空間データベースサーバを用いた
Web システムの総合的性能評価

島根大学　総合理工学部　数理・情報システム学科

計算機科学講座　田中研究室

S053032　熊谷 恵理

目次

第１章　序論

１．１．研究目的

現在利用されているベンチマークには，Dacapo ベンチマーク[1]や，アプリケーションサーバと Web サービスの評価用ベンチマークである TPC-App[2]などがあるが，これらのベンチマークは空間データベースに未対応である．そこで，本研究では，ベンチマークテストの前段階として，先行研究で構築された「国道沿いの人口分布を表示するシステム」[3]を用いて，処理時間を中心とした総合的計測を行った．

１．２．研究概要

前述したシステムでは，地図情報などの空間データを容易に扱うことが出来，近傍検索のような空間関係を指定して検索することが出来る ORDBMS を使用しており，Web サーバには拡張性が高い Apache，クライアントーWeb サーバ間をつなぐ言語には Java サーブレット，サーブレットコンテナには Tomcat を用いて実装されている．そして，クライアントには，Google Maps 上に追加して描画することが出来る Google Maps API を用いて描画を行っている．

本研究では，このシステムの処理単位別の時間，DB サーバと Web サーバのリソース等を計測した。

１．３．先行研究

先行研究調査として，Scopus を使用した．Scopus とは，世界の 4,000 以上の出版社から出版される 15,600 以上の科学・技術・医学・社会科学の論文情報・特許情報・Web 情報を網羅する書誌・引用文献データベースである．

検索キーワードと検索結果を表１.3.1 に示す.

表 1.3.1　Scpus での検索結果

	キーワード	検索結果(件)
①	(((**improvement** AND (**bottleneck** OR **speed**)) OR (**performance evaluation**)) AND ((**web** OR **server**) AND **database**) AND **system**	502
②	((**improvement** AND (**bottleneck** OR **speed**)) OR (**performance evaluation**)) AND ((**web** OR **server**) AND **database**) AND **system** AND **object** AND **relational**	6
③	((**improvement** AND (**bottleneck** OR **speed**)) OR (**performance evaluation**)) AND ((**web** OR **server**) AND **database**) AND **system** AND **object** AND **relational** AND **(map** OR **spatial)**	1

②の検索キーワードに**(map** OR **spatial)** を付け加えたところ(③)，１件の検索結果が得られたが，本研究と関連のあるものではなかった．

第２章　国道沿いの人口分布を表示するシステム

２．１．システムの概要

このシステムは「島根県内の国道沿いの左右帯域幅 x km($0.5 \leq x \leq 5$: 500m 毎) 内に含まれる選択された人口データに対する1km × 1km の基準地域メッシュの表示」という検索をWebサーバやDBサーバを用いて行い，Google Maps上に結果を表示するというシステムである．

図2.1.1にブラウザでの入力画面，図2.1.2にGoogle Mapsでの実行結果を示す．

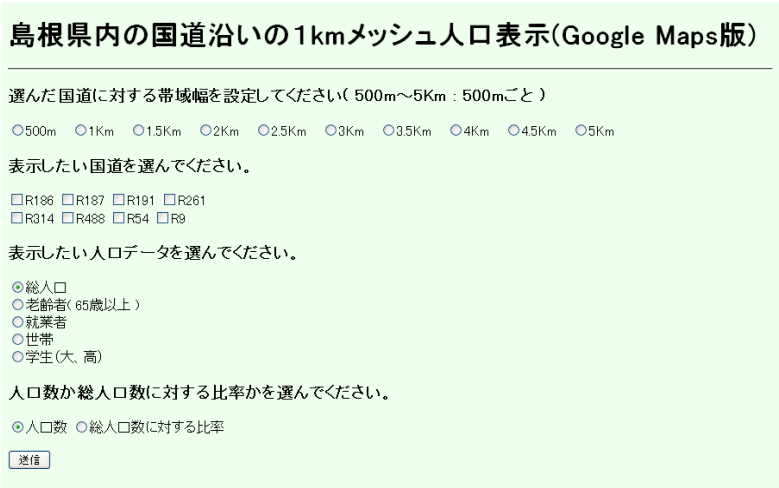


図 2.1.2　Google Maps での実行結果

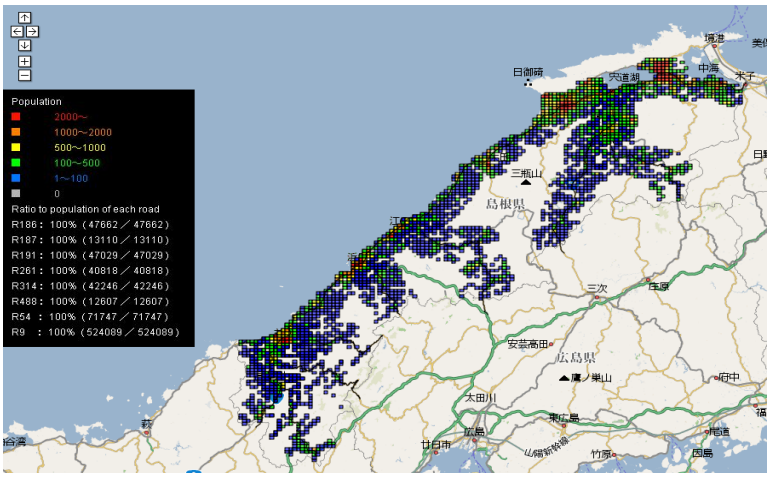


図 2.1.2　Google Maps での実行結果

２．２．システムの流れ

使用方法はまず，ユーザがブラウザ上で表示したい国道をチェックボックスで選択（複数可）し，その国道に対する検索範囲である帯域幅を指定する．帯域幅の範囲は500mから5kmまで500m毎の値を選択ボックス又はラジオボタンから１つ選択する．次に，知りたい人口データの種類と表示方法を選択する．表示方法は人口データの結果を人口数にするか比率にするかをラジオボタンで選択する（①）．最後に，送信ボ

タンをクリックし必要なデータをWebサーバへ送信する（②）．

Webサーバはそれらの引数からJavaサーブレットを用いてDBの操作に必要なSQL言語を生成する（③）．そして，JDBCを用いてＤＢMSに接続し，検索を行なう（④⑤）．最後に，DBMSから受け取った検索結果を用いてJavaサーブレットでGoogle Maps APIを生成し，Google Mapsに結果を描画する（⑥⑦⑧⑨）．

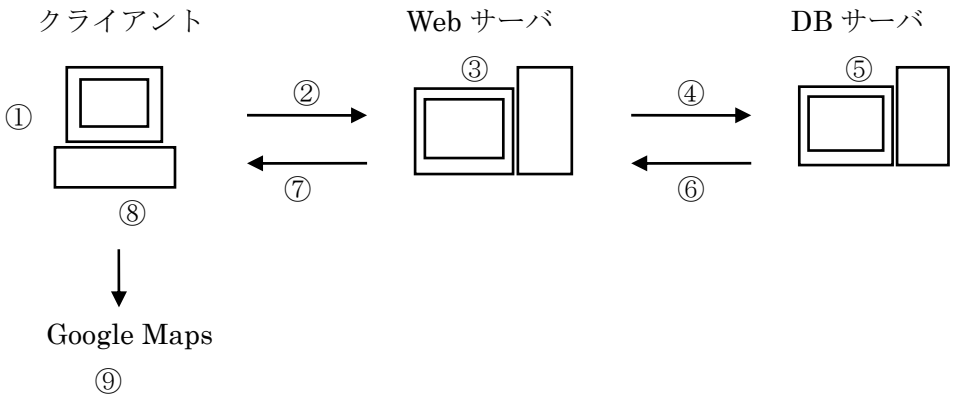


図2.2.1　システムの流れ

２．３．DB へのデータの格納

国道沿いの人口を検索するために，それぞれの国道に対して Buffer メソッドを用い，幅を持たせた Polygon 型の国道座標を生成する．この生成した国道座標に対して，Intersects メソッドを用い，人口メッシュとの交差を判定し，「交差している」又は「完全に含まれている」ものという検索を行う．

格納するデータは国道座標値データ，Polygon 型に変換した国道座標値データ，人口データ，基準地域メッシュのポリゴン座標値データである．

まず，これらの空間データを格納するには，ユーザ定義型の列を含む表を生成する必要がある．

・国道座標を格納する表「KOKUDOU」の生成

この表には，データ ID，国道名，国道座標（世界測地系）を格納する．

```
CREATE TABLE    KOKUDOU (
    ID              INTEGER,
    NAME            VARCHAR(100),
    GEO_CLMN        GEOMETRY
);
```

・人口データを格納する表「JINKOU」の生成

この表には，３次メッシュコード，総人口数，老齢者数，就業者数，世帯数，学生数，人口メッシュの座標（世界測地系）を格納する．

```
CREATE TABLE    JINKOU (
    MESH_CODE    INTEGER,
    JINKOU       INTEGER,
    OLDER        INTEGER,
    WORKER       INTEGER,
    FAMILY       INTEGER,
    STUDENT      INTEGER,
    GEO_CLMN     GEOMETRY
);
```

座標を格納する列の型名に GEOMETRY 型を指定している．GEOMETRY 型とは HiRDB 空間検索プラグイン(HiRDB Spatial Search Plug-in Version 3)によって定義されているユーザ定義型で，Point 型，Line String 型，Polygon 型といった空間データを格納することができる．国道座標の表については帯域幅毎に別の表を生成する．

次に空間座標データ格納方法について空間座標データは抽象定義型である GEOMETRY 型の列に格納するので，データの格納にはコンストラクタ関数を使用する．

・国道座標データ格納例

```
INSERT INTO KOKUDOU ( ID, NAME, GEO_CLMN)
    VALUES( 8, 'R9', GeomFromText('LINESTRING(60421 -50000,60570
            -50386,61473 -51420,61643 -51706, ･･･ , 186815 -171489,1
            86916 -171608)',9,1));
```

表「KOKUDOU」

ID	NAME	GEO_CLMN
8	R9	LineString(60421 -50000, ･･･)

・人口データ格納例

```
INSERT INTO JINKOU(MESH_CODE,JINKOU,OLDER,WORKER,FAMI
                        LY,STUDENT,GEO_CLMN)
    VALUES( 51313688, 8, 5, 0, 0, 0, GeomFromText('POLYGON((6800
            0 -38000,68000 -39000,69000 -39000,69000 -38000))',1,1));
```

表「JINKOU」

MESH_CODE	･･･	GEO_CLMN
51313688	･･･	Polygon((68000 -38000, ･･･))

GeomFromText 関数が GEOMETRY 型のコンストラクタ関数である．この関数を使用すると WKT 形式のあらゆる空間データを GEOMETRY 型データ列に格納することができる．

・空間インデックス生成方法

空間座標データを操作するには索引付けを行う必要がある．空間検索プラグインにおける空間インデックス生成例を以下に示す．

```
国道座標データへの空間インデックス作成
> CREATE INDEX  GEOIDX
    using type spatial on KOKUDOU（ GEO_CLMN ）
    in（ RLOB1 ）
    PLUGIN 'EXTENT=100,-174810,186916,300;
                DEPTH=1;PRECISION=1';

人口座標データへの空間インデックス作成
> CREATE INDEX  GEOIDXJ2
    using type spatial on JINKOU（ GEO_CLMN ）
    in（ RLOB2 ）
    PLUGIN 'EXTENT=54000,-280000,191000,-38000;
                DEPTH=4;PRECISION=1';
```

ここでは国道座標データへの空間インデックス生成について説明する．CREATE INDEX 文がインデックスを生成する SQL 文である．ここでは GEOIDX という名前で，表「KOKUDOU」の GEO_CLMN 列（Geometry 型）への空間インデックスを生成している．HiRDB では検索に Quad-Tree を用いている．この例では，深さ 1（DEPTH=1）の整数座標系（PRECISION=1）としている．

・空間座標データ検索方法

空間座標の検索も一般的な SELECT 文を用いて行う．座標データを抽出したい場合，AsText()関数を使用すると，空間座標データを WKT として返す．また，WHERE 句に条件を指定することも可能である．以下にその例を示す．

・国道座標を WKT 形式で検索する例

国道名が‘R54’である国道の座標を抽出する

```
> SELECT AsText(GEO_CLMN,1) FROM KOKUDOU
      WHERE NAME='R54' WITHOUT LOCK NOWAIT;

出力結果
linestring(140000 -111658,139458 -112439, ･･･,152496 -168087,152419 -168393)
```

この例では，国道座標データは GEO_CLMN 列に格納されている．AsText 関数によって，国道名が'R54'と一致するレコードの国道座標データを WKT 形式で出力している．

・国道沿いの人口を検索する例

国道座標を拡張（Buffer()）し，その範囲内（Intersects()）の人口データを抽出する．

```
> SELECT jinkou FROM JINKOU
      WHERE IntersectsIn(GEO_CLMN, RegionFromText(:R_GEOM, 500)
IS TRUE WITHOUT LOCK NOWAIT;
```

GEO_CLMN 列に人口メッシュ（Polygon），変数:R_GEOM は幅を持たせた国道座標（Polygon）が格納されているものとする．

この拡張した国道座標と人口メッシュ（GEO_CLMN）の共通集合演算をおこない，国道沿いの圏内に含まれていればその人口数を抽出する．

第 3 章　性能評価

３．１．計測方法

システム全体の処理単位別に時間を計測し，同時に DB サーバと Web サーバの CPU 使用率，物理メモリ使用量も計測した.

まず，システム全体のデータ・処理の流れについて説明する.

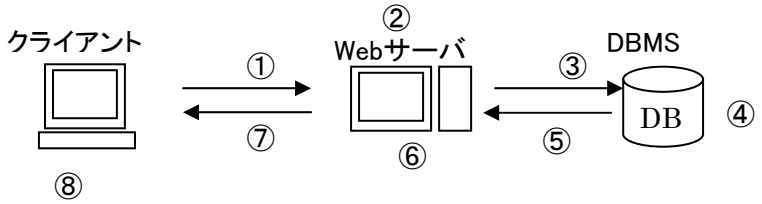


図 3.1.1　システム全体の流れ

① クライアントの Web ブラウザ上から Web サーバにリクエストを送信する.
② Web サーバがリクエストを受信すると，サーブレットに処理が渡される.
③ サーブレットは必要なデータ（座標，人口など）を得るために DBMS へ接続する.
④ DBMS 内部で必要なデータを検索する.
⑤ 検索したデータをサーブレットへ渡す.
⑥ サーブレットは受け取ったデータをもとに，Google Maps API を生成する.
⑦ 検索結果である Google Maps API を Web ブラウザへ送信する.
⑧ Google Maps API の場合は，Web ブラウザは Google Maps にアクセスし，結果 Google Maps 上に表示する.

①と⑦ではクライアントの Web ブラウザーサーバ間の通信である．①では国道，人口の種類，帯域幅などのデータを送信し，⑦では Google Maps API のデータを送信している．ここでは，大きなデータの送信はしていない.

②と⑥では同じサーブレットが処理を行っている．ここで行っていることは，リクエストに含まれるパラメータ（国道，人口の種類，帯域幅など）を JDBC を用いて，国道および人口を検索する SQL 文（SELECT 文）を作成し，DB にアクセスして実行する．ここでの主な処理は Google Maps API を生成することである.

③と⑤では Web サーバーDB サーバ間の通信である．③では生成した SQL 文を送信し，⑤では検索結果を送信している．ここでは，大きなデータの送信はしていない.

④の箇所では，主に２つのデータの検索を行っている．1 つは，指定した国道の空間データ（LineString）の検索である．この検索は，国道名もとに格納された空間データを AsText 関数で WKT 形式に返す，というものである．２つは，人口の検索である．１つ目で検索した国道座標（LineString）に Buffer メソッドを用い，帯域幅をもたせ Polygon に変換し，その Polygon と全ての人口メッシュ（Polygon）との共通集合演算

（intersection()）を行う．その結果 TRUE となる人口データを返す，というものである．

システム全体の処理時間①～⑦間の各箇所において計測を行うことにする．⑧については計測が困難であることから，本研究では，この計測を省くこととする．

３．２．計測箇所

実際に計測した箇所について，具体的に説明する．計測方法は，表 3.2.1 の時刻(1)～(13)を取得し，差を取ることによって表 3.2.2(a)～(h)の時間を計算した．なお，計測は 6 回ずつ行い，2～6 回目の計測結果の平均を求めた．

表 3.2.1　取得した時刻

(1)	送信ボタンが押されたときの時刻
(2)	サーブレット実行開始時刻
(3)	国道検索 SQL 文を動的に作成する直前の時刻
(4)	国道検索 SQL 文を動的に作成した直後の時刻
(5)	国道検索 SQL 文(SELECT 文)を実行する直前の時刻
(6)	国道検索 SQL 文を実行した直後の時刻
(7)	人口検索 SQL 文を動的に作成する直前の時刻
(8)	人口検索 SQL 文を動的に作成した直後の時刻
(9)	人口検索 SQL 文(SELECT 文)を実行する直前の時刻
(10)	人口検索 SQL 文を実行した直後の時刻
(11)	サーブレットが結果をブラウザへ返す直前の時刻
(12)	検索結果である Google Maps API を Web ブラウザが受信した時刻
(13)	Web ブラウザがページ読み込みを完了した時刻

表 3.2.2　計測箇所

	計測箇所	
(a)	クライアント→Web サーバ通信時間	(2)－(1)
(b)	国道検索 SQL 発行時間	(4)－(3)
(c)	国道検索 SQL 実行時間	(6)－(5)
(d)	人口検索 SQL 発行時間	(8)－(7)
(e)	人口検索 SQL 実行時間	(9)－(8)
(f)	サーブレット全体処理時間	(11)－(2)
(g)	Web サーバ→クライアント通信時間	(12)－(11)
(h)	送信ボタン→ページ読込完了時間	(13)－(1)

a. Web ブラウザから Web サーバまでの通信時間

まず,①でクライアントからリクエストを送信してから Web サーバで受信するまでの時間を計測する．リクエストの送信は，HTML を用いた送信フォームを用いている．ここでの送信時に，送信ボタンが押されたときの時刻を送信データに含めて送る．<FORM>エレメント（フォーム名：form1）の中に次のコードを記述する．

```
       :
01：<INPUT type="hidden" name="SendTime">
02：<INPUT type="hidden" name="TZOffset">
03：<SCRIPT language="JavaScript">
04：  <!−
05：  function nowTimeSet(){
06：    now = new Date;
07：    form1.SendTime.value=now.getTime();
08：    form1.TZOffset.value=now.getTimezoneOffset();
09：  }
10：  -->
11：</SCRIPT>
      :
12：<INPUT onClick="return nowTimeSet(form1)" type="submit" value="送信">
      :
```

01：と 02：は送信時刻とその時差を格納するフィールドである．これらは type=”hidden”とすることで非表示としている．03：～11：は 01：02：で用意したフィールド SendTime，TZOffset にそれぞれ時刻と時差を入力するための関数 nowTimeSet()（Java Script で記述）である．12:が送信ボタンの INPUT タグであるが，クリック時（onClick）に nowTimeSet()関数を呼び出し，時刻を入力後，リクエストを送信する．

送信されたデータは Web サーバが受信し，サーブレットが呼び出される．このサーブレットでは次のようにして送信データを受信する．

```
       :
13：java.util.Date rsvTime=new Date();//時刻を取得
       :
14：//クライアントの送信時刻(GMT,ミリ秒)
15：Long sendTime=Long.valueOf(req.getParameter("SendTime"));
16：Long ctTZ=Long.decode(req.getParameter("TZOffset"));
17：//eTime1:クライアント→Web サーバの通信時間
18：long eTime1=(rsvTime.getTime())+(rsvTime.getTimezoneOffset()*6000)
                          -(sendTime.longValue()+ctTZ.longValue()*6000);
       :
```

13：はサーブレット実行開始時に時刻を記録しておく．15：16：で HTML フォ

ームからのフィールド名 SendTime および TSOffset のデータを取得する．18：がサーブレット実行開始時刻からフォームデータの送信時刻の差をとることで求める．getParameter()メソッドを用いて HTML フォームからデータを取得しているが，文字列として取得する．差をとるためには，long 型などの数値データに変換する必要がある．ここでは，longValue()メソッドを用いて変換をおこなっている．

b. SQL による検索時間の計測

次に，SQL による検索時間を計測する（③～⑤の区間における計測）．国道検索および人口検索は JDBC による SQL プログラムを用いている．HiRDB 内部による計測は困難であるため，Java プログラムによる測定を行う．

Java プログラム内では，Date クラスの Date()メソッドにより時刻を取得し，上記の場合と同様に求める．以下はその記述例である．

```
        :
09：java.util.Date start=new java.util.Date();//開始時刻取得
10：
11：//計測対象のコード
12：
13：java.util.Date finish=new java.util.Date();//終了時刻取得
14：long eTime=finish.getTime()-start.getTime();//実行時間
        :
```

09：で開始時刻を取得する．11：の部分に計測対象のコードを記述する．この場合計測対象とは，SQL 文を作成する箇所と，DB へ接続し SQL によってデータを取得する箇所である．13：により終了時刻を取得し，14：によって実行時間を計算する．

c. サーブレット全体の実行時間

次に，②～⑥の区間におけるサーブレット全体の実行時間の計測である．サーブレットは Java 言語を用いているので，「２．SQL による検索時間の計測」と同様に Date クラスの Date()メソッドにより時刻を取得する．

```
        :
09：java.util.Date start=new java.util.Date();//開始時刻取得
10：
11：//計測対象のコード
12：
13：java.util.Date finish=new java.util.Date();//終了時刻取得
14：long eTime=finish.getTime()-start.getTime();//実行時間
        :
```

この場合，計測対象はサーブレットプログラム全体であるので，リクエストを受信しサーブレットが呼び出された時点から，結果をブラウザへ返す直前までである．

d. Web サーバから Web ブラウザまでの通信時間

次に，⑦で Web サーバが Google Maps API を送信して，それをブラウザが受信

するまでの時間を計測する.「3．サーブレット全体の実行時間」において取得したサーブレット終了時刻と，サーブレットがブラウザへ返すスクリプトの実行開始時間の差を求める.

サーブレットに以下のように記述する.

```
       :
09 : PrintWriter out = res.getWriter();
       :
13 : out.print("<script type=¥"text/javascript¥">¥n" +
14 :           "  <!--¥n" +
15 :           "    var scrstartTime=0;¥n" +
16 :           "    now = new Date;¥n" +
17 :           "    scrstartTime = now.getTime();¥n" +
18 :           "  -->¥n");
       :
```

16：で，サーブレットが結果をブラウザへ返し，スクリプトが開始された時刻を取得する.

e. システム全体の処理時間

最後に，システム全体の処理時間を計測する．なお，この時間には，ブラウザへの描画時間を含めない.「1．Web ブラウザから Web サーバまでの通信時間」で取得した送信時刻と，検索結果の読み込みが完了した時刻との差を求める.

サーブレットに以下のように記述する.

```
       :
09 : PrintWriter out = res.getWriter();
       :
13 : out.print("<body onload=¥"loadtime()¥">¥n" +
                              //ページが読込まれたときに loadtime 関数を呼び出す
14 :           "<script type=¥"text/javascript¥">¥n" +
15 :           " <!--¥n" +
16 :           "  var ltime=0;¥n"+
17 :           "  function loadtime(){¥n" +
18 :           "    ltime=(new Date().getTime()
                  -"+Long.valueOf(sendTime)+")/1000;¥n"+ //処理時間の計算
19 :           "  -->¥n");
       :
```

13 : の onload イベントでページが読み込まれたときに，システム全体処理時間を求める loadtime 関数を呼び出す． 18：では，new Date().getTime()で時刻を取得し，送信ボタンが押された時刻 Long.valueOf(sendTime)との差を計算している.

f. CPU 使用率・物理メモリの計測

計測に用いたツールはCRNMonitor1.10[4]である．これはCPU，メモリ，ネット

ワークをそれぞれ監視し，マルチプロセッサ対応，各種ログ出力機能を持つ．DBサーバとWebサーバにCRNMonitor1.10 を常駐させ，ログの記録を 1 秒ごとに行った，

g．実行環境

Web サーバの性能

CPU	: Celeron　2.0GHz
メモリ	: 512MB
OS	: Windows Server 2003 R2(SP1)
Web サーバ	: Apache2.2.8
Web サーバ･サーブレットコンテナ	: Tomcat 6.0.16

DB サーバの性能

CPU	: Celeron　2.8GHz
メモリ	: 512MB
OS	: Windows Server 2003 R2 (SP1)
DBMS	: HITACHI HiRDB/Single Server Ver.8
空間検索プラグイン	: HITACHI HiRDB Spatial Search Plug-in Ver.3

クライアントの性能

CPU	: PentiumⅣ 3.2GHz
メモリ	: 512MB
OS	: Windows XP Pro (SP2)
Web ブラウザ	: Firefox3.0.5

ネットワークの性能

LAN	: 100BASE-TX
ハブ	: CentreCOM MR820T

h．各マシン時間の同期

各マシンの時間の同期は Simple Network Time Protocol（SNTP）を用いた．

Network Time Protocol（NTP）とはネットワークで結ばれたコンピュータ同士で時刻を同期させるためのプロトコルで，SNTP はクライアントの時刻合わせ用途に向けて NTP を軽量化したプロトコルである．実験を行うたびに各クライアントの SNTP 設定を手動で行い時刻の同期をはかった．「net time」コマンドのオプションで SNTP サーバアドレスを指定してやると，SNTP サーバを参照しマシンの時刻が同期するようになる．今回 SNTP サーバ（時刻合わせ元）は 192.168.1.9 である．コマンドは以下のようになる．

```
＞net time ¥¥192.168.1.9 /set /yes
```

３．３．計測結果

検索条件を以下のようにし，処理時間を計測した結果の一例を表 3.3.1 に示す.

検索条件	
国道	：R54（153 点）
帯域幅	：500
人口	：総人口

表 3.3.1　計測結果(単位)

	計測箇所		結果
(a)	クライアント→Web サーバ通信時間	(2)-(1)	0.03
(b)	国道検索 SQL 発行時間	(4)-(3)	0.02
(c)	国道検索 SQL 実行時間	(6)-(5)	0.14
(d)	人口検索 SQL 発行時間	(8)-(7)	0.04
(e)	人口検索 SQL 実行時間	(9)-(8)	1.70
(f)	サーブレット全体処理時間	(11)-(2)	2.17
(g)	Web サーバ→クライアント通信時間	(12)-(11)	0.28
(h)	送信ボタン→ページ読込完了時間	(13)-(1)	2.72

表 3.3.1 計測結果より，システム全体の処理時間が 2.72 秒なのに対し，人口検索 SQL 実行時間は 1.70 秒となっており，この処理がボトルネックとなっていることが分かる．なお，SQL 発行時間は SQL 文を動的に作成し，実行の準備にかかる時間である．SQL 実行時間は SQL 文（SELECT 文）を実行しデータを抽出するのにかかった時間である.

次に，検索メッシュ数と処理時間の関係を表 3.3.3 と図 3.3.1 に示す．検索メッシュ数とは，国道に帯域幅をもたせた Polygon と全ての人口メッシュ（Polygon）との共通集合演算（intersection()）を行った結果 TRUE となる人口メッシュ数のことである．なお，検索条件の国道と帯域幅を変更した際の，メッシュ数は表 3.3.2 のようになる.

表 3.3.2　メッシュ数

帯域幅(m)	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
R187(92 点)	50	60	76	85	98	107	126	137	154	167
R191(158 点)	69	89	107	125	147	156	174	190	210	226
R261(143 点)	88	110	136	156	184	204	235	259	291	313
R314(117 点)	64	92	110	132	153	175	200	224	252	280
R488(169 点)	37	47	60	72	85	100	117	134	149	168
R54(153 点)	129	164	217	257	305	339	385	418	462	489
R9(475 点)	412	546	670	766	869	962	1049	1140	1242	1335

表 3.3.3　検索メッシュ数と処理時間の関係

メッシュ数	人口検索 SQL 実行時間	サーブレット全体 処理時間	送信ボタン→ 読込完了	メッシュ数	人口検索 SQL 実行時間	サーブレット全体 処理時間	送信ボタン→ 読込完了
37	1.4346	1.8686	2.282	156	1.4436	1.944	2.358
46	1.247	1.6782	2.0848	159	1.281	1.703	2.188
47	1.25	1.6936	2.1288	164	1.5092	1.9688	2.4112
50	1.3312	1.8	2.2064	167	1.4094	2.1028	2.5382
60	1.2564	1.6906	2.1142	168	1.3156	1.7378	2.4664
60	1.2124	1.6532	2.3368	174	1.3564	1.8186	2.2464
64	1.2718	1.7376	2.2154	174	1.3592	1.8062	2.2514
68	1.219	1.656	2.297	175	1.3374	1.7596	2.2644
69	1.5782	2.0312	2.4486	184	1.5094	1.9622	2.3912
72	1.25	1.672	2.133	190	1.4092	1.853	2.289
76	1.2534	1.6938	2.107	193	1.3782	1.8126	2.2762
85	1.2626	1.7062	2.1802	200	1.3782	1.8188	2.3138
85	1.2406	1.6748	2.1126	204	1.4688	1.9128	2.377
86	1.2718	1.7032	2.123	210	1.4088	1.8594	2.3012
88	1.6406	2.0934	2.5068	217	1.6066	2.0564	2.5352
89	1.3876	1.8814	2.3002	224	1.3934	1.825	2.3582
92	1.2592	1.7026	2.2206	226	1.4344	1.8844	2.3286
98	1.225	1.6688	2.0948	235	1.5346	1.9908	2.4664
99	1.25	1.703	2.236	252	1.4376	1.8688	2.3754
100	1.2562	1.6844	2.5146	257	1.653	2.1184	2.607
107	1.2216	1.6596	2.0896	259	1.4656	1.9034	2.411
107	1.3624	1.831	2.2522	280	1.5064	1.947	2.463
110	1.4718	1.9438	2.352	291	1.5126	1.969	2.479
110	1.3688	1.8316	2.2752	305	1.8314	2.3094	2.7876
113	1.2714	1.6972	2.1368	313	1.6844	2.147	2.626
117	1.247	1.6814	2.3692	339	1.7968	2.2656	2.7944
125	1.4374	2.1158	2.5612	385	1.9032	2.397	2.9412
126	1.2376	1.6814	2.117	412	4.6062	5.2656	5.7058
129	1.6844	2.1376	2.5694	418	2.0596	2.556	3.1408
130	1.312	1.75	2.225	462	2.2564	2.7346	3.2938
132	1.3748	1.7874	2.2934	489	2.3592	2.8312	3.4146
134	1.2842	1.7092	2.1602	546	4.7312	5.5472	5.9396
136	1.4686	2.1218	2.6098	670	5.1218	5.844	6.3566
137	1.3908	1.8626	2.2938	766	5.5816	6.2746	6.8458
143	1.25	1.688	2.151	869	6.2376	6.9218	7.5314
147	1.375	1.8438	2.2238	962	7.0218	7.7468	8.3866
149	1.297	1.734	2.794	1049	7.853	8.5566	9.2398
153	1.331	1.7628	2.228	1140	8.8906	9.6154	10.344
154	1.3564	1.8188	2.2464	1242	10.5812	11.256	12.0142
156	1.3626	1.8186	2.2546	1335	12.6	13.3	14.0996

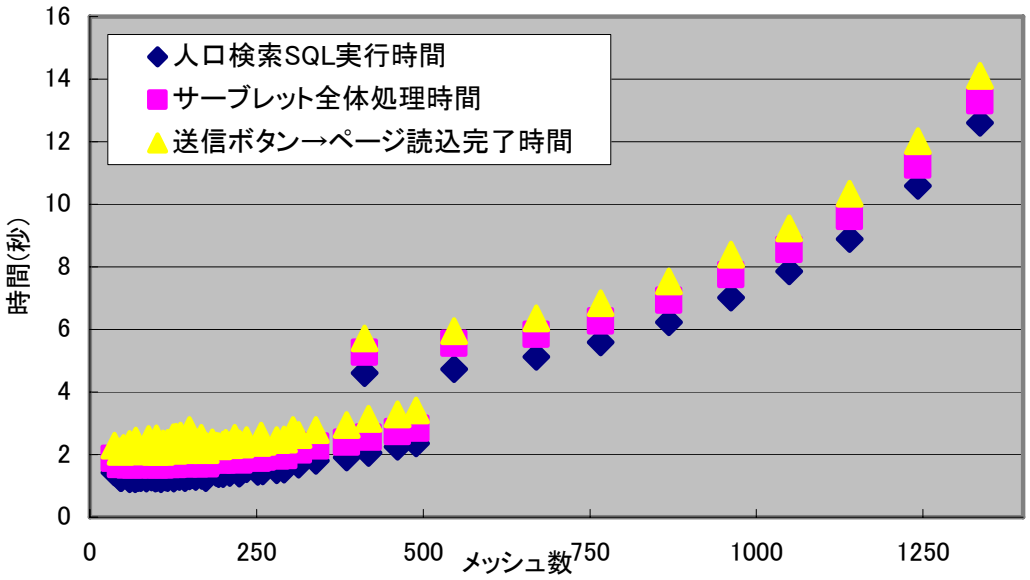


図 3.3.1　検索メッシュ数と処理時間の関係

次に，検索条件を以下のようにしたときの，各サーバの CPU 使用率と物理メモリ使用量は図 3.3.2，図 3.3.3 のようになる．なお，横軸はクライアントが送信ボタンを押してからの経過時間である．すなわち，横軸の値が 0 のときが，送信ボタンの押された時刻となっている．

検索条件
　国道　　：R9（475 点）
　帯域幅　　：1000
　人口　　：総人口

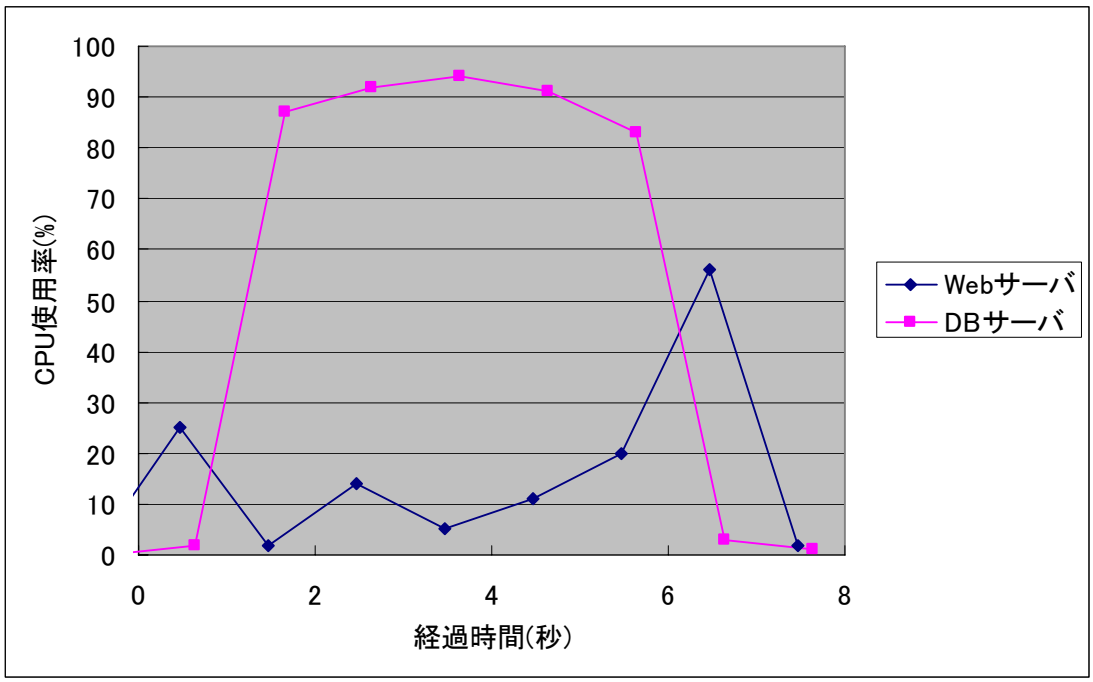


図 3.3.2　CPU 使用率

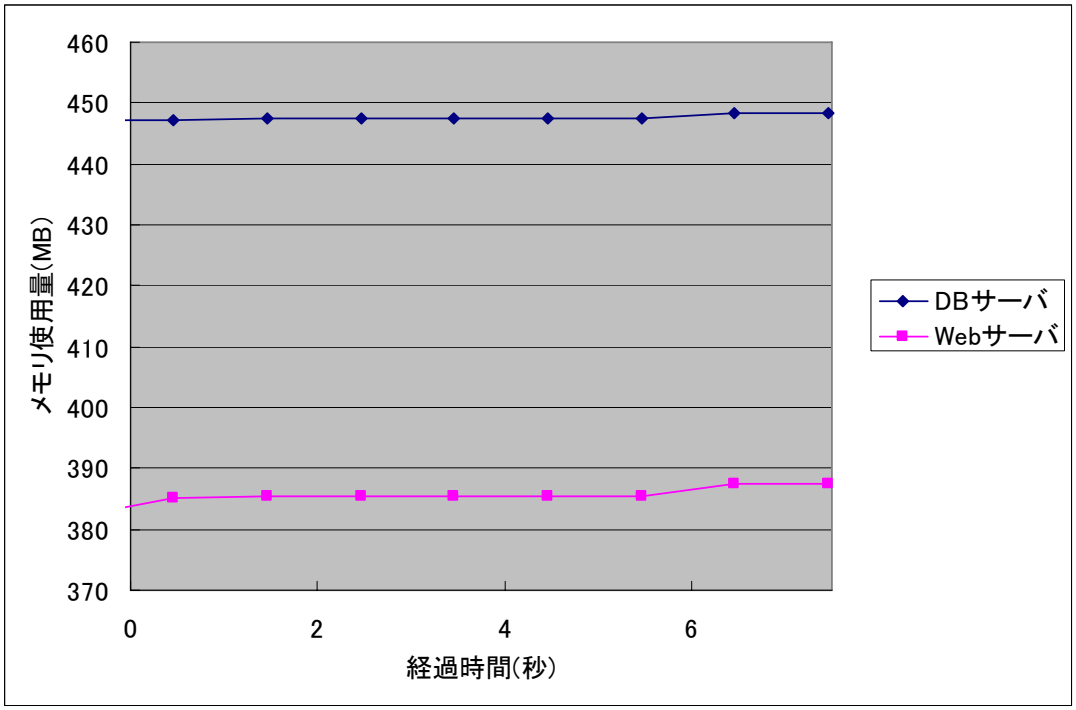


図 3.3.3　メモリ使用量

３．４．考察

表 3.3.3 より，一部抜粋したものが表 3.4.1 である．表 3.4.1 を見ると，①と②のメッシュ数はほぼ同じであるが，処理時間には約 2 倍の差がある．

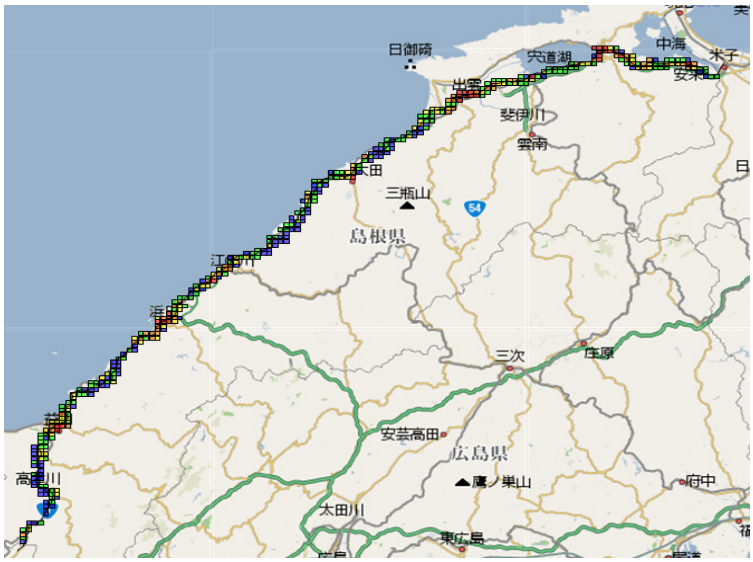
表 3.4.1　メッシュ数と処理時間の関係

	メッシュ数	人口検索 SQL 実行時間	サーブレット全体処理時間	送信ボタン→読込完了
①	412	4.6062	5.2656	5.7058
②	418	2.0596	2.556	3.1408

処理時間に差が出る原因として一つ目に，四分木が考えられる．

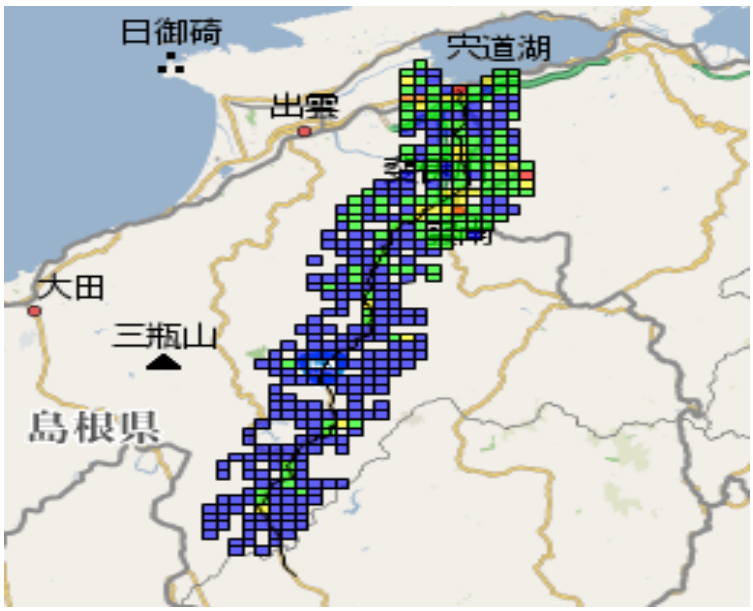
①と②は以下の検索条件のときの計測結果である．

①



R9	475 点
帯域幅	500 m
メッシュ数	412
人口 SQL 実行時間	4.60s

②



R54	153 点
帯域幅	4000m
メッシュ数	418
人口 SQL 実行時間	2.05s

このことから，検索範囲となる帯域幅を持たせた Polygon と人口メッシュが交差する部分(図 3.4.1 の色付けしてある部分)が多くなれば，処理に時間がかかると思われる．

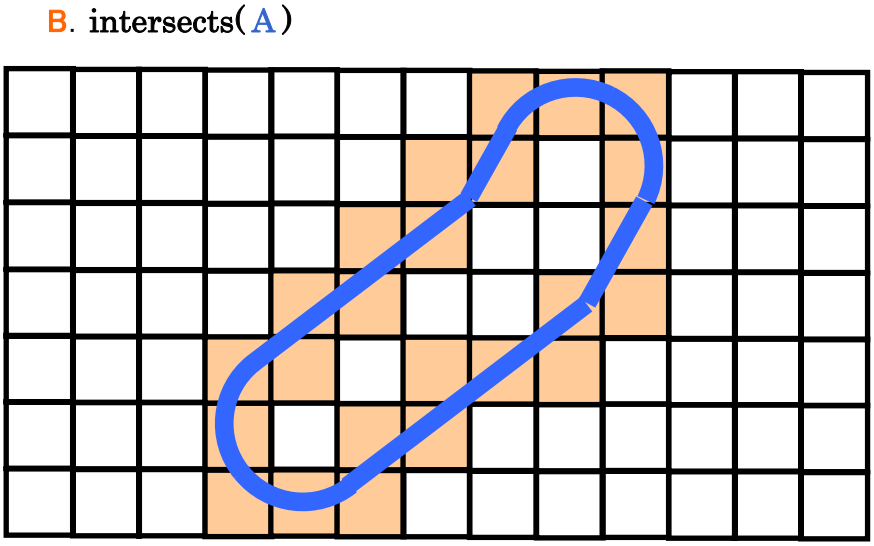


図 3.4.1　検索メッシュにおいて交差するメッシュ

空間インデックスでは，GEOMETRY 型のデータを高速に検索できるように四分木（Quad-Tree）が使われており，図 3.4.2 ，図 3.4.3 の例のようにして空間データを管理している．

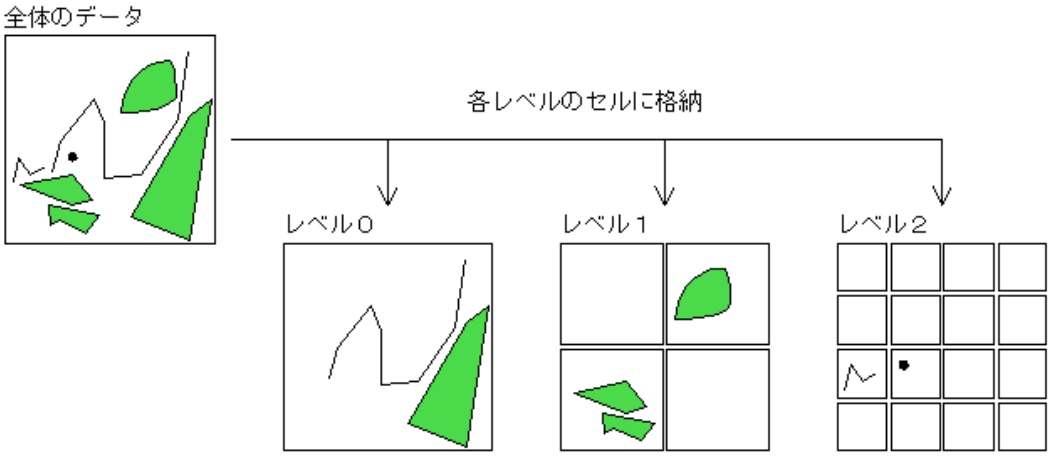


図 3.4.2　各レベルのセルに格納された空間データの例

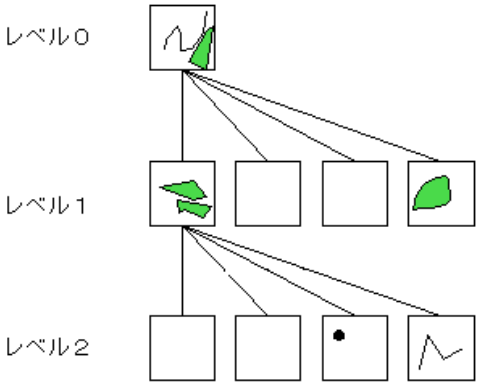


図 3.4.3　各レベルのセルに格納された空間データの例

今回のシステムは，国道データを深さ 0(レベル 0)のセルに格納しており，人口データを深さ 7 のセルに格納している．ただし，人口データに関しては，セルのサイズと人口メッシュのサイズを一致させている．また，国道データの木と人口データの木は別である．

次に，人口メッシュを検索する場合について，①と②を簡略化した図 3.4.4(a)(b)を用いて考える．

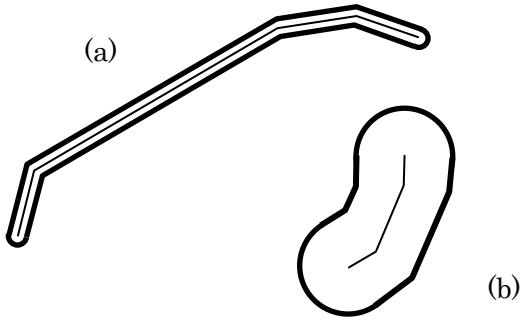


図 3.4.4　簡略化した国道

例として，人口データの空間インデックス四分木の深さを 6 とし，その最深層に人口メッシュが格納されているとする(人口メッシュのサイズと最深層のセルの大きさは等しい)．図 3.4.5 の色付けしている部分のように，最深層までセルの 4 分割を繰り返し，人口メッシュの検索を行う．ただし，分割したセルが検索範囲である国道図形に完全に含まれている場合，最深層のセルでなくてもそれ以上分割せず，そのセルより下の深さの人口メッシュはすべて TRUE を返すものとする．

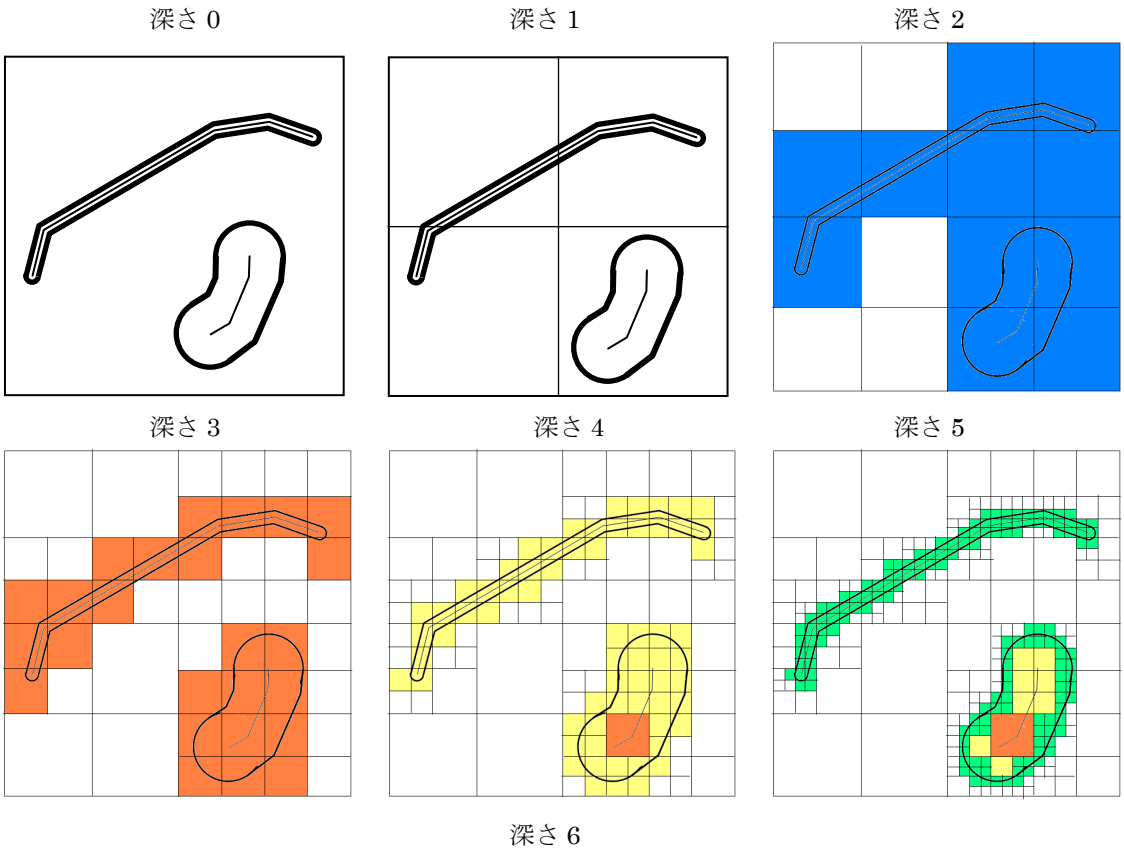


図 3.4.5　人口メッシュ検索方法

この例での結果は以下のようになる。

(a)の場合

	セルの個数	含まれるメッシュ数	計算量
深さ 3	0	$0\times4^3=0$	$0\times3=0$
深さ 4	0	$0\times4^2=0$	$0\times4=0$
深さ 5	11	$11\times4^1=44$	$11\times5=55$
深さ 6	220	$220\times4^0=220$	$220\times6=$
合計	231	264	1320

(b)の場合

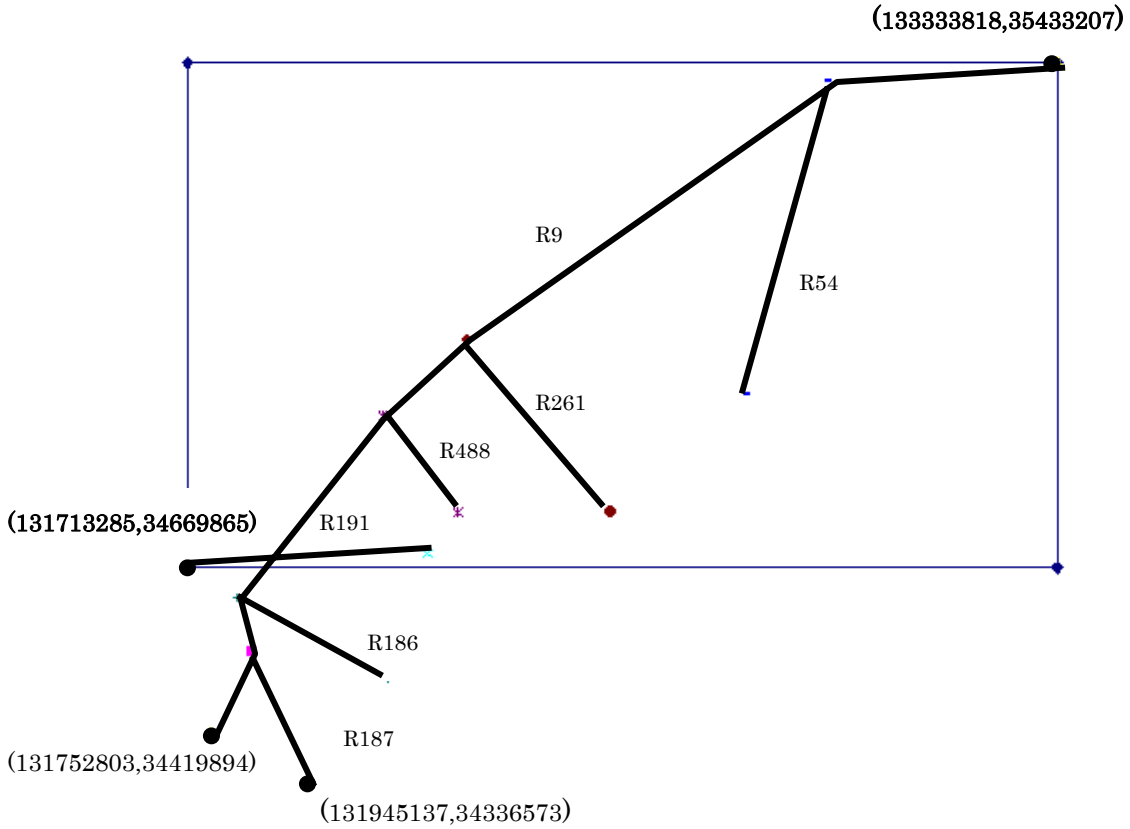
	セルの個数	含まれるメッシュ数	計算量
深さ 3	1	$1\times4^3=64$	$1\times3=3$
深さ 4	8	$8\times4^2=128$	$8\times4=32$
深さ 5	16	$16\times4^1=64$	$16\times5=80$
深さ 6	133	$133\times4^0=133$	$133\times6=798$
合計	277	389	913

結果を見ると，検索メッシュ数は(a)264 に比べて(b)389 の方が 100 以上多いのに対し，計算量は(b)913 より(a)1320 の方が約 1.5 倍多くなっている．ここで，単位メッシュあたりの平均計算量を計算すると(a) 1320/264=5 (b)913/389=2.35 となり，約 2 倍の差があることが分かる．これは表 3.4.1 と似た結果になっている．

二つ目に考えられる原因としては，空間インデックス領域外データである．空間インデックスは次のような SQL 文で作成している．

```
国道座標データへの空間インデックス作成
> CREATE INDEX  GEOIDX
    using type spatial on KOKUDOU ( GEO_CLMN )
    in ( RLOB1 )
    PLUGIN 'EXTENT=131713285,34669865,133333818,35433207;
                DEPTH=1;PRECISION=1';
人口座標データへの空間インデックス作成
> CREATE INDEX  GEOIDXJ2
    using type spatial on JINKOU ( GEO_CLMN )
    in ( RLOB2 )
    PLUGIN 'EXTENT=131672542,34319906,133384834,36336370;
                CELL=131797535,34386565,12499,8332;PRECISION=1';
```

EXTENT で空間インデックス領域の範囲を指定しており，空間座標データの空間インデックス作成領域は以下の四角の範囲である．



図を見ると，最上位となるレベル 0 のセル（インデクス領域の最大範囲）に含められない空間データ(R186,R187,R9 の一部)が存在していることが分かる．これらは，**空間インデクス領域外データ**として空間インデクス領域内の空間データとは別に管理され，空間インデックス領域外データを処理しなければならない検索条件のときに，検索性能が低下することがある．このような検索性能の低下を避けるためには，空間インデックスを作成するときのオプションにできるだけすべての空間データを含むように空間インデックス範囲を設定する必要がある．

次に，図 3.3.1 検索メッシュ数と処理時間の関係に注目する．前述したとおり，空間インデックスで四分木を用いているため，処理時間は一般的に $O(\log n)$ に従うはずであるが，グラフを見ると検索メッシュ数増加に伴い，処理時間は指数関数的に増加しているように見える．この原因は，HiRDB 処理系に依存している可能性もあり，今回行った研究だけで判断するにはデータが少なすぎるため，今後さらに計測データを増やして考察する必要がある．

次に，図 3.3.2 を見ると，DB サーバに強い負荷がかかっていることがわかる．CPU 使用率の増加が顕著に現れている部分は，人口検索 SQL 実行の処理を行っている箇所である．また，人口検索 SQL を実行し，データ検索が終了すると，DB サーバの CPU 使用率は下がり，Web サーバの CPU 使用率は増加する．これは，Web サーバが受け取

った検索結果を用いてJavaサーブレットでGoogle Maps APIを生成しているためである．この結果より，DBサーバを性能の良いマシンにかえれば，検索時間の短縮につながるのではないかと考えられる．

第４章　終論

先行研究で構築されたシステムを用いて，処理単位別に精密な計測を行った．その結果，システムのボトルネックとなっている箇所が人口検索 SQL であることと，検索条件を変更した際，メッシュ数増加に伴い処理時間も増加することが分かった．また，同時に DB サーバと Web サーバのリソースを計測することにより，DB サーバに負荷がかかっていることが分かった．

今後は，表の再構成や空間インデックスオプションの変更などをすることで，システムの最適化を行い，検索処理の高速化を図りたいと考えている．そして，今回得られたデータを基に具体的なベンチマークテストの計画を立て，Web サーバとの性能を総合的に評価できる ORDBMS ベンチマークテストの開発を行いたい．

第５章　謝辞

本研究にあたり，最後まで熱心な御指導をいただきました田中章司郎教授には心より御礼申し上げます．また，同じ研究室の清水洋志さん，坂口隼さん，中山裕太さん，森滝昌志さんには本研究に関しまして，数々の御協力と御助言を頂きました．厚く御礼申し上げます．

なお，本論文，本研究で作成したプログラム及びデータ，並びに関連する発表資料等の全ての知的財産権を本研究の指導教官である田中章司郎教授に譲渡致します．

参考資料・文献

[1]Dacapo benchmarks home page:
http://dacapobench.org/
[2]TPC-App:
http://www.tpc.org/tpc_app/default.asp
[3]木村眞吾 「目的別空間データベースサーバを用いた Web システムの設計と実装」
島根大学総合理工学部 数理・情報システム学科 卒業論文 2008
[4] CRNMonitor1.10
http://www.vector.co.jp/soft/dl/win95/hardware/se331669.html