www.mturk.com`.

[9] Data Systems, Adam Marcus, Eugene Wu, David R Karger, Samuel Madden, and Robert C Miller. Crowdsourced Databases : Query Processing with People. *Cidr*, 2011.

[10] Hyunjung Park, Aditya Parameswaran, and Jennifer Widom. Query Processing over Crowdsourced Data. 2012.

[11] Ron Avnur and Jm Hellerstein. Eddies: Continuously adaptive query processing. *ACM SIGMoD Record*, pp. 261–272, 2000.

[12] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems: The Complete Book (GOAL Series)*. Prentice Hall, us ed edition, 10 2001.

[13] a Sorokin and D Forsyth. Utility data annotaton with amazon mechanical turk. *Proceedings of the 1st IEEE Workshop on Internet Vision at CVPR 08*, No. c, pp. 1–8, 2008.