2004年度 卒業論文

Prologを用いた大規模データ検索言語 Verno-SPの実装と評価

提出日: 2005年2月2日
指導: 上田和紀 教授

早稲田大学理工学部 情報学科

三上 啓太
学籍番号：1G00P118–9

概 要

今日、WWW には膨大な情報が溢れている。全文検索システムはそのリソースを有効に活用するための手段であり、Verno もそのような全文検索システムの一つである。

Verno は「プログラマブルな検索エンジン」をコンセプトとしており、検索言語サーバー Verno-SCM が Scheme で書かれたクエリを処理することにより、単純な論理演算を越える検索を可能としている。しかし、クエリの自由度が高い反面、クエリを作成するためにはアルゴリズムの知識が必要であり、またそうして作成したクエリ文字列は一般の検索エンジンのそれと比べて長く読みにくいものとなってしまいがちである。

一方、 Prolog は、アルゴリズムを意識せずに宣言的にプログラムが記述でき、コード長も短いという特長を持つ。しかし、Prolog は WWW 検索のような大規模なデータ運用に用いられた例がほとんどなく、その実用性は未知であった。

そこで、本研究では Prolog の処理系である SICStus の内部 DB および外部 BerkeleyDB を用いて、大規模なデータを処理する場合の実行時間、メモリ効率、ディスク効率を測定した。

また、この測定に基づいて、SICStus 上に Verno 検索を行うモジュール「Verno-SP」の設計と実装を行った。Verno-SP の実装においては、内部 DB 以外にも、BerkeleyDB 及び検索言語サーバー fhandle2 という外部のシステムを利用する。本研究では、これら外部システムのデータも、ユーザーから見るとあたかも内部 DB に格納されているように見えるという、「データの単一ビュー」を実現した。

この結果、Verno-SP を利用しての検索クエリは従来の Verno でのクエリに比べ大幅に簡易化・短縮化され、また実行速度面においても申し分ない性能を示した。

目 次

第 1 章　序論 **3**
1.1 研究の背景 3
1.1.1 WWW と検索エンジン 3
1.1.2 全文検索システム Verno 3
1.1.3 Verno が抱える課題 4
1.1.4 論理型言語 Prolog 4
1.2 研究の目的 4
1.3 本論文の構成 5

第 2 章　全文検索システム Verno **6**

第 3 章　Prolog 処理系 SICStus の性能測定 **7**
3.1 性能測定の目的 7
3.2 測定環境 7
3.2.1 測定に用いるデータ 7
3.2.2 測定に用いるマシン 8
3.3 使用メモリ量および DB 構築時間の比較 9
3.3.1 測定内容 9
3.3.2 測定結果 9
3.3.3 考察 10
3.4 インデクシングの有無による実行速度差 10
3.4.1 測定内容 10
3.4.2 測定結果 11
3.4.3 考察 12
3.4.4 追試とその考察 12
3.5 項目数と実行時間の関係 13
3.5.1 測定内容 13
3.5.2 測定結果 14
3.5.3 考察 14

第 4 章　Verno-SP の設計 **16**
4.1 データの単一ビュー 16
4.2 Verno-SP で実装した述語 17
4.2.1 ページ ID 17
4.2.2 リンク情報 19
4.2.3 タイトル及び URL 情報 19
4.2.4 文字列検索 19
4.2.5 要素構造検索 20

第 5 章　Verno-SP の実装 **22**
5.1 概観 22
5.2 内部 DB を用いる述語の実装 22
5.2.1 URL 情報の抽出 23
5.2.2 リンク情報の抽出 24
5.2.3 link/2 におけるインデクシングに関する工夫 26
5.3 BerkeleyDB を用いる述語の実装 27
5.3.1 xml/2 のデータ規模 27
5.3.2 xml/2 の実装 31
5.4 Socket 通信を行う述語の実装 32
5.4.1 簡易検索言語サーバー fhandle2 の仕様 33
5.4.2 get_index/2 の実装 34
5.4.3 contain/3 の実装 34
5.5 Prolog を用いた単一ビュー環境の構築 35

第 6 章　Verno-SP を利用した検索 **36**
6.1 Prolog の記述性を活かした検索 36
6.1.1 宣言&探索による検索 36
6.1.2 要素構造に対する検索 39
6.1.3 探索的な検索 40
6.2 Scheme との比較 41

第 7 章　まとめと今後の課題 **42**
7.1 まとめ 42
7.2 今後の課題 42
7.2.1 全国版への対応 42
7.2.2 サービスの公開 43

参考文献 **45**

第1章　序論

1.1　研究の背景

1.1.1　WWW と検索エンジン

今日、World Wide Web(以後 WWW と略す) では、個人の日記から企業の製品情報、政府の公報に至るまで、実に多用な情報が公開されている。これらの情報はページを単位として HTML で記述されており、利用者が文書中に埋め込まれたリンクを辿ることにより、ページからページへと移動しながら情報を閲覧できる仕組みになっている。しかし、WWW 上に存在するページの数は膨大であり、有用な情報も多く含まれている反面、非常に混沌としている。

サーバ数 (台)	総ページ数 (万ページ)	総ファイル数 (万ファイル)	総データ量 (ギガバイト)
425,000	8,590	29,173	13,609

表 1.1: jp ドメインの総コンテンツ量
(平成 16 年 7 月発表「WWW コンテンツ統計調査報告書 [1]」より)

こうした状況の WWW から、利用者が必要とする情報を探し出すための支援を行うのが検索エンジンである。検索エンジンは WWW 上のデータを収集、整理してデータベースを構築・保持する。利用者が検索エンジンに対して検索条件を指定すると、検索エンジンはその条件に合致するページをデータベースから引き出し、検索結果として表示する。

1.1.2　全文検索システム Verno

全文検索エンジン Verno は、「プログラマブルな検索エンジン」をコンセプトに作成された。一般的な検索エンジンのインターフェースが、キーワードをスペース等で区切って入力するシンプルな形式であるのに対して、Verno では検索クエリをプログラムで記述することにより、複雑な検索要求を自由に記述することを可能としている。このコンセプトを具現化しているのが Scheme 処理系 SCM を

元にした検索言語 Verno-SCM である。また、簡易検索言語である fhandle を CGI を経由して利用することにより、ブラウザを用いた従来の検索エンジンと同様の検索も行える。さらに、200 年 4 月からは、早稲田大学のサイトに対して、学内約 10 万ページを対象とする検索サービスを提供している。

1.1.3 Verno が抱える課題

Verno は Scheme を用いた検索クエリを処理することにより、従来の検索エンジンと比べて複雑で自由度の高い検索処理を実現している。しかし、検索の自由度が高い反面、検索クエリを作成するためには Scheme の知識だけでなく、アルゴリズムの知識も必要となる。また、そうして作成した検索クエリはどうしても長く読みにくいものになってしまいがちであった。検索の自由度を保ちつつも、クエリ作成の難易度と手間を軽減することが、Verno の抱える課題の一つであった。

1.1.4 論理型言語 Prolog

Scheme は関数型言語 LISP の流れを汲む言語だが、この LISP とともに、非手続き型言語としてよく挙げられるのが論理型言語 Prolog である。本研究では以下の理由から Prolog に着目した。

第一に、Prolog では、モノとモノとの間の関係を宣言することがそのままプログラムになるという、宣言的なプログラミングスタイルが採用されている。この「宣言的な記述」により、プログラマは、あまりアルゴリズムを意識しなくてもプログラムを記述することができる。(勿論、複雑なプログラミングを行うためにはアルゴリズムの知識が必要である。)

第二に、Prolog では定義した fact(事実) を内部データベースに格納する仕組みになっており、Prolog プログラムにはこの内部データベースへの問い合わせの記述であるという性質がある。この性質ゆえ、Prolog はデータベースとの親和性が高い。

第三に、Prolog 処理系はユニフィケーションとバックトラックによる探索機構を有している。この「解を探索する」機能が、検索エンジンの「目的のページを検索する」ことに生かせるのではないかと考えた。

しかし、Prolog は WWW 検索のような大規模なデータ運用に用いられた例がほとんどなく、その実用性は未知であった。

1.2 研究の目的

研究例の少ない、Prolog 上での大規模 WWW データ運用の性能を測定する。

また、Verno の検索言語として Prolog を採用することにより、「プログラマブ

ルな検索」のクエリとしてのプログラムを、「より分かりやすく」、「より書きやすく」することを目指す。このPrologベースの検索言語のデザイン・実装を行うにあたり、処理系の能力の活用と、実際に検索を行う際の利便性を考慮する。

1.3 本論文の構成

第2章　全文検索システムVerno

第3章　Prolog処理系SICStusの性能測定

3.1　性能測定の目的

第一の目標は、大規模なデータ運用における Prolog の性能測定である。従来の Prolog の研究では、比較的小規模なデータベースを対象に高度な処理を行っている場合が多く、Verno で運用するような大規模なデータに対して、どこまでの性能を発揮できるかは未知であった。そこで本研究では、Prolog を用いた検索言語の実装に着手する前に、入力するデータの規模に対して、Prolog が使用するメモリサイズ、そしてデータベースに対するアクセスに要する時間を測定する。

第二の目標は、効率的なデータ運用法の模索である。本研究で Prolog を採用した一番の理由は、Prolog の記述性が高いことであるが、実際に WWW を対象とした検索システムを実装する場合、その処理に要する時間を短く押さえる事が前提条件となる。Verno の持つデータ群を Prolog 処理系に取り込む場合、どのような形式を取るのが良いかを検討するために、以下の 3 つのデータ保持形式を比較、検討する。

- 通常のデータ読込 (consult)
- 最適化と内部コードへの変換を行うコンパイル (compile)
- 外部 DB を用いてディスク上にデータを保持する (BerkeleyDB モジュール)

3.2　測定環境

3.2.1　測定に用いるデータ

測定に用いるテストデータは、ランダムグラフサーバ [2] を用いて作成した。ランダムグラフサーバは二村研究室で開発された、アルゴリズム性能評価のためのグラフデータを作成するシステムである。今回は頂点数を 10 万、辺数を 10 万、20 万、40 万、80 万、160 万、320 万に設定し、6 つのテストデータを作成した。

加工前のデータの形式

```
89689 65449 0.7991573316095805
65449 67980 0.0899282728152907
67980 53765 0.3886090118429538
53765 46389 0.7682428122876312
...
```

加工後のデータの形式

```
edge(89689, 65449, 0.7991573316095805).
edge(65449, 67980, 0.0899282728152907).
edge(67980, 53765, 0.3886090118429538).
edge(53765, 46389, 0.7682428122876312).
...
```

consult、compile、BerkeleyDB のいずれの形式においても、Prolog はこのデータの 1 行を 1 つの項として読み込む。3 引数の述語 edge は第一引数と第二引数が区間 [0,100000) のランダムな整数、第三引数が区間 (0,1) のランダムな小数となっている。

3.2.2 測定に用いるマシン

測定に用いたマシンのスペックを表 3.1 に示す。このマシンに SICStus3.12.0 をインストールして測定を行った。

マシン名	marc
CPU	AMD Opteron240(1.4GHz) x2
Memory	4.0GB(Dual channel: 1GB x4)
SWAP	8.0GB
HDD	System:S-ATA 80GB Data:S-ATA 1TB(250GB x4)
OS	FedoraCore2

表 3.1: 測定に用いたマシンのスペック

3.3 使用メモリ量およびDB構築時間の比較

3.3.1 測定内容

内部DBへの通常読込、内部DBへのコンパイル読込、および外部BerkeleyDBに対して、前述のテストデータを格納し、その際のメモリ使用量 (BerkeleyDBの場合はディスク使用量) と、それに要する時間を測定した。BerkeleyDBはSICStusのモジュールを通じて利用することができる。今回はテストデータを述語 `db_store/3` により1項ずつ全て格納し、その時間を測定した。

3.3.2 測定結果

データベースサイズの測定結果は表3.2、構築時間の測定結果は表3.3のようになった。

項数 (万)	text	consult	compile	BerkeleyDB
10	3.7	18	20	17
20	7.3	34	37	38
40	15	68	74	71
80	30	130	143	132
160	59	258	284	245
320	117	516	568	451

表 3.2: データベースサイズ (MB) の比較

項数 (万)	consult	compile	BerkeleyDB
10	4.5	13.2	13.0
20	9.2	26.6	27.0
40	18.8	54.0	61.2
80	38.8	109.2	184.5
160	82.1	226.7	782.8
320	185.0	490.2	4116.3

表 3.3: データベース構築に要する時間 (秒) の比較

3.3.3 考察

データベースサイズに関して

いずれの形式においても、データベースサイズは項数に綺麗に比例した。従って、ある程度の数の項でのデータサイズを調べれば、その数値を元に項数を大きくした場合のデータサイズも予測できる。また、consult と compile のデータサイズが殆ど同じである点も注目に値する。この程度の差であれば、容量の制約から compile を諦める必要は生まれない。

構築時間に関して

consult と compile はともに項数に比例する値を示し、compile は consult に比して、約 2.8 倍の時間を要する。対して、BerkeleyDB は 40 万項を境として構築に要する時間が急速に増加している。構築時間が長くても、一度データベースを構築してしまえば、それ以降の処理には影響しない。従って実現可能な時間で終了すれば良いのだが、BerkeleyDB は結果を見ると、1000 万項程度が限界となるかもしれない。

3.4 インデクシングの有無による実行速度差

3.4.1 測定内容

SICStus のマニュアル [5] によると、SICStus の内部 DB は項の第一引数もしくはその関数記号に対してハッシュテーブルによるインデクシングを行う。このインデクシングにより、同じ項に対するアクセスであっても、それがどの引数に対するアクセスであるかによって、速度が大幅に変わると予測される。この測定では、consult 及び compile により内部 DB に格納されたデータに対し、先頭から 1 万項に対してのアクセスに要する時間を測定した。

第一引数に対する参照を 1 万回行う

```
test1(Time):-
        findall(A,edge(A,B,C),List1),
        cut(List1,10000,List2),
        time(t1(List2),Time).

t1([]).
t1([H|T]):-
        edge(H,B,C),
        test1(T),!.
```

上記のプログラム中、cut/3 及び time/2 は自分で定義した述語である。cut(L1,N,L2)

はL1を先頭からN要素取り出したリストをL2として返す。time(P,Time)は述語Pを実行するのに要した時間(ms)をTimeとして返す。全体としてこのプログラムは、要素数1万のリストを先頭から順に取り出し、その整数を第一引数として持つ述語edge/3を内部DBに問い合わせる。第二引数と第三引数に対しても同様のプログラムを実行し、その実行速度を測定した。

3.4.2 測定結果

通常のデータ読込(consult)に対しての測定結果を表3.4、コンパイルを行っての読込(compile)に対しての測定結果を表3.5に示した。また、後から生じた疑問に対する追試の結果も表3.4の下部に追記した。この追試に関しては考察の後に節を設けて後述する。

項数(万)	第一引数	第二引数	第三引数
10	20	14100	16340
20	20	13580	16980
40	20	13750	16730
80	20	12770	15840
160	20	12710	15930
320	20	12630	15850
10'	20	15120	14840
20'	20	15180	14180

表3.4: consultにより格納したデータの各引数に対するアクセスに要する時間(ms)

項数(万)	第一引数	第二引数	第三引数
10	10	8570	10090
20	10	7980	9890
40	10	8320	10180
80	10	7410	9740
160	10	7750	9760
320	10	7510	9640

表3.5: compileにより格納したデータの各引数に対するアクセスに要する時間(ms)

3.4.3 考察

まずは表 3.4 と表 3.5 を見比べてみる。これらの差は、格納データをコンパイルしたか否かの違いのみである。結果として、コンパイルしたデータは全ての項目において 40%近い速度の向上が見られる。[1]しかし、表 3.5(compile されたデータに対する測定結果) 内の数値同士の関係は表 3.4 のそれとほぼ同じである。このことから、今回のような単純な構造のデータに対してコンパイルを行った場合、基本的な処理速度は一定の割合で増加するものの、データベースとしての特性はコンパイルを行わない通常読込の場合と変わらないことがわかる。

次に、第一引数と第二、第三引数に対するアクセス時間を見比べると、その差は歴然である。1 万件に対してアクセスしても 0.1 秒以下である第一引数に対し、第二、第三引数では約 15 秒もの時間を要している。実用に際して、第一引数以外の引数を指定しての大量アクセスは、なるべく避けなくてならない。

3.4.4 追試とその考察

第二引数と第三引数の間にも、若干の差がみられる。この差がその位置によるものなのか、それともその桁数によるものなのかを確認するため、以下のように内部 DB に格納するデータの第二引数と第三引数を入れ替えて追試を行った。

当初のデータ形式

```
edge(89689, 65449, 0.7991573316095805).
edge(65449, 67980, 0.0899282728152907).
edge(67980, 53765, 0.3886090118429538).
edge(53765, 46389, 0.7682428122876312).
...
```

追試のデータ形式

```
edge(89689, 0.7991573316095805, 65449).
edge(65449, 0.0899282728152907, 67980).
edge(67980, 0.3886090118429538, 53765).
edge(53765, 0.7682428122876312, 46389).
...
```

この結果が、表 3.4 の下部の 10' と 20' である。比較しやすいように、配置を変えて再掲する。

表 3.6 では表の配置を整理し、5 桁の整数に対するアクセスと 16 桁の小数に対するアクセスを分けて並べた。この表を見ると、データの桁数の増加による時間

[1]第一引数のデータに関しては、測定の最低単位が 10ms であるため、この計算には含めないこととした。

項数 (万)	第二引数	第三引数
10(整数)	14100	14840
20(整数)	13580	14180
10(小数)	15120	16340
20(小数)	15180	16980

表 3.6: 第二、第三引数に配置された整数及び小数に対するアクセスに要する時間 (ms)

の増加と、第二引数か第三引数かによる時間の増加の両方が見られる。共にインデクシングがなされていない引数同士の比較でも、若い引数に対するアクセスの方が僅かに短い時間で済むことが分かった。しかしこれらの差は共に非常に僅かであり、運用上配慮する必要はなさそうである。

最後に着目するのは、表 3.4 と表 3.5 において、共に項数が増えても実行時間が増加していない点である。しかし、これはプログラム上の問題であった。内部 DB に格納されている項のうち、先頭 1 万項へのアクセスを行うプログラムとなっているため、全体の項目数が増加しても実行時間が増加しなかったと考えられる。

3.5 項目数と実行時間の関係

3.5.1 測定内容

前節の終わりで述べたように、前節の測定では項目数の増加が実行時間に及ぼす影響を知ることが出来ない。そこで、この節では以下のようなプログラムを使用して、実行時間の測定を行った。この述語 rand_test/1 を、「`|?- rand_test(10000).`」として実行する。

ランダムな項の第一引数に対するアクセスを 1 万回行う

```
:- use_module(library(random)).
rand_test(0).
rand_test(N):-
        random(0,100000,Random),
        edge(Random,A,B),
        N1 is N-1,
        rand_test(N1),!.
rand_test(N):-
        N1 is N-1,
        rand_test(N1),!.
```

まず述語 random/3 で区間 [0,100000) のランダムな整数を生成する。そして第一引数にその整数を持つ項に対してアクセスを行う。該当する項が存在しない場合でも、無視してカウントを進め、次の項にアクセスする。

前節までの実験で、consult と compile のデータは性質的には同じであることが分かったので、この節の測定では consult と BerkeleyDB の比較のみを行った。

3.5.2 測定結果

測定結果を表 3.7 に示す。

項数 (万)	形式	第一引数	第二引数
10	consult	60	189300
20	consult	90	305330
40	consult	100	355980
80	consult	100	642490
160	consult	100	776520
320	consult	100	901120
10	Berkeley	2710	2820
20	Berkeley	1460	3520
40	Berkeley	2520	2490
80	Berkeley	2290	2340
160	Berkeley	3730	3760
320	Berkeley	4830	5160

表 3.7: 第一、第二引数に対するランダムな 1 万回のアクセスに要する時間 (ms)

3.5.3 考察

通常の内部 DB に関して

前節での測定と同様、インデクシングの有無の差がはっきりと現れている。前節での測定結果と異なる点は、第二引数に対するアクセスに要する時間が、全体の項数の増加にともなって、対数的に増加している点である。これをグラフにしたのが図 3.1 である。

最も項数の多い 320 万項のデータでは、1 万項へのアクセスに 15 分間も掛かってしまっており、格納するデータ数が増えれば増えるほど、「インデクシングが行われていない項への大量アクセスを避ける」ための工夫が必要であると言えそうだ。

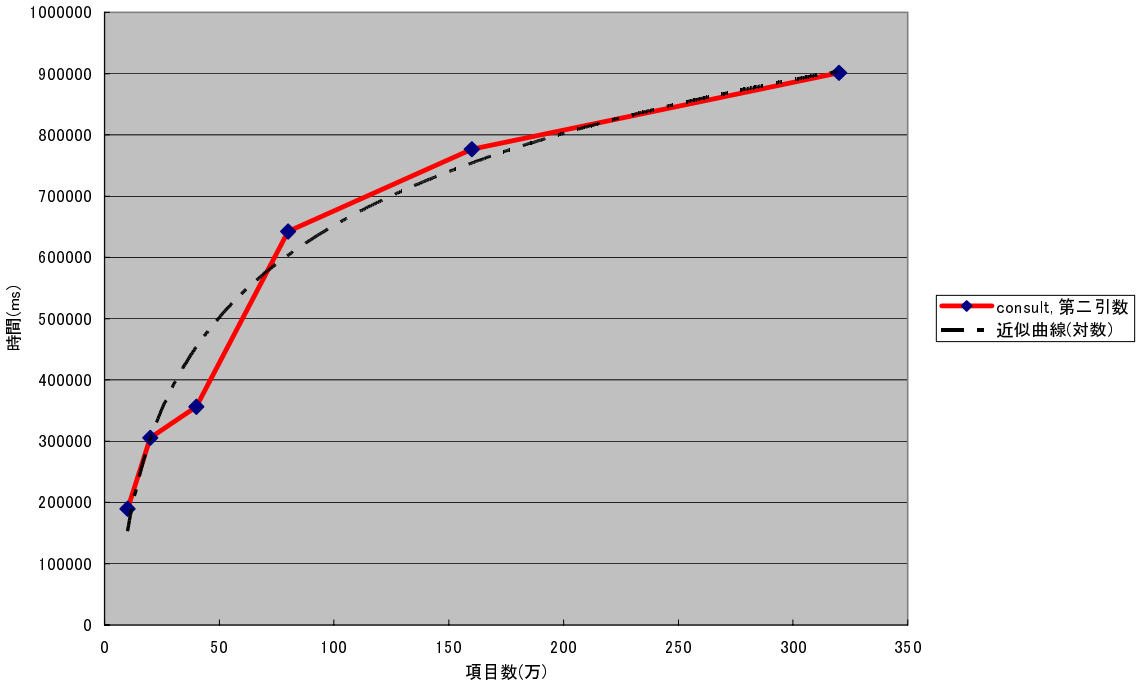

図 3.1: インデクシングされていない項に対するランダムな 1 万回のアクセス

外部 BerkeleyDB に関して

BerkeleyDB は構築の際に明示的にインデクシングを指定することができる。今回作成した DB では第一、第二引数ともにインデクシングを指定しており、測定結果でも第一引数と第二引数での結果がほぼ同じになっている。しかし、BerkeleyDB を用いる場合の測定結果は、実行する毎に結果が変動し、なかなか数値が安定しなかった。これは BerkeleyDB 独自の仕組みの他に、OS によるディスクアクセスのキャッシング等が行われたためだと考えられる。

BerkeleyDB の示した性能は、内部 DB の第一引数に比べると数十倍の時間が掛かってしまっているが、それでもインデクシングのない内部 DB の第二引数に比べると十分高性能である。

第4章　Verno-SPの設計

4.1　データの単一ビュー

Verno-SP の設計にあたり最も重視した方針が「データの単一ビュー」である。「データの単一ビュー」とは、ユーザーが複数のデータを、その格納形式や実装を意識せずに利用できる形態を指す。Verno-SP では、Prolog の内部 DB だけでなく、外部 BerkeleyDB 及び、Verno の擁する簡易検索言語サーバー fhandle2 との通信を通じて Verno の持つデータ群とアクセスを行う。しかし、ユーザーがこれらを利用する際に、一々どのシステムにアクセスするかを意識して、明示的にクエリを作成しなくてはならないのでは、せっかくの Prolog の記述性を損なってしまう。そこで、Verno-SP の実装では、どのデータに対するアクセスであっても、あたかも内部 DB へのアクセスであるかのように記述できる、という「データの単一ビュー」を目標とした。

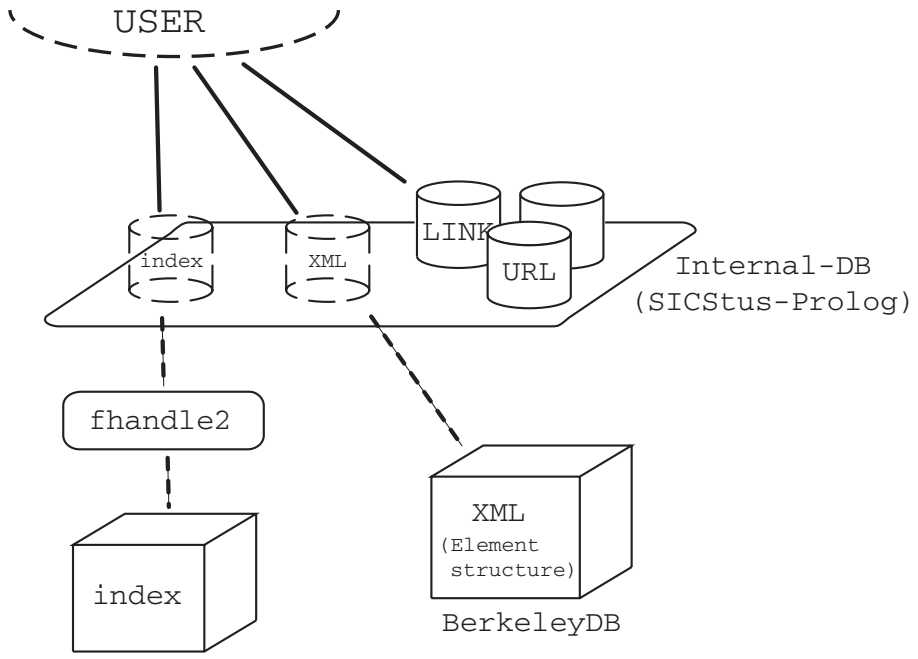

図 4.1: Verno-SP におけるデータの単一ビュー
外部にある index や XML も、内部 DB 上に存在するように見える

4.2 Verno-SP で実装した述語

実際の実装についての解説は 5 章に譲るが、本節では Verno-SP において実装した述語の用法 (つまり Verno-SP のユーザーが知るべき知識) に絞って解説を行う。

まず、使用可能な述語一覧を表 4.1 に示す。表の読み方で特殊なのは、述語の引数の左に付記されている「?」、「+」、「−」の 3 種の記号である。Prolog では述語を呼び出す際に、その引数に特定の数を表すような変数[1]を指定して呼び出すだけでなく、具現化されていない変数を指定することも可能であり、こういった具現化されていない変数は Prolog が処理を行う間に、処理系による推論を経て、条件を満たすような項によって具現化される。これを表すのが前述の 3 種の記号であり、「+」は具現化した状態で述語を呼び出す必要がある引数、「−」は具現化されていない返値としての引数、「?」はそのどちらでも良い引数に付記される。Prolog を宣言的に記述する場合、理想的なのは、引数が具現化していてもしていなくても宣言することのできる「?」の述語であり、本研究で定義した述語は、手続き的な動作をする get_index/2 を除いて、全ての述語がこれに該当する。

4.2.1 ページ ID

Verno-SP では、ページを扱う際には、単純に 1 ページに対して 1 つ割り振られた「ページ ID」と呼ばれる整数を用いる。Verno には以前から「ページ ID」という概念が存在していたが、これはあくまで内部で用いられる概念であり、ページを扱う際の基本となる単位は「URL 型」と呼ばれる、ページを表す型を持つ変数であった。今回「URL 型」を採用しなかったのは以下の二つの理由からである。

第一に、Prolog には Scheme に存在するような型の概念が存在しないこと。(integer 等は存在しているが、明示的に独自の型を定義することはできない)

第二に、「URL 型」というページを表す単位を用いると、内部 DB によるインデクシングを生かせない状況が増えてしまうこと。例えば、単純な整数のページ ID を用いる代わりに、`url(PageID,POS,PageRank)` といった構造を持った項を用いることも可能であるが、この場合、この項を第一引数に持つ述語に対する内部 DB によるインデクシングは、「url」という関数子に対してのみ行われ、真に重要である「PageID」に対してはインデクシングが行われないことになってしまう。

第一引数 PageID がインデクシングされる

```
title(PageID, Title).
```

関数子 url がインデクシングされ、PageID はインデクシングされない

```
title(url(PageID,POS,PageRank), Title).
```

[1] 一般的に「具現化された変数」と呼ぶ。本論文でも以下そう呼ぶこととする。

関数名	内容
link(?*int PageID1*, ?*int PageID2*).	*PageID1* から *PageID2* へのリンクが存在することを表す。Verno-SP ではページを、ページを表す整数 PageID で扱う。
linkednum(?*int PageID*, ?*int N*).	*N* は、*PageID* の被リンク数を表す。
title(?*int PageID*, ?*atm Title*).	*Title* は *PageID* のタイトル文字列を Prolog の atom に変換したもの。
url(?*int PageID*, ?*atm URL*).	*URL* は *PageID* の URL を表す atom。
urllist(?*int PageID*, ?*list URLList*).	*URLList* は、URL の持つドメインを、順に格納したリスト。例えば、*PageID* の URL が`'http://www.waseda.jp/~mikami/a.html'` である場合には、*URLList* が表すリストは `['www.waseda.jp','~mikami','a.html']` となる。
get_index(+*atm Key*, −*list Hit*).	fhandle2 に対して Socket 通信を行い、*Key* を検索文字列として送信し、その検索結果をリスト *Hit* として返す。使用は推奨されず、通常は下記の contain/3 を用いる。
contain(?*int P*, ?*atm Key*, ?*int N*).	ページ *P* が、文字列 *Key* を *N* 個含む事を表す述語。(内部では上記の get_index/2 を呼んでいる)
xml(?*int PageID*, ?*doc XML*).	*XML* は、*PageID* が表すページの内容を、SICStus の xml モジュールを用いて、Document-Value モデル形式に変換したもの。Document-Value モデルは、XML 文書を要素構造を保持したまま Prolog で読み込める形式に変換したものである。

表 4.1: Verno-SP で使用することが出来る述語

4.2.2 リンク情報

リンクを扱う情報としては、link/2 及び linednum/2 を定義した。この述語を用いる場合、何も工夫しない実装では、3 章で示した結果の通り、第一引数を指定してのアクセスと第二引数を指定してのアクセスで大きな性能差が生まれてしまう。しかし、5 章で後述する工夫により、この述語 link/2 は、どちらの引数に対するアクセスでもインデクシングによる恩恵を得られる。

link/2 は、一つのページ内に同じページに対するリンクが複数あっても、そのリンクを二重にカウントすることはなく、一つのリンクと見なす。

また、自ページから自ページへのリンクは link(4649, 4649) の様にして記録されているが、被リンク数を表す linednum/2 には自らに対するリンクは含まれない。

4.2.3 タイトル及び URL 情報

Web ページ内の TITLE タグに挟まれた要素であるタイトルの情報を、title/2 で表した。

HTML 中に含まれるタイトル情報

```
<html>
<head>
<title>ここに書かれているのがタイトル情報です</title>
</head>
<body bgcolor="#000000" text="#FFFFFF">
<h1>見出し</h1>
ただいま工事中
<a href="./index.html">戻る</a>
</body>
</html>
```

url/2 及び urllist/2 は共に、ページの URL を表す述語だが、URL による階層構造を生かしたプログラムの書きやすさを意識して定義したのが urllist/2 である。

4.2.4 文字列検索

文字列による検索は、Verno が擁する fhandle2 との Socket 通信によって行う。get_index/2 は fhandle2 の仕様に素直に実装した述語であり、そのためあまり Prolog 的でない、手続き的な述語となっている。

一方、contain/3 は、内部では get_index/2 を呼び出しているものの、Prolog らしい宣言的な記述性を念頭に置いた述語である。実装に当たっての工夫は 5 章で述べる。

4.2.5 要素構造検索

SICStus の xml モジュールとしても採用されている、xml.pl[3] により提案されている、xml を Prolog 内部に取り込める形に変換した形式が、Document-Value モデルである。前ページで示した HTML をこの Document-Value 形式に変換すると、以下の様になる。

Document-Value モデルの例

```
xml( [version="1.0"],
  [
  namespace( 'http://www.w3.org/1999/xhtml', "",
    element( html, [],
      [
      element( head, [],
        [
        element( title, [],
          [
          pcdata("ここに書かれているのがタイトル情報です")
          ] )
        ] ),
      element( body,
        [bgcolor="#000000", text="#FFFFFF"],
        [
        element( h1, [],
          [
          pcdata("見出し")
          ] ),
        pcdata("ただいま工事中"),
        element( a,
          [href="./index.html"],
          [
          pcdata("戻る")
          ] ),
        ] )
      ] )
    )
  ]
).
```

Document-Value 形式の項を扱うには、主に xml モジュール [3] 内で定義されて

いる述語を利用する。しかし勿論、独自にこの形式を対象としたプログラムを作成することも可能である。[2]

この要素構造データは、容量的な制約から、実装に辺り BerkeleyDB を用いている。そのため、「存在する全てのページに対する Document-Value 形式のデータを取り出し、操作を加える」といった利用を行うと、かなり操作に時間が掛かってしまう。従って、主に想定するのは「他の操作によってある程度ページを絞り込み、その対象に対してさらに詳しい情報を得たり、操作を行う」という用途である。

[2] この章で述べた述語群の実際の利用に関しては、6 章にて述べる。

第5章　Verno-SPの実装

5.1　概観

本章では、前章までに示したVerno-SPの実装の詳細を示す。Verno-SPに実装を行った述語は大まかに分類すると、以下の3種類に分類することができる。

1. 内部DBに格納された情報を利用する述語
   link/2, linkednum/2, title/2, url/2, urllist/2
2. BerkeleyDBに格納された情報を利用する述語
   xml/2
3. fhandle2とSocket通信を行い情報を取得する述語
   get_index/2, contain/3

次節以降では、上記の分類毎に、各述語の実装を解説する。

5.2　内部DBを用いる述語の実装

本研究では、内部DB以外にもデータを持つ形式を取っているが、それでも最も重要なのは内部DBの利用だと考える。Prologという言語とその処理系が、内部DBへの問い合わせを核として構築されており、内部DBを利用することでPrologの能力を最大限に生かすことができるためである。

3章での測定により、コンパイルを行う場合のデメリットはその構築に時間が掛かることのみであることが判明した。また、このコンパイルは一度実行して、述語save_files/2により保存を行えば、それ以降は述語load_files/1を用いることにより、短時間でにメモリ上に読み込むことが可能である。よって、構築に要する時間が増えることは、Verno-SPの実装においては問題とならない。そこで、Verno-SPの実装を行うにあたり、内部DBに格納するデータに対しては、全てコンパイルを行うこととした。

この節で扱う述語はいずれも、内部DBに読み込むためのデータファイルを作成し、そのデータをコンパイルして保存することで実装されている。以下では、述語ごとに、このデータファイルの作成方法を解説する。

5.2.1 URL 情報の抽出

2 章で触れたように、Verno が用いる収集ロボット larbin[4] は、収集したページの HTML2000 ファイル毎に一つのディレクトリを作成し[1]、ディレクトリ内に、ファイル「f00000」〜「f01999」とファイル「index」を作成する。「fxxxxx」は 2000 個の HTML をそのまま改名して保存したものであり、「index」は 2000 個の HTML とその URL との対応を記録したファイルである。

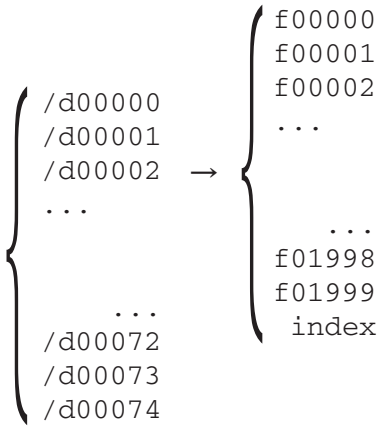

図 5.1: larbin 収集データのディレクトリ構造 (学内 14 万件対象の場合)

index ファイルの書式

```
   0 http://www.waseda.ac.jp/
   1 http://www.waseda.jp/top/index-j.html
   2 http://www.wiaps.waseda.ac.jp/default.asp
   3
                    〜中略〜
1998 http://denryoku.narita.elec.waseda.ac.jp/menu.html
1999 http://www.fuka.info.waseda.ac.jp/~koseki/index.html
```

Verno におけるページ ID の割り振りは、下記の式により、一意に決定される。

PageID の割り振り規則

$$\text{PageID} = \text{ディレクトリ番号} \times 2000 + \text{ファイル番号}$$

URL 情報抽出のアルゴリズムは極単純であり、全ディレクトリの index ファイルを順に開きながら、上記の式を適用して URL に対応する PageID を割り振るだけである。

[1] URL 構造をそのままディレクトリ構造に反映させる保存モードも存在する。

5.2.2 リンク情報の抽出

2 章でも述べたように、Verno は構築の際にタグ情報をファイルとして保存している。リンク情報のデータファイルは、このタグファイルを解析することにより作成される。

タグファイルの例 (一部)

```
4:2354:br /
4:2361:table cellspacing="2" cellpadding="2" width="115"
4:2424:tr bgcolor="#000055"
4:2447:td
4:2451:a href="http://www.waseda.ac.jp/index-j.html"
4:2545:font color="#FFFFFF"
4:2578:/font
4:2585:/a
4:2589:/td
4:2595:/tr
4:2601:/table
4:2610:br /
4:2617:table cellspacing="2" cellpadding="2" width="115"
```

タグファイルは上記の様な行が羅列されたテキストファイルであり、下記の様な書式になっている。

タグファイルの書式

PageID : 出現バイト位置 : タグ ␣ 要素 1 ␣ 要素 2...

上記の例は PageID=4 の文書の情報の一部であり、バイト位置 2354 から 2617 にかけて br タグ、table タグ、tr タグ、td タグ、a タグ、font タグとその要素が記録されているのが読み取れる。この中で link 情報を表しているのは、a タグの href 要素である。

link 情報抽出のアルゴリズムは永澤氏の卒業論文 [6] に基づいているが、前述の URL データファイルの作成と併せて、その大まかな流れを図 5.2 に示す。

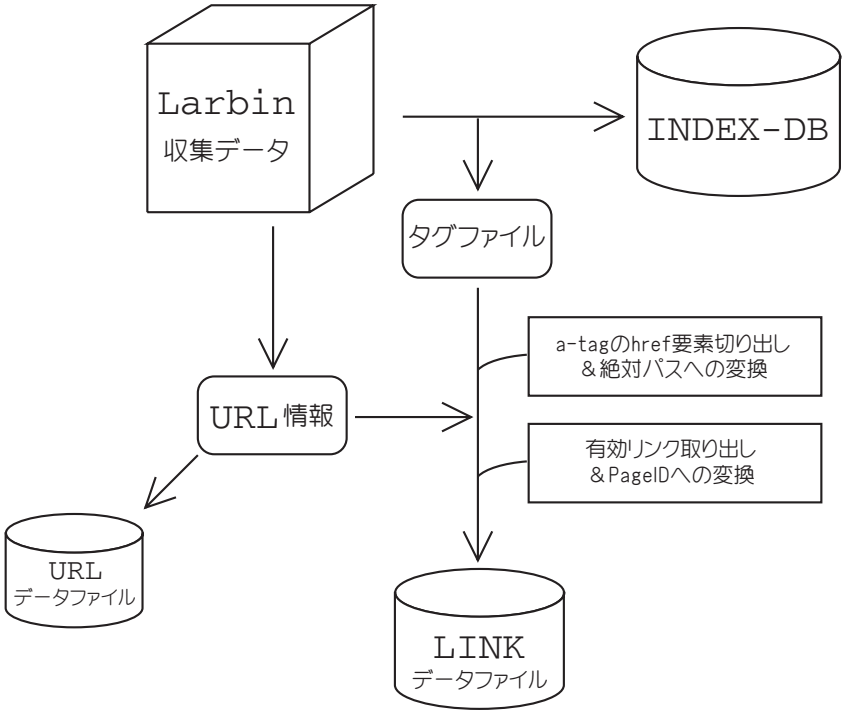

図 5.2: URL 情報、リンク情報の抽出およびデータファイルの作成

本研究では、リンク情報ファイル及び、URL 情報ファイルを作成する java プログラム「LDBMake」を作成した。LDBMake はまず前節で述べたアルゴリズムで URL 情報を抽出し、URL データファイルに書き出す。その後、タグファイルを 1 行目から読込みながら以下の処理を行う。

1. a タグを探し、その href 要素を取り出す
2. リンク先が相対パスで指定されている場合、
   PageID から URL を調べ、それを元に絶対パスに復元する
3. リンク先の絶対パスが URL 情報に含まれるかを判定し、
   既知リンクと未知リンクに分類する
4. 既知リンクを PageID に変換する
5. リンク情報ファイルの書式に出力する

URL 情報ファイルは、こうして完成したデータファイルを SICStus 上でコンパイルすることで完成する。

5.2.3 link/2 におけるインデクシングに関する工夫

4 章でも触れたが、ここでは link/2 の実装にあたって行った工夫に関して述べる。

Prolog では内部 DB に格納された項の第一引数にのみインデクシングが行われるため、3 章での測定の通り、第一引数とそれ以外の引数では、そのアクセスに掛かる時間に大きな差が生まれる。第二引数を指定してのアクセスがあまり行われないと考えられる述語に関しては、インデクシングが行われないことは殆ど問題にならない。しかし、ここで実装する link/2 においては、第二引数によるアクセスを利用する機会が多いと考えられる。(「あるページからリンクされているページ」という使い方だけでなく、「あるページにリンクしているページ」という調べ方も有効だと考えられるため)

この、第二引数にインデクシングが行われないという問題を解消するため、Verno-SP の link/2 では以下のような実装を行った。

link/2 の実装

```
link(A,B):-
          ground(A),
          verno_link(A,B).
link(A,B):-
          \+(ground(A)),
          verno_linked(B,A).
verno_link(a1,b1).
verno_link(a2,b2).
...
verno_linked(b1,a1).
verno_linked(b2,a2).
...
```

述語 ground/1 は、変数が具現化されているか否かを判定する組み込み述語であり、link/2 はその第一引数が具現化されているかを判定し、異なる内部述語にアクセスする。verno_link/2 は通常通りのリンク情報であり、verno_linked/2 はその第一引数と第二引数を入れ替えたものである。(図 5.3)

こうすることにより、格納するリンクデータ量は 2 倍になるものの、第二引数に対するアクセスにおいてもインデクシングの効果を得られる。また、Verno-SP 全体を一つのモジュールとし、内部用の述語は隠蔽することにより、ユーザーはこの内部の仕組みを意識することなく、述語 link/2 がそのまま内部 DB に格納されている場合と全く同様に利用することができる。(データの単一ビュー)

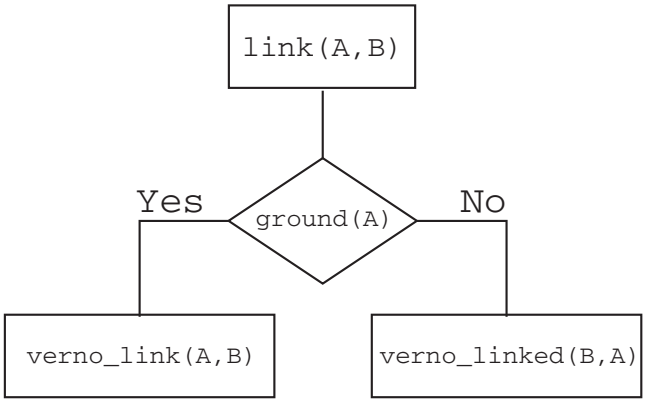

図 5.3: link/2 の実装
どの引数へのアクセスかにより、内部で異なるデータを呼ぶ

第二引数にも多数のアクセスを行う例として被リンク数の算出を行う述語を定義し、第二引数をインデクシングする工夫を行わない場合と、行った場合での実行時間の比較を行った。この結果を表 5.1 に示す。

通常実装	192160
工夫した実装	30

表 5.1: 1000 ページの被リンク数の算出に要した時間 (ms)

とても極端な例なので、実際の運用での効果はこれほど劇的ではないだろうが、しかしインデクシングの有無はこれだけの性能差を生む可能性がある。

5.3 BerkeleyDB を用いる述語の実装

これに該当する述語は唯一 xml/2 のみである。3 章における BerkeleyDB の性能測定結果は、内部 DB に比べると大きく劣っていた。しかし、この述語 xml/2 は対象とするデータの合計が 2.8G と非常に大きく、これをそのまま内部 DB に格納することは不可能であるため、外部 BerkeleyDB を用いての実装を行った。

5.3.1 xml/2 のデータ規模

述語 xml/2 で扱うデータの具体的な作成手順については後述するが、ここではまず、text 形式でのデータ量および、内部 DB 格納時のメモリ使用量、そして BerkeleyDB 格納時のディスク使用量を表 5.2 に示す。

件数 (万)	text 容量	内部 DB	BerkeleyDB
1	162	708	87
2	335	1466	177
4	702	3084	364
6	1194	4987	589
8	1604	6695	790
10	1952	8162	971
12	2273	9514	1162
15	2844	11898	1500

表 5.2: 要素構造データにおける、項数と DB サイズ (MB) の関係

BerkeleyDB にした場合の容量が、text 容量の約半分に減少しているが、これは、xml データが文字列を文字コードの羅列として保存しているためである。例えば「上田研究室」という文字列は、text 形式の xml データでは「[51091,99,6308,5766,314]」という文字コードのリストとして記述されており、バイナリで保存されている BerkeleyDB に比して容量が大きくなってしまっていると考えられる。

この測定結果の中で一番問題なのは内部 DB に格納したときのメモリ使用量である。1 万件を格納した時点で既に 700MB ものメモリを消費しており、4 万件と 6 万件の間でマシンのメモリを使い果たし、スワップ領域を使用し始めている。最終的な容量は 11G にも上った。容量だけから見ても、これは現実的とは言えない値である。

次に、実際にこのデータを使用するのに、どの程度の時間が掛かるのかを測定した結果を表 5.3 及び、図 5.4、5.5 に示す。

	処理 1		処理 2	
件数 (万)	内部 DB	BerkeleyDB	内部 DB	BerkeleyDB
1	670	10280	860	23010
2	1390	40040	850	22930
4	2920	82210	850	22700
6	390160	132710	61140	23030
8	583310	199390	67660	23190
10	800020	226700	59030	25150
12	1007350	251420	60790	26500
15	1463500	305070	63760	29320

表 5.3: 要素構造データにおける、項数と処理速度の関係

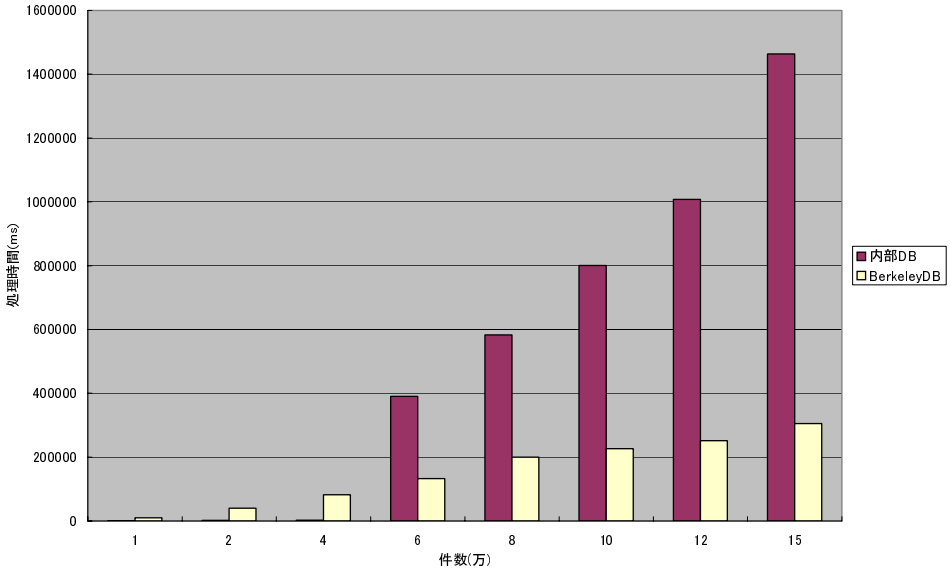

図 5.4: 要素構造データの利用における、内部 DB と BerkeleyDB の処理速度比較 (1)

「処理 1」は以下のようなクエリである。

処理 1

```
findall(N,xml(N,malformed(_,_)),Results).
```

述語 xml/2 で内部 DB に問い合わせ、第二引数に格納されている項が「malformed(_,_)」であるもの[2]を全て見つけて、その第一引数 N をリスト Results に格納する。この処理は格納されている全ての項を一通り走査する必要があるので、

BerkeleyDB の示した結果を見ると、こちらは完全に比例関係を示しており、大量のデータを格納していても安定した速度で実行出来ていることがわかる。一方、内部 DB が示した結果では、マシンのメモリを使い果たした 4 万〜6 万項の間で大きく実行速度が変化している。当然の事ではあるが、スワップアウトが発生してしまうと、全体の性能が大幅に低下してしまうことが示された。

[2]後述する xml パーサが、何らかの理由でパースに失敗した場合に生成される項。

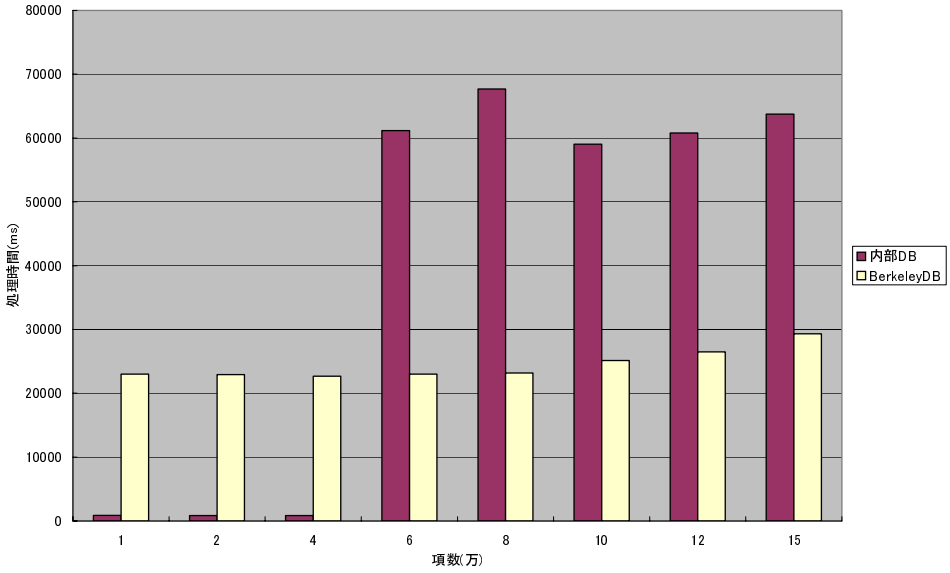

図 5.5: 要素構造データの利用における、内部 DB と BerkeleyDB の処理速度比較 (2)

このスワップアウトによる、純粋に単位処理あたりの性能低下がどの程度なのかを調べるため、さらに以下の処理 2 に要する時間を測定した。

処理 2

```
test(9999).
test(N):-
     xml(N,malformed(_,_)),!,
     N1 is N+1,test(N1).
test(N):-!,
     N1 is N+1,test(N1).
```

処理 2 は処理 1 とは異なり、DB 内の先頭 10000 にのみアクセスする。測定結果は予想通り、BerkeleyDB の測定結果がほぼ一定値、内部 DB の結果がスワップアウトの前後で大きく異なる一定値となった。

内部 DB は処理 1、処理 2 共にスワップアウト後の性能低下が著しく、Verno-SP への実装においては、BerkeleyDB を採用することとした。

5.3.2 xml/2 の実装

まず、以下に xml/2 用のデータの作成及び、そのデータを BerkeleyDB に格納する手順を示す。

1. TIDY[12] により、larbin 収集 HTML を XML に変換する
2. SICStus 上にて xml モジュールに含まれる述語 xml_parse/2 により、
   XML 文書を Document-Value 形式に変換する
3. SICStus 上で BerkeleyDB モジュールを用いて、
   Document-Value 形式のデータを BerkeleyDB に格納する

xml_parse による変換で、XML 文書はその要素構造を保持したまま、Prolog の項として表現された Document-Value 形式に変換される。要素構造 DB の構築は、このデータの全ての項を BerkeleyDB に、「xml(*PageID*, *Document-Value*)」という書式で格納することで完了する。

しかし、こうして作成した DB をそのまま利用するのには一つの欠点がある。BerkeleyDB モジュールで定義されている、BerkeleyDB へのアクセス用の述語をそのまま利用する場合、その書式が以下のように若干煩雑になってしまう。

BerkeleyDB の利用 (通常)

```
% DB 名、モードを指定して DB をオープン
  db_open(xml,read,_,DBRef),
% DB への参照を指定して目的の項を取り出す
  db_fetch(DBRef,xml(PageID,Document),_).
```

この利用法の場合、ユーザーは BerkeleyDB の存在を意識して DB をオープンしなくてはならず、これでは Verno-SP で目標とした「データの単一ビュー」に大きく反してしまう。また、利用する上で常に DB への参照を変数として引き回さなければならないため、記述性も低下する。

そこで、Verno-SP から BerkeleyDB を利用するにあたり、以下のように皮を被せて xml/2 を定義することにより、BerkeleyDB へのアクセスを隠蔽した。

BerkeleyDB の利用の隠蔽

```
:- db_open(xml,read,_,DBRef), bb_put(dbref,DBRef).

xml(PageID,Document):-
        bb_get(dbref,DBRef),
        db_fetch(DBRef,xml(PageID,Document),_).
```

bb_put/2 及び bb_get/2 は共に、内部 DB やバックトラックとは別に項を保存する、副作用のある SICStus の組み込み述語である。上記の例では、起動時に BerkeleyDB がオープンされ、その参照がこの bb(BlackBoard) 手続きにより保存される。そして、xml/2 が呼ばれた際にはこの参照が bb_get/2 により取り出され、述語の内部で db_fetch/3 による BerkeleyDB へのアクセスが行われる。

これら xml/2 内部での述語は、link/2 の実装の時と同様、モジュール化により隠蔽され、ユーザーは BerkeleyDB の存在を意識することなく xml/2 を利用することができる。

5.4 Socket 通信を行う述語の実装

この節では get_index/2 及び contain/3 の実装について述べる。これらは共に、検索エンジンの核とも言える「文字列による検索」を扱う述語である。この重要な機能を、処理系の外部のプログラムの利用により実装したのは、以下の理由からである。

第一に、本研究が目標としているのはあくまで、「Prolog の記述性を生かした検索システム」であり、その構築を全て Prolog によって行うことではないということ。Verno では N-gram によるインデックスを作成する方式で文字列検索機能を提供しているが、このインデックス DB を Prolog を用いて実装する場合、Prolog が得意としない手続き的なプログラムを作成することとなり、しかもその内容の多くは研究の本筋とはあまり関係しない脇道となってしまう。

第二に、Socket 通信方式を採用しても、その通信によるオーバーヘッドは少ないと考えられること。Verno においてはインデックス DB を利用するための簡易検索言語サーバー fhandle が存在しており、Verno のインデックス DB を効率的に利用するためのチューニングが施され、実際に学内への検索サービスを提供している。通信が必要となるのは一つの文字列に付き一度の送信と、その結果の受信だけであり、fhandle を利用できるメリットと比べると、この通信によるデメリットは極僅かであると考えられる。

5.4.1 簡易検索言語サーバー fhandle2 の仕様

今回 Socket 通信を行う相手となるのは、WWW クロウラー larbin が収集したデータから作られたインデックス DB を利用するためのサーバープログラム「fhandle2」である。fhandle2 には以下の命令が用意されている[3]。

命令	内容
GET␣index:*Keyword*	全文から *Keyword* の出現位置を検索し、最初にその HIT 数を返す。また、その出現ページ ID、ページ内のバイト位置情報、ページ内の HIT 件数を、一行につき一件ずつ、全件分羅列する。同一ページ内に複数 HIT した場合、その全件が表示に含まれる。
get␣index:*Keyword*	全文から *Keyword* の出現位置を検索し、最初にその HIT 数を返す。また、その出現ページ ID、ページ内のバイト位置情報、ページ内の HIT 件数を一行につき一件ずつ羅列する。同一ページ内に複数 HIT した場合、そのページでの最初の HIT 情報のみが表示される。
getdig␣index:*Keyword*	全文から *Keyword* の出現位置を検索し、その出現ページ ID、ページ内のバイト位置情報、ページ内の HIT 件数を表示する。さらにページの URL、及びキーワードの周囲の文字列も同時に表示する。

表 5.4: fhandle2 で使用することが出来る命令

述語 get_index/2 が Socket 通信を用いて利用する命令「get␣index:*Keyword*」は具体的には以下のような文字列を返す。

get index 命令が返す文字列の例

```
hits: 290
911 4120 1
1658 1786 3
... (中略) ...
149305 1921 1
149310 2129 1
```

[3] この仕様は本論文提出時の暫定的なものであり、これから拡張が施される可能性が高い。

5.4.2 get_index/2 の実装

get_index/2 のアルゴリズムは実に単純である。4 章で述べた仕様の通り、この述語は第一引数によって検索キーワードを指定し、第二引数として検索結果のリストを返す。get_index/2 は以下の動作を、単純に手続き的に記述した Prolog プログラムである。

1. ホストとポートを指定して、fhandle2 とのソケット通信を開始する
2. キーワードを元に検索クエリを作成し、fhandle2 に送信する
3. fhandle2 から検索結果の文字列を受信し、ソケットを閉じる
4. 検索結果の文字列を解析し、ヒット情報のリストとして返す

この仕様と実装は、fhandle2 の仕様を忠実に反映したものである。その結果、この述語自体はあまり Prolog らしくない述語となった。

前項で示した fhandle2 の返す文字列は、get_index/2 では、以下のようなリストとして扱われる。

get_index/2 の動作例

```
|?- get_index('論理型言語',Results).
Results = [hit(911,4120,1),hit(1658,1786,3),
 ...(中略)...
,hit(149305,1921,1),hit(149310,2129,1)] ?
yes
```

5.4.3 contain/3 の実装

前述の get_index/2 を用いれば、取りあえずの文字列検索は行える。しかし、get_index/2 の動作はあまり Prolog らしいとも言えず、また使いやすいとも言えない。そこで、Verno-SP ではさらに、get_index/2 を用いつつ、宣言的な記述を実現するための述語 contain/3 の実装を行った。

contain/3 の仕様である「contain(*PageID*, *Keyword*, *HitNum*)」を単純に実現するためには、検索キーワード *Keyword* を元に前述の get_index/2 を呼び、その結果のリストを元にページ ID *PageID* と HIT 件数 *HitNum* を求めれば良い。しかし、そうした実装では、同じキーワードによる検索であっても、contain/3 が呼び出されるたびに get_index/3 が呼び出され、結果として何度も無駄な Socket 通信を行うことになってしまう。

今回は、この無駄を避けるために、内部 DB を利用した以下のような実装をおこなった。

contain/3 の動作

***Keyword* がまだ一度も検索されていない場合**

get_index/3 を用いて *Keyword* を検索し、
assert/1 を用いてその結果を以下の形式で内部 DB に展開する。
verno_contain(*PageID*, *Keyword*, *HitNum*).

***Keyword* が既に検索されていた場合**

すぐに
verno_contain(*PageID*, *Keyword*, *HitNum*).
を呼び出す。

この際、*Keyword* による検索が行われたかどうかは、内部DB に「verno_contain(_, *Keyword*, _)」が存在するかどうかで容易に判定できる。また、assert/1 は内部DB に fact を追加する Prolog の組み込み述語である。

この実装の結果、宣言的で自然な記述を実現しただけでなく、Socket 通信による検索結果を内部 DB に保持して再利用することにより、無駄な通信を防ぐこともできた。

また、内部用述語である verno_contain/3 はモジュールにより隠蔽され、ユーザーはこれを意識することなく contain/3 を利用することができる。

5.5 Prolog を用いた単一ビュー環境の構築

本章では、Verno-SP における述語群の実装に関してかなり詳細に記述を行った。これは、今回 Verno の持つ多種の DB に対して行った各種の実装が、異なるデータベース群を対象としても応用可能であると考えたためである。

- 単純に内部 DB を用いるだけでなく、
  link/2 の様に複数の項にインデックスを効かせる手法
- 大規模なデータを BerkeleyDB に格納して利用する手法
- Socket 通信で外部プログラムを利用し、
  その結果を内部 DB に展開して利用する手法
- 及び、上記のような手法を一般ユーザーから隠蔽する手法

以上の様な手法は Verno-SP 以外の環境においても有効であり、本研究では、Verno-SP の実装を、「Prolog を用いた単一ビュー環境の構築」の一つのテストケースとして提示する。

第6章　Verno-SPを利用した検索

6.1　Prologの記述性を活かした検索

Prologを検索言語として採用したのは、その高い記述性を、検索クエリを記述する能力として活かすことが出来ると考えたためである。本節では、Prologの記述性を活かした検索を、具体的な例を挙げながら解説する。

6.1.1　宣言&探索による検索

Prologの特長の一つとして、「宣言的な記述」が可能であることが挙げられる。モノとモノとの間の関係を宣言することがそのまま述語の定義になるというものだ。5章でも述べたように、Verno-SPの設計においてもこの点は重視されており、Verno-SPに実装した述語群を用いて目標とするページを宣言することにより、単純かつ明快に検索クエリを作成することができる。

例1

「上田研究室」というキーワードを含み、かつ「上田研究室学生一覧のページ」からリンクされていて、かつ「上田研究室日本語トップページ」へリンクしているページを検索する。

例1

```
sample1(PageID):-
      contain(PageID,'上田研究室',_),
      url(A,'http://www.ueda.info.waseda.ac.jp/student_j.html'),
      link(A,PageID),
      url(B,'http://www.ueda.info.waseda.ac.jp/home_j.html'),
      link(PageID,B).
```

この例題は、実は永澤氏の卒業論文[6]において挙げられた、リンク情報を用いた検索の例題の一つである。Schemeでは1ページ分のソースコードになったこの例題が、Verno-SPにおいては僅かに6行で宣言できる。また、この記述を行うに当たって必要な知識は、Verno-SPに実装されている述語と、最低限のPrologの知識だけであり、それさえあれば上記のプログラムはすぐに書くことが出来る。

上記の sample1/1 の定義の場合、このままでは「例題の条件を満たすあるページ」を表すクエリとなっており、実行すると Prolog は、条件を満たすページを一つだけ表示する。その解答に対して「;」を入力してバックトラックを行うと、以下のように次々と条件を満たすページが表示される。

sample1/1 の実行例

```
| ?- sample1(PID),url(PID,URL).
PID = 2686,
URL = 'http://www.ueda.info.waseda.ac.jp/~tomohito/index.html' ?
;
PID = 3867,
URL = 'http://www.ueda.info.waseda.ac.jp/~ichibe/index.html' ?
;
PID = 12036,
URL = 'http://www.ueda.info.waseda.ac.jp/~n-kato/index-j.html' ?
yes
```

[1]

また、Prolog には、この様にして条件に合う解を一つ一つ探索する方法以外にも、バックトラッキングにより全ての解を求め、解のリストを返す手続きが用意されている。

findall/3 を利用しての探索

```
| ?- findall(PID-URL,(sample1(PID),url(PID,URL)),Results).
Results = [2686-'http://www.ueda.info.waseda.ac.jp/~tomohito/
index.html',3867-'http://www.ueda.info.waseda.ac.jp/~ichibe/i
ndex.html',12036-'http://www.ueda.info.waseda.ac.jp/~n-kato/i
ndex-j.html'] ?
yes
```

findall/3 は、第二引数の式を満たすような解をバックトラッキングにより全て見つけ出し、その解を第一引数で指定した書式で、第三引数のリストとして返す。Verno-SP ではこのように、「宣言的に定義を行い、探索により定義を満たす解を見つける」という Prolog 的な記述を推奨する。

[1] sample1/1 だけだと結果が PageID のみでわかりにくいので、
上記の例ではさらに url/2 を用いて結果のページの URL を表示させた

例 2

サイトのトップページ (URL の最後の要素が'index.html'、'index.htm'、もしくは省略されているページ) を検索する。

例 2

```
sample2(PageID):-
      urllist(PageID,URL),
      (
        last(URL,'index.html');
        last(URL,'index.htm' );
        last(URL,'/')
      ).
```

述語 urllist/2 は、URL をその「/」区切り毎に分割したリストを返す。この例 2 のケースでは、このリストの最終要素 (つまり URL の最後尾) が、'index.html' or 'index.htm' or '/' [2]であることを宣言している。

例 3

相互リンクされている 2 つのページを探し、その中でも特に、お互いのドメインが異なっている例を見つける。

例 3

```
sample3(PID1,PID2):-
      link(PID1,PID2),
      link(PID2,PID1),
      urllist(PID1,[Domain1|_]),
      urllist(PID2,[Domain2|_]),
      \+(Domain1 = Domain2).
```

単純に相互リンクだけなら、最初の 2 行で表現できる。上記の例ではさらに、同一ドメイン内では面白くないということで、そのトップドメインを呼び出し、それらが等しくない事を宣言している。

例 3 を少し改造して、「同じ単語を含むような相互リンク」の検索を行うのはとても簡単である。これらの述語の組み合わせは単純な様でいて、その組み合わせ次第で実に多用な検索が行える。

[2]最終要素が省略されている場合、urllist/2 が返すリストの最後尾には'/' が格納される

6.1.2 要素構造に対する検索

従来の Verno にはなく、今回 Verno-SP において新たに加わった機能として、HTML の持つ、要素の木構造に対するデータ処理が挙げられる。これは、SICStus の xml モジュール [3] の機能を活用したものであり、HTML 形式の文書を、同モジュールが提唱する Document-Value モデルに変換して Prolog に取り込むことにより、Prolog の持つ構造に対する探索能力を活かして、HTML の要素構造を扱える。

例 4

あるページの HTML から、TITLE タグに挟まれた文字列を見つける。

例 4

```
sample4(PageID,Title):-
      xml(PageID,Doc),
      xml_subterm(Doc,element(title,_,Elements)),
      Elements = [pcdata(TitleCode)|_],
      name(Title,TitleCode).
```

例 4 ではまず、xml/2 により Document-Value 形式の構造データ *Doc* を取り出す。そして xml モジュールで定義されている述語 xml_subterm/2 により、*Doc* に含まれる構造の中でも、「element(title,*Args*,*Elements*)」という構造を取り出す。最後に、title タグに挟まれた構造を表す *Elements* から、文字列 pcdata(*Code*) を取り出し、文字コードである *Code* をアトムに変換して返す。

Prolog は元来構造データを扱うのに優れた言語であり、例 4 においてもそれを活かして構造に対する検索を行っている。しかしさすがに、こうした検索クエリを作成するためには、前節で挙げた知識に加えて、さらに Document-Value 形式の構造の文法、及び Prolog での構造に対する検索の書き方の知識が必要となる。

sample4/2 の実行例

```
| ?- sample4(1,Title).
Title = '早稲田大学' ?
yes
```

6.1.3 探索的な検索

Prolog には、バックトラックによる「探索」が書きやすいという特長がある。このバックトラックによる探索を活かした例題として、以下の様なものを考えた。

例 5

ある特定のページを根として、リンクを辿り、ページの内容によってはバックトラックを行いながら、複数のページに処理を行う。

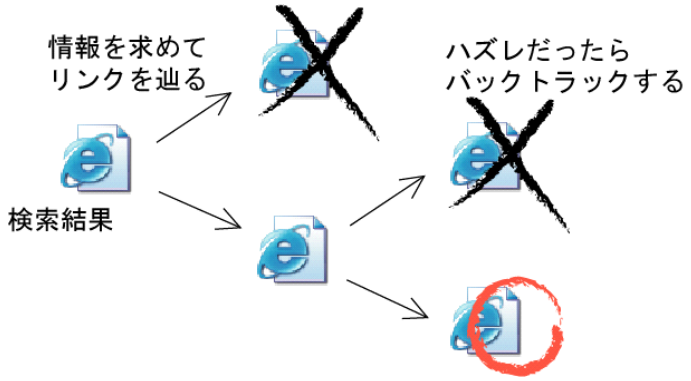

図 6.1: バックトラックを行いながら WWW ページを探索する

特定のページを根とした探索的検索 (一部)

```
tree_search(PageID,Lv):-
    set_known([]),
    repeat,
        ts(PageID,D,Lv,Lv,Ctrl),
        (Ctrl = end,!;
        not_known(D),add_known(D),do_something(D),
    fail),
    bb_delete(known,_).

ts(D,     D,Lv,ILv,Ctrl):-Lv >= 0,
    Ctrl = continue.
ts(PageID,D,Lv,ILv,Ctrl):-Lv >= 0,
    Lv1 is Lv-1,
    ts(PageID,D0,Lv1,ILv,Ctrl),
    link(D0,D).
ts(_,_,Lv,Lv,end).
```

このプログラムにおいて、後半で定義されている述語 ts/5 は、Prolog で反復深化によりグラフ探索を行う場合の典型的な書き方である。前半では、repeat-fail ループを用いて、この ts/5 をバックトラックさせることにより、ts/5 の返すページが条件[3]を満たすまで、バックトラックによる探索が行われる。

このプログラムでは、既知ページ[4]を管理するために、バックトラックの外側で情報を保持することの出来る組み込み述語 bb_put/2、bb_get/2 等を用いて、not_known/1 や add_known/1 を定義した[5]。

この例はもはや「検索クエリ」というより「検索を行うプログラム」そのものである。当然、プログラミングにはアルゴリズムの知識も必要となる。しかし、こうした複雑なプログラムを、言語と処理系が許す範囲において自由に活用できるというのも、Verno の持つ大きな利点である。

6.2 Scheme との比較

本章ではここまで、Verno-SP を用いた検索クエリの作成が、容易かつ簡潔であることを述べてきたが、本節では、

- その記述性が具体的にどの程度高いか
- そうして作成したクエリがどの程度の実行速度を持つか

という 2 点に関して、従来の検索言語 Scheme との比較を行うことで、Verno-SP の能力を評価する。

[3] この例のプログラムでは、探索を行う深さが、最初に指定した探索深度 Lv を越えるまで
[4] 既に訪れたページ
[5] それらの述語の定義は紙面の関係で割愛した

第7章　まとめと今後の課題

7.1　まとめ

本研究では、テストデータによる実験と Verno-SP の実装を通じて、Prolog による大規模データ処理の性能評価を行った。これらの測定データは Verno-SP の実装を行う際の指針となっただけでなく、今後 Prolog を用いて大規模なデータを扱う際においても参考にすることができるであろう。

また、Verno-SP の実装においては、外部のデータであっても内部のデータと同様に扱える、「データの単一ビュー」を実現した。この単一ビューを実現するために用いたいくつかの手法は、本研究で扱った Verno データ以外に対しても適用可能である。

最終的に、今回の Verno-SP の実装により、検索クエリの記述性が大幅に向上し、従来の Verno における検索クエリと比べ、直感的かつ自然な検索クエリの記述が可能となった。

7.2　今後の課題

今後の課題としては以下のようなことが挙げられる。

7.2.1　全国版への対応

今回の Verno-SP の実装は、早稲田学内約 15 万件を対象として行われている。しかし、現在 Verno では larbin による大規模な収集及びそのインデクシングにより、1000 万件規模の全国版サービスに向けての実装が着々と進んでいる最中であり、Verno-SP もできればこの規模に対応したい。

全国版への対応を行う場合、多くのデータはその規模が 100 倍近く膨れあがり、またリンク情報はさらに莫大になるはずである。こうしたデータの増大に対応するためには、述語の実装のさらなるチューニング及び、外部データベースの効率的な利用を模索する必要があるだろう。

7.2.2　サービスの公開

本研究では、実際に Verno-SP を用いて様々な検索クエリを記述した。しかし、これらの検索を利用するためには、Verno-SP が稼働しているマシンにログインして実行を行う必要があり、Verno のサービスとして一般公開するには至っていない。

今後は Verno-SP を利用した、何か面白い検索を模索すると共に、是非近日中にはサービスの公開を行いたい。

謝辞

参考文献

[1] 佐伯千種, 島田博也, 田畑伸哉 :
WWW コンテンツ統計調査報告書, 総務省情報通信政策研究所, 2004.

[2] 河邊昌彦，二村良彦 :
Random Graph Server
http://www.ispt.waseda.ac.jp/~kawabe/rgs/index.shtml

[3] xml.pl – Parsing XML with Prolog :
http://homepages.tesco.net/binding-time/xml.pl.html

[4] larbin :
http://larbin.sourceforge.net/index-eng.html

[5] SICStus Prolog User's Manual :
http://www.sics.se/isl/sicstuswww/site/documentation.html

[6] 永澤大介 :
WWW 全文検索システム Verno におけるリンクデータベースの設計と実装,
早稲田大学理工学部情報学科 2001 年度卒業論文，2001

[7] Yahoo! : http://www.yahoo.com/

[8] AltaVista : http://www.altavista.com/

[9] goo : http://www.goo.ne.jp/

[10] Google : http://www.google.com/

[11] infoseek : http://www.infoseek.co.jp/

[12] HTML TIDY : http://www.w3.org/People/Raggett/tidy/

[13] Verno : http://verno.ueda.info.waseda.ac.jp/

[14] 早稲田大学ホームページ : http://www.waseda.ac.jp/

[15] verno-elk : http://verno.ueda.info.waseda.ac.jp/network/scheme.html

[16] Iron33 : http://verno.ueda.info.waseda.ac.jp/iron33/

[17] w3grep : http://verno.ueda.info.waseda.ac.jp/w3grep/