データベースシステム 116－34
（1998. 7. 9）

永続データをサポートする実時間推論シェル

西山 智　　小野 智弘　　小花 貞夫
(株)KDD 研究所

本論文では，網管理等の分野でのエキスパートシステム (ES) への適用を目的として，網から発生するアラーム等の外界の変化に実時間で応答可能で，かつデータベース上の大量のデータを用いて推論できる推論シェルの実装と評価について論じる．最良優先探索に基づくマッチングアルゴリズム LEAPS に外界の変化を実時間で取り込む拡張を行い，マッチングアルゴリズムとして組み込んだ．また，データをデータベース上に永続的に格納し，その永続データを主記憶上の非永続データと区別なく推論可能とした．評価の結果，ほぼ実時間で外界のデータが推論に反映できており，また永続的なデータを推論に用いる場合は，マッチング処理で非永続データに対して 1/4.5～1/7.3 程度の性能で実現できた．

Real-time Production System Shell Supporting Persistent Data

Satoshi Nishiyama　　Chihiro Ono　　Sadao Obana
KDD R&D Laboratories, Inc.

This paper discusses the design and the evaluation of a production system shell which can take in the changes of data to the inference process in real-time and which can handle large amount of data. To realize the former feature, we developed a new matching algorthm based on LEAPS, a best-first matching algorithm. The new matching algorithm can interrupt the matching step at any time when the data is updated and can resume the matching using the updated data. To realize the latter feature, we added database management functions to the production system shell so that the shell can directly match the rule with the persistent data on the database. We confirmed the correctness of our design through the evaluation with sample programs.

1　はじめに

網管理等の分野でのエキスパートシステム (ES) は，例えば網から発生するアラーム等の外界の変化に実時間で応答 (実時間応答) し，かつ網のトラフィック情報等大量のデータを用いて推論 (大量データ推論) する必要がある．これまで，実時間応答や大量データ推論が可能な汎用の推論シェル (以下 ES シェルと呼ぶ) はなかった．従って，このような ES を構築するためには実時間応答性に関する要求条件を緩和したり，大量データを扱うためデータベースと連係するためのプログラムを ES の一部として作成する必要があった．筆者らは，このような ES の構築を容易とするために，実時間応答と大量データ推論が可能な ES シェルを開発した．このシェルは，最

良優先探索を行うアルゴリズム LEAPS[1] をベースに実時間で外界の変化が推論に反映できる処理を拡張したマッチングアルゴリズムを使用する．また大量のデータをデータベースに永続データとして格納し，その永続データと主記憶上の非永続データを区別無く推論に使用可能とした．本論文では，その実装と評価について報告する．2章では，実時間応答と大量データ推論に関する従来の汎用 ES シェルの問題点を述べる．3章では，これらの問題に対する本 ES シェルでの解決方法を論じる．4章では，解決方法に基づくシェルの設計と実装を述べる．最後に，5章ではシェルの性能評価を示す．

2 従来の汎用 ES シェルの問題点

OPS5[2], OPS83[3] 等，従来の汎用 ES シェルには実時間応答と大量データ推論に関して以下のような問題点がある．

2.1 実時間応答

網管理等の分野への応用では，重要障害が発生した場合の障害箇所の特定や原因の特定など，通常の推論処理中に優先度の高い推論が必要となる場合がある．従来の汎用 ES シェルにはルールに優先度を指定できるものがあるが，そこで使用されている，RETE[4], TREAT[5], LEAPS 等のマッチングアルゴリズムは，マッチング処理において実行可能なルールとデータ (WME:Working Memory Element と呼ぶ) の組 (この組をインスタンシエーションと呼ぶ) を全数探索し，その後競合解消戦略と呼ばれる規則に従って実行すべきインスタンシエーションを選択する．一旦マッチング処理を開始すると，その時点での全てのインスタンシエーションが見つかるまで処理を中断しない．従って，マッチング処理中に ES シェルの外部でデータが変更されても，それが WME として取り込まれるのはマッチング処理が終わった後となる．これらのアルゴリズムのマッチング処理はルール内に含まれる各条件節 (CE:Conditional Element) に対応する WME を入れ子ループによりジョインするため，その計算量はデータ量を n, CE 数を c とすると $O(n^c)$ である．ルールが複雑になる (CE 数が増加する) とマッチング処理に必要な時間は指数的に増加する．このため，従来の汎用 ES シェルではルールに優先度を指定しても，外界のデータの変化を WME として推論に取り込み優先度の高い推論を開始するまでの時間を保証する事は困難であった (図 1(a))．

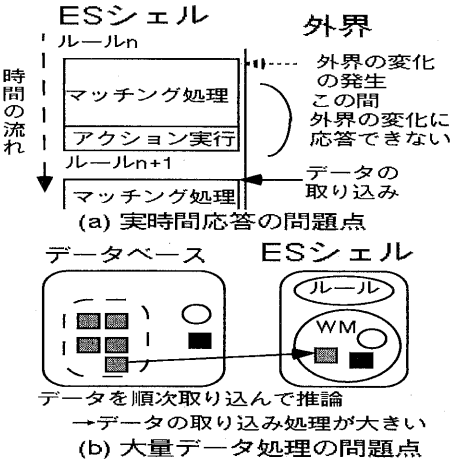


図 1: 従来の汎用 ES シェルの問題点

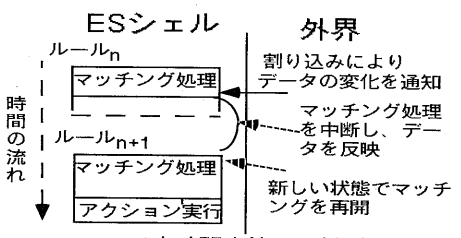
2.2 大量データ推論

従来の汎用 ES シェルは，推論に使用する WME を主記憶に格納する．このため，そのプロセスが確保できるメモリ量を超える大量のデータが推論に必要な場合，ES の実装に何らかの工夫が必要である．例えば，図 1(b) に示すようにデータをデータベースに格納し，そこから順次データを WME として読み込んで推論し削除し，また次のデータを読み込むといった手順が必要となる．しかしながら，この手順は，ES シェルでの応用プログラムとして記述する事となり，応用プログラムが複雑化する．また，動的に WME として登録/削除する負荷が大きく，大量のデータを高速に扱えない．さらに，一時的ではあるが，データベースとシェル上のデータと WME の一貫性が保証されないこととなり，推論の精度が低下する．

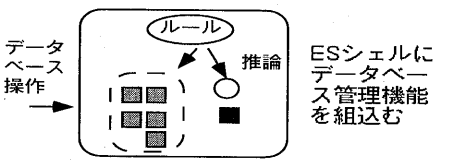
3 実時間推論シェルでの解決方法

本 ES シェルでは上記の問題点を以下の方法で解決する．

1. 実時間応答:
   マッチングアルゴリズムとして，最良優先探索を行う LEAPS を拡張し，マッチング処理中に外界からのデータの変化を割り込みにより受け付ける，実時間対応のアルゴリズムを開発し使用する (図 2(a))．これにより，全てのインスタン



(a) 実時間応答への対応



(b) 大量データ処理への対応

図 2: 実時間応答と大量データ推論の実現

シエーションの探索を待つことなく外界の変化が推論に反映できる.

2. 大量データ推論:
   ES シェルにデータベース機能を付加して, WME を永続的に格納し (永続データ), その WME と主記憶上の WME を区別無く推論に利用できるようにする. これによりメモリ量に制限されず大量データを直接用いて推論できる (図 2(b)).

次節以降で, それぞれの解決方法を詳細に述べる.

3.1 実時間応答のアルゴリズム

LEAPS は競合解消戦略で選択されるべきインスタンシエーションが最初に見つかるように最良優先探索を行うマッチングアルゴリズムである. 見つかったインスタンシエーションは, 競合解消戦略で選択されるべきものであるので, インスタンシエーションが見つかると直ちにそれを実行できる. 図 3に LEAPS でのマッチング処理におけるジョインの実現方式を示す. ここで, A, B は WME のクラス, $b_1, a_2, ..., b_n, a_{n+1}$ は各々のクラスに含まれる WME を示す. ここで WME に付与した添字は, 全ての WME を順序付けるためのタイムスタンプ (タイムタグと呼ぶ) である. LEAPS はスタックを持ち, 新たに生成された WME は, マッチングの起点となるシード WME として, スタックにプッシュされる. マッチングでは, スタックからシード WME をひとつポップし, その WME を起点としてマッチング (ジョイン) を行う. 図 3に示

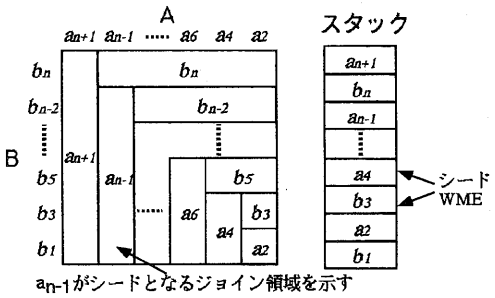


図 3: LEAPS によるマッチング (ジョイン)A⋈B の実現

すように, ある WME が起点となってジョインされる領域は, ジョインされるべき全領域の一部の方形領域である. また, WME が削除された場合, その WME をスタックから取り除く事で結果的に実行されないインスタンシエーションの検索を行わない.

LEAPS は, OPS5 での LEX, MEA の厳密な実現ができないなど, 実現できる競合解消戦略に制限があるものの, 不要なインスタンシエーションを探索しないため, 一般に RETE や TREAT 等, 全てのインスタンシエーションをまず探索し, 競合解消戦略に基づき実行するインスタンシエーションを選択するという幅優先探索アルゴリズムと比較して高速である [1]. また, インスタンシエーションが見つかるとマッチングを中断し実行に移る方式であるため, 事実の変化によりマッチングを中断する機構が組み込みやすい. このため, 本 ES シェルでは LEAPS をベースとし, 実時間で外界の変化に応答する機構を拡充する.

図 4に拡張した LEAPS のメインループを示す. 拡張アルゴリズムではマッチングループ中に外界からの WME の変更を割り込みとして受け付けると, 直ちにマッチングを中断して途中経過 (マッチングのループで最後にジョインした WME の組) をスタックにプッシュする. 続いて WME の変更を反映させ, 対応する α メモリ (条件に合致する WME に対する索引の一種. 本論文では AMEM と示す) に WME を追加し, スタックにもプッシュする. 一連の操作が終了した後に, スタックからシードをポップしてマッチングを再開する.

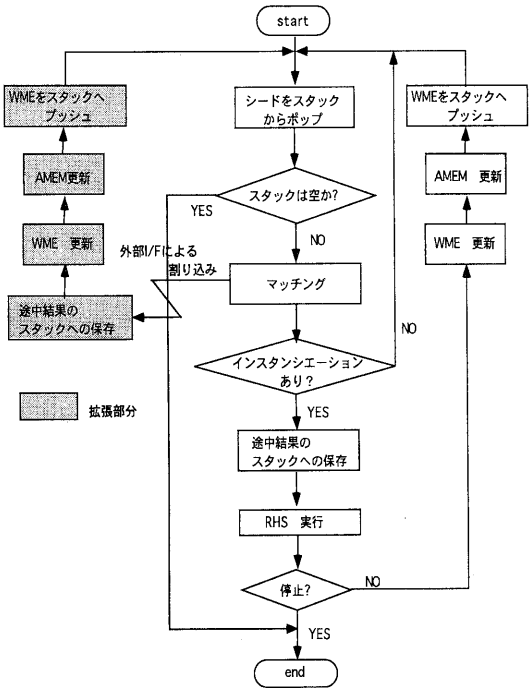


図 4: 拡張した **LEAPS** のアルゴリズム (メインループ)

3.2 永続データを用いた大量データ推論

LEAPS は RETE, TREAT と同様に全ての WME をシードとしてマッチング操作を行なうアルゴリズムであり, かつ全ての WME がプログラムの起動後に生成されることを仮定している. ところが, 本 ES シェルでは非永続 WME のみならず, プログラムが起動された時点で既に存在する永続的 WME も扱わなければならない. 主記憶上の非永続データとデータベース上の永続データを区別なく推論可能とするために, 以下の機構を組み込む.

- α メモリの永続化と共有
  大量の永続データを効率的にマッチングするためには, 条件に合致する WME に対する索引 (LEAPS では AMEM) が重要である. 本 ES シェルではルールは動的に変化しないことを仮定しているため, 永続 WME に対する AMEM も永続化した. これにより永続 WME とその AMEM の内容は常に一貫しており, シェルを起動する毎に AMEM を作成する手間を省く事ができる. AMEM は CE 毎に生成されるため, 多数のルールを含む応用プログラムでは, 単一の WME に対して多数の AMEM が存在し, 大量の格納領域を必要とする. このため, 等価な (条件が等しい)CE 間で AMEM を共有することで, AMEM の格納効率を図る.
- スタックとタイムタグ
  スタックについては, ES シェルの過去の起動時のスタックの内容はその時点での推論の途中経過を示しており, 新しい起動時には必ずしも有効でない. このため, 本 ES シェルではスタックは永続化せずにメモリ上におく. 推論はシェル起動後のデータの変化差分のみを対象として行う. 一般に, プロダクションシステムでは先頭の CE により推論の制御を行い, その制御用 CE に用いられる WME は起動後に作成される. この場合, 制御用の WME は更新 (削除+追加) を繰り返すので, 過去のスタックの内容は破棄しても実行に問題がない.

  タイムタグについては, 永続 WME も含めて WME の新旧関係とタイムタグ値の大小関係を常に正しく保つため, 最新のタイムタグをデータベース上に永続的に格納することとした. 新たな WME には永続/非永続の区別なく最新のタイムタグの最大値+1 をタイムタグとして付与する.

4 設計と実装

4.1 設計方針

本 ES シェルでは, 3章で示した解決方法に加えて以下の点を設計方針とした.

- 将来の並列処理を考慮したソフトウェア構成とする.
- C++によるオブジェクト指向の実装とする.
- ルール記述のためのコンパイラを提供する.
- UNIX 上に実装する.

4.2 ソフトウェア構成

本 ES シェルのソフトウェア構成を図 5に示す. 実行時に必要な各プロセスのメインルーチンや推論機能等はライブラリの形態で ES の開発者に提供される. 開発者は応用プログラムを OPS83 準拠のルール記述言語で記述し, ルールコンパイラがそれを C++

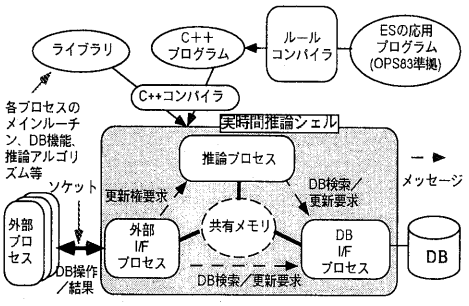


図 5: ソフトウェア構成

に変換する．そのソースとライブラリを C++コンパイラによりコンパイルし，その応用に対する ES を生成する．生成されるソフトウェアは，推論プロセス，DB I/F プロセス，外部 I/F プロセスの 3 プロセスからなる．これら 3 つのプロセスは，共有メモリを介して情報を共有する．プロセス間は，UNIX の提供するメッセージキューと共有メモリを併用して通信する．

推論プロセス

- 3章で述べた実時間応答の推論アルゴリズムを用いて推論する．
- 外部 I/F プロセスとの間の更新ロック管理を行なう．

外部 I/F プロセス

- 外界に対して，make(作成)，modify(更新)，remove(削除)，search(検索) の 4 種類のデータベース操作を提供する．

DB I/F プロセス

- 他のプロセスからの要求により DB と共有メモリ間でデータを含むページの読み込み/書き戻しを行なう．
- トランザクション機能を提供する．
- 拡張可能 DBMS である ASSIST/D[7] をベースに実現する．

4.3 オブジェクト指向の実装

4.3.1 クラス定義

図 6に示すような C++のクラスを定義した．図 6 での上段のクラスはスキーマ情報を格納したクラスである．中段のクラスは応用で記述された事実の型 (WM:Working Memory) や推論アルゴリズムに必要

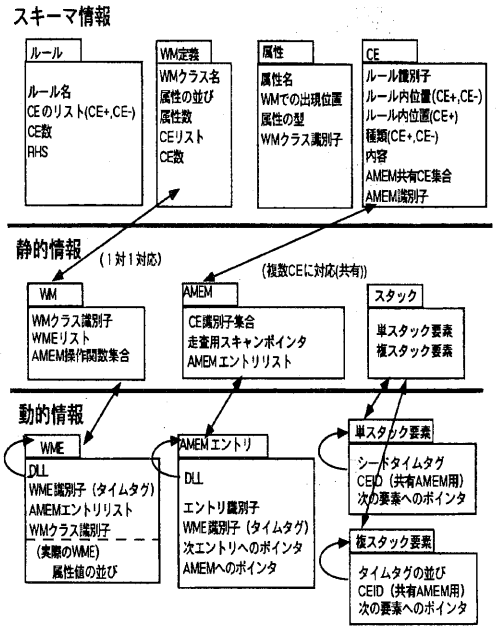


図 6: 推論部分のクラス定義

な AMEM のためのクラスである．下段のクラスは，WM や AMEM に格納される要素 (WME や AMEM エントリ) のためのクラスである．

4.3.2 永続/非永続データの混在

非永続サブクラスではポインタとして物理的なアドレスを使用できるが，永続サブクラスではデータベース上の永続データを扱うためポインタとして論理的なアドレス (ここではディスク頁番号+オブジェクト識別子) を使用する必要がある．ルールコンパイラのコード生成を容易とするために，WME や AMEM エントリ等への操作はそのクラスへの仮想関数として定義し，クラス操作時に永続/非永続データ (オブジェクト) の混在を意識する必要をなくした．これらのクラスに対しては永続/非永続毎にサブクラスを設け，実際の操作関数を定義した．

4.3.3 共有メモリによる情報共有

WME や AMEM エントリ等のクラス定義では C++の仮想関数を使用しているが，仮想関数を用いるとオブジェクトには物理的なアドレス (仮想関数へのポインタ) が含まれる．しかし仮想関数の実体がマップされるアドレスはプロセス毎に異なるため複数プロセスでオブジェクト自身を共有メモリを介

して共有することはできない. このため共有メモリにはオブジェクトではなく必要な情報を含む構造体のみを載せて, 各プロセス内のオブジェクトから指し示してアクセスすることとした.

4.4 ルールコンパイラ

OPS83準拠の言語で書かれた応用プログラムを入力とし, 本 ES シェルを構成する 3 つのプロセスを生成する際に, 応用プログラムに依存したコードを出力するコンパイラである. パーザ部分には YACC, LEX を使用した. ソフトウェアの規模は全体で 7K 行程度である.

入力言語は OPS83 に対して以下の 2 点の変更を加えたものである.

- ディスク上に書かれる永続的 WME を作成するために, エレメント宣言に以下のようにキーワード persistent を指定可能とした.

  <elementdef>::= [persistent] element (<要素の並び>)

- OPS83の組み込み関数のうち, LEAPSアルゴリズムでは使用しない競合集合に関するもの (select, qselect, cssize 等 8 つ) を削除し, 代わりに発火すべきインスタンシエーションを求める関数 getfirable を追加した.

また出力については, 下記の推論ライブラリにおいてデータの永続／非永続を C++の仮想関数を用いて隠蔽させているため, ルールコンパイラはデータの永続／非永続に依存しないコードを生成する.

推論ライブラリは実行プロセスのメインルーチンや推論機能などのライブラリであり, 推論プロセス用, DB I/F プロセス用, 外部 I/F プロセス用, 共通関数群の 4 つのライブラリからなる. ソフトウエアの規模は全体で 24K 行程度である.

5 評価

本 ES シェルを以下の方針で評価した.

1. 永続データの推論性能
2. 外界のデータの変化に対する実時間応答性能
3. 既存汎用 ES シェルとの性能比較

5.1 評価環境

評価では, 図 7に示すルールシステム (以下 simpleJoin と呼ぶ) を使用した. 図 7において, クラス a, b の宣言部の [persistent] はデータの永続/非永続に応じてデータの永続性を示すキーワード persistent を付与した/ しないことを示す. この simpleJoin に対して, クラス a および b について初期に生成するデータ件数を変化させた. 評価は, Sun SPARC Server 1000 (Solaris2.5.1) 上で行い, 非永続データ格納用に 2M バイト, 永続データのバッファ用に 3M バイトの共有メモリを割り当てた.

```
-------------------------------------------
type a = [persistent] element (
     a1: integer ;
     a2: symbol ; );
type b = [persistent] element (
     b1: integer ;
     b2: symbol ; );
type c = persistent element ( c1:integer; );
rule match_AB { -- 条件節 (CE) を 2 つ含む
   &CE1 ( a ) ;
   &CE2 ( b b1 = &CE1.a1 ; );
                -- a と b を join --
   -->
      -- 実際のアクション無し -- }
rule match_C {
      ( c ) ;
   -->
      -- 実際のアクション無し -- }
-------------------------------------------
```

図 7: 評価に使用したルールシステム simpleJoin

5.2 永続データと非永続データによる性能変化

永続データと非永続データを用いる場合の推論性能を比較するために, クラス a および b をそれぞれ永続データ, 非永続データとした場合の, データ作成時間を測定した (評価 1,2). また, クラス a, b と永続, 非永続の全ての組合せに対してマッチング処理時間を測定した (評価 3〜6). なお, クラス a, b の順で作成する事によりクラス b のデータが常にシードとなるようにした. 図 8に 1 データ当りの作成時間を, また図 9に 1 シード当りのマッチング処理時間を示す.

(1) データ作成性能： 評価 2 より非永続データの作成性能は, データ件数にかかわらずほぼ一定であった. 評価 1, 2 より永続データの性能は, 非永続データに対して 1/20〜1/17.7 程度であり, データ件数の増加に伴い改善する傾向にある. DB I/F プロセスでデータをバッファしている事から, そのヒット率の向上によると考えられる. データ件数をクラス a, b それぞれ 3.3×10^3件とした場合, 評価 2 は共有メモリ不足により作成できなかったが, 評価 1 は DB

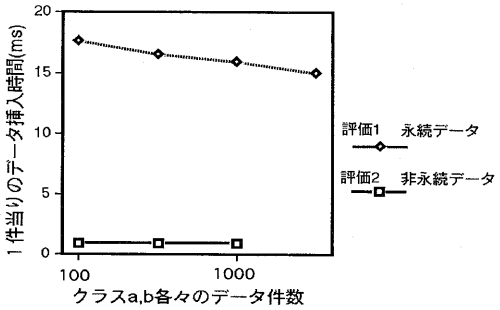


図 8: データ作成時間

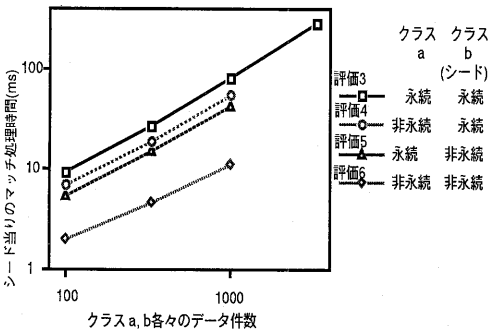


図 9: マッチング処理時間

に格納しているためデータ作成が行えた. このことから, 使用できるメモリ量にかかわらず, 大量のデータを用いて推論が行える事が実証できた.

(2) マッチング処理性能： 前述のように LEAPS は, シードを起点にルール内の残りの CE を入れ子ループによりジョインするため, シード当りのマッチング処理の計算量はルール内 CE 数を c, データ件数を n とすると, $O(n^{c-1})$ である. 評価に用いたルール match_AB の CE 数は 2 であるので, シード当りのマッチング時間はデータ件数に比例する. 評価の結果, 評価 3〜6 のいずれの場合にもほぼデータ件数に比例する処理時間となった. また評価 3 は評価 6 の 1/4.5〜1/7.3 の性能であった. 評価 4,5 は評価 3,6 のほぼ中間の性能であったが, 評価 5 の方が約 20%高速であった. 評価 5 は入れ子ループの内側, 評価 4 は外側が永続データである事からバッファのヒット率の違いによるもとの考えられる.

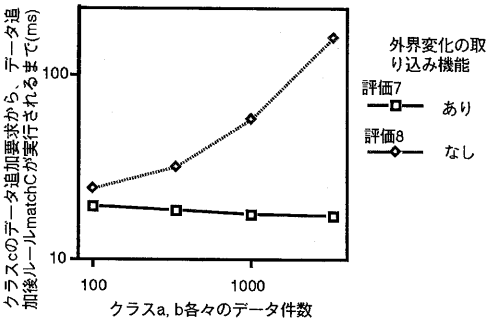


図 10: 外界の変化の取り込み時間

5.3 外界の変化の取り込み性能

クラス a, b が両方とも永続データの場合において, マッチング処理中に外部 I/F プロセスがクラス c のデータの追加要求を受信してから, 推論プロセスがルール match_C を実行するまでの時間を測定した. 外界の変化取り込みを行う場合 (評価 7) と, その機能を停止した場合 (評価 8) の実行時間を図 10に示す. 評価 7 では追加するデータの作成時間とほぼ同等の時間で match_C が実行されており, データ件数 (シード当りのマッチング時間) には殆ど影響していない. 一方評価 8 では, 他のルール起動後に初めて外界の変化が取り込まれるため, ほぼデータ件数に比例して実行時間が延びた. このことから, データ件数にかかわらず外界の変化に対して実時間応答性が実現できていると言える.

5.4 既存汎用 ES シェルとの比較

ここでは, RETE アルゴリズムを使用する代表的な汎用 ES シェルである OPS83 との推論性能の比較を行った. 評価には先に使用した simpleJoin (図 7), tourney(ブリッジの対戦相手組合せプログラム), waltz(配線プログラム) を使用した. 全ての WME を非永続とした場合の実行時間を表 1に示す.

LEAPS は, 最良優先探索により結果的に発火しないインスタンシエーションの探索を行わない事により高速化を図るが, simpleJoin では全てのインスタンシエーションが発火する. これは, LEAPS に最も不利な条件である. simpleJoin での性能差は, 最良優先探索を行うためのオーバヘッド, OPS83 が直接アセンブリ言語にコンパイルするのに対し本 ES シェルは C++へのコード生成を行うことによるオーバヘッド, 等に起因すると考えられる. 生成される

表 1: OPS83 との実行時間の比較

	OPS83	本シェル	相対速度 (OPS83 を 1 とした場合)
simpleJoin	1.3 Sec	7.9 Sec	0.16
tourney	2.7 Sec	4.8 Sec	0.56
waltz	1.2 Sec	0.7 Sec	1.71

C++コードにはまだ最適化の余地があるため, 今後より simpleJoin での性能差を短縮できると考えられる. tourney, waltz のサンプルプログラムでは幅優先探索を行うと実際には発火しないインスタンシエーションが存在する事から, simpleJoin よりも OPS83 に対する性能が向上した. 特に waltz は OPS83 以上の性能を示した. 実際の応用でも, 幅優先探索より LEAPS の方が高速であると報告されている [1] ため本 ES シェルは主記憶のみを使用する場合でも, OPS83 とほぼ同等の性能を持つと言える.

6 おわりに

本論文では, 実時間応答と大量データ処理が可能な ES シェルの実装と評価について報告した. マッチングアルゴリズムとして, 最良優先探索を行う LEAPS を拡張し, マッチング処理中に外界からのデータの変化を割り込みにより受け付けることで実時間応答を実現した. また, ES シェルにデータベース機能を付加して, WME を永続的に格納し (永続データ), その WME と主記憶上の WME を区別無く推論に利用できるようにし, 主記憶量に制限されず大量データを推論できるようにした. 評価の結果, 外界の変化に対してほぼ実時間で応答できる事, ならびにまた主記憶量に係わらず大量のデータを用いて推論できることを確認した. 性能面ではバッファリング等により主記憶上の非永続データに対して 1/4.5〜1/7.3 程度の性能でデータベース上の永続データをマッチング処理できた. また主記憶上の非永続データのみを用いた場合, OPS83 とほぼ同等の性能である事を確認した. 最後に日頃御指導頂く (株)KDD 研究所 村谷所長, ならびに御討論頂いた鈴木副所長に感謝します.

参考文献

[1] D. P. Miranker et. al.:"On the Performance of Lazy Matching in Production Systems," Proc. of Eighth National Conference on Artificial Intelligence, (1990).

[2] C. Forgy: "OPS5 User's Manual", CMU Tech. Report CMU-CS-81-135,(1981).

[3] C. Forgy: "The OPS83 User's Manual Ver. 2.2,"(1986).

[4] C. Forgy et. al.: "RETE: A Fast Match Algorithm for the Many Pattern/ Many Object Pattern Match Problem," Artificial Intelligence, no. 19, pp17-37, (1982).

[5] D. P. Miranker, et. al.: "TREAT: A Better Match Algorithm for AI Production Systems," Proc. of the 1987 National Conf. on Artificial Intelligence, (1987).

[6] 小野 他: "エキスパートシステムのための実時間推論データベースシステムの設計–推論アルゴリズム–," 第 51 回情処全大 3D-10, (1995).

[7] 西山 他 : "拡張可能 DBMS 構築技法に基づく高速 OSI ディレクトリ専用 DBMS の設計と評価," 情処論文誌 Vol.34,No.6,(1993).

[8] David Andrew Brant :"INFERENCING ON LARGE DATA SETS," Ph.D Dissertation, University of Texas at Austin, (1993)