拡張可能グリッドファイルによる空間データの検索

三好　涼介†　　三浦　孝夫†

† 法政大学 工学部 電気電子工学科　〒 184–8584 東京都小金井市梶野町 3-7-2
E-mail: †{c00d3131,miurat}@k.hosei.ac.jp

あらまし　現在の空間データ処理システムでは、高い検索効率や高度な動的特性、データ分布への非依存のいずれの条件を満たすことは難しい。本論文では、これらの特性を実現する拡張可能グリッドファイル方式を提案し、各種実験によってその有用性を示す。

キーワード　空間データ処理システム , ハッシュ , グリッドファイル

Querying Spatial Data on Extended Grid Files

Ryosuke MIYOSHI† and Takao MIURA†

† Dept.of Elect.& Elect. Engr., HOSEI University　3-7-2, KajinoCho, Koganei, Tokyo, 184–8584 Japan
E-mail: †{c00d3131,miurat}@k.hosei.ac.jp

Abstract　Nowadays we see a variety of spatial applications However, as well-known, it is hard to realize both efficient and dynamic access techniques yet independent of data distribution. In this investigation we propose a sophisticated access mechanism carrying all the aspects and we show some experimental results.

Key words　Spatial data processing system , hash , grid file

1. 前　書　き

近年のデータベースの分野においては、地理データや CAD データなどの空間データを扱う場合が少なくない。現在の空間データ処理システムは SS 木や SR 木などの R 木を基盤とした階層的な空間インデックス構造が最も主流である。R 木 [1] は、空間データ集合を格納すべき最小方形領域（MBR）と呼ばれる超長方形を領域分割方式に従って再帰的、階層的に分割する多分木である。従来の階層インデックス構造と違い MBR の重なりを許容するため、平衡した木となることが知られている。しかし、重なりを許容することによって、質問によっては無駄なノードアクセスが発生したり、動的な挿入、更新時の処理負荷が増加するなどの問題がある。

本論文では、拡張ハッシュ法 [2] と呼ばれるデータ量に応じてキー数を増減できる動的ハッシュ法を、グリッドファイル [3] に適応させた空間インデックス構造（以下拡張可能グリッドファイルと呼ぶ）を提案する。この方式により各検索の効率向上、動的挿入の高度化、データ分布への非依存化が可能となる。本稿では実験によってその有用性を示す。2 章では空間データ処理、3 章では拡張ハッシュの手法、4 章ではグリッドファイルについて述べ、5 章で拡張可能グリッドファイルの構造について論じる。6 章では実験結果を示し，7 章で結びとする。

2. 空間データ処理

2.1 空間データ処理

空間データとは、各次元の相互関係などが情報の中身として意味を持つデータを指し、

（1） データ源情報は必ずしもコンピュータ可読な形になっていない

（2） データ量が非常に大きい

（3） 対象の位置や相互関係を明示的に表現しなければならない

（4） 位置や相互関係を考慮したデータ操作が必要となる

などの性質を持ち、通常のデータとは違った処理が求められる [4]。

特に (2),(3) の処理がデータ構造に求められ、そのとり方によって要求される操作の実行効率に大きな差が生じることもある。従って (2),(3) の最適な処理を行える空間データ構造を構築することが空間データを扱う分野での課題である。

2.2 空間データ表現と空間データ操作

n 次元の空間点データは各次元に対応する n 個の情報とそのデータに付属する情報を併せた $m(m \geqq n)$ 個の情報で表現される。また、空間データを管理する場合必要となる操作としては次のようなものが挙げられる。

（1） 完全一致型検索

各次元のキーをすべて指定して、それに一致するレコードを求める検索。データの存在を調べたり、検索時の興味が付属する情報にある場合に使用される。「東経 133 度、北緯 33 度に駅はあるか」などが使用例である。

（2） 範囲検索

ある範囲を指定して、その範囲に含まれるすべてのレコードを求める検索。一般に検索範囲はすべての軸 a_i について、$LOW_i \leqq a_i \leqq HIGH_i$　という形で指定される。「東経 130 度から 133 度、北緯 30 度から 35 度の範囲にある駅をすべて挙げよ」などが使用例である。

（3） 近傍検索

各次元のキーをすべて指定して、それに最も近い位置にあるレコードを求める検索。「東経 133 度、北緯 33 度から最も近い駅を挙げよ」などが使用例である。

2.3 代表的な手法

主な空間データ処理システムとして R 木、R*木、SS 木、SR 木、GBD 木などが挙げられる。

R*木は R 木において、体積を小さくするような領域分割戦略を使用したものである。そのため範囲検索には優れるが、ある次元の辺長が大きくなる可能性もあるため、近傍検索には向かない。

SS 木はノードの形を球としたものである。径を小さくするような領域分割戦略を使用し近傍検索には優れるが、長方形領域質問の範囲検索では性能が劣化する。

SR 木 [5] は球と長方形を組み合わせ、重なる部分をノードとしたものである。径も体積も小さくなり検索に優れたものとなっている。

総じて R 木と同じように MBR の重なりを許容するため、平衡木にはなるが無駄なノードアクセスの発生やデータ更新の点などで問題を抱えている。

GBD 木 [6] は領域を二等分割していき、その階層を簡約化した木構造で管理する多分木である。簡約化とは、領域式と呼ばれる矩形領域の一次元的な表現方式を各領域となるノードに付与し、片側にしかデータを持たないノードの発生を防ぐものであり、これによりファイルアクセス回数の削減を行っている。しかし木構造で管理しているため、一般的にはバランスをとることが難しいとされる。

3. 拡張ハッシュ

3.1 空間データにおける拡張ハッシュの必要性

ハッシュ法はキーによるレコードのアクセスをもっとも高速に行える手法の一つであるが、キー数が固定されているため空間データ等でのデータ量が大きい場合やレコード数が大きく変動する場合には向かない。そこで考え出された手法が拡張ハッシュ法 [2] である。この手法を使用することによりレコード数に応じてファイルを伸縮させることができる。

3.2 データ表現とデータ操作

拡張ハッシュ法では、ハッシュ関数を使用しキーに対するハッシュ値を得る。

次にハッシュ値を n 進法で表現し上位 P 桁を取り出し、切り出された数がバケットと呼ばれるデータを収納するファイルへのインデックスとなる。このインデックスの配列をディレクトリと呼び、n^P 個の要素からなる。本論文では $n = 2$ とする。

また、ハッシュ値から切り出される桁数 P はバケット数に応じて伸縮する。そして、そのインデックスよりディレクトリが指定され、ディレクトリからアクセスすべきバケットを取り出し各種操作を行う。

図 1 は、ハッシュ値の上位 3 桁をディレクトリへのインデックスとしている。ディレクトリはそれぞれ 000,001 はバケット A、010 はバケット B、011 はバケット C、100,101,110,111 はバケット D へのポインタとなる。ここで $H(X) = 0110010011$ となるデータ X の挿入を考える。いま $P = 3$ であるから $H(X)$ の上位 3 桁を取り出しディレクトリへのインデックスとする。011 はバケット C を指しているのでバケット C を取り出し挿入を行う。

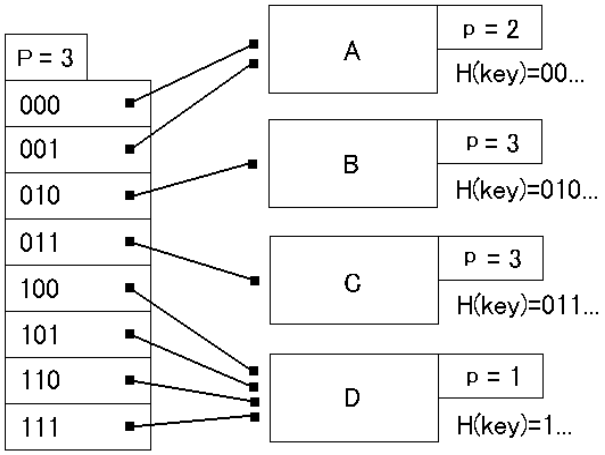

図 1　拡張ハッシュ法

3.3 データ更新による構造変化

一定のバケットへのデータの偏りを防ぐため、バケットにはあらかじめ最大容量 B が決められており一定のデータ数を超えると拡張操作が行われる。このとき複数のディレクトリから参照されているバケット、例えば図 1 のバケット D が一杯になるとバケット自体が分割され、単ディレクトリから参照されているバケット、例えば図 2 のバケット C が一杯になるとディレクトリが拡張される。

したがってバケットにも自身の管理している領域の桁数 p を記憶しておかなければならない。この値は常に $P \geqq p$ であり、そのバケットを指すディレクトリ数は 2^{P-p} である。

3.4 バケット分割操作

バケットの分割の例として図 1 のバケット D があふれた場合を考える。バケット D は $H(X) = 1...$ となるデータを格納する $p = 1$ のバケットである。このバケット D が管理する領域を二等分割し片方を新しいバケット E が管理するように設定する。

ディレクトリ 110,111 をバケット E に設定した後、バケット D の p を 2 とし $H(X) = 10...$ となるデータを D に、またバケット E の p を 2 とし $H(X) = 11...$ となるデータを E に挿入しなおす。図 2 はバケット D を分割後の状態である。

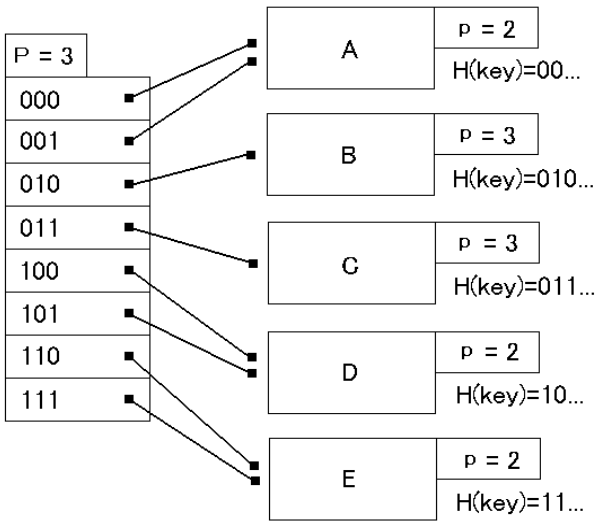

図 2　バケット D 分割後のキー分布

3.5　ディレクトリ拡張操作

ディレクトリの拡張の例として図 2 のバケット C があふれた場合を考える。バケット C は $H(X) = 011...$ となるデータを格納する $p = 3$ のバケットである。このバケット C はディレクトリ 011 からしか参照されておらずこのままでは分割が行えない。そこで P の値、すなわち $H(X)$ より取り出す桁数を増やした後バケットの分割を行う。

ディレクトリを 2 倍に増やし各バケットへの設定を行った後、ディレクトリ 0111 を新バケット F に設定しデータを挿入しなおす。図 3 はディレクトリの拡張及びバケット C の分割後の状態である。

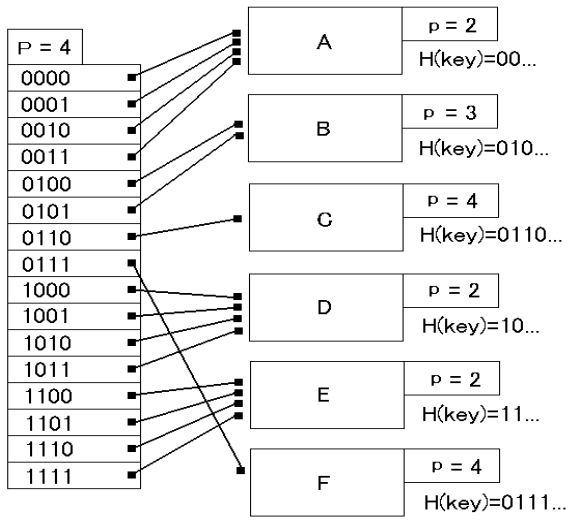

図 3　ディレクトリ拡張後のキー分布

4.　グリッドファイル

グリッドファイル [3] とは、ファイルの負荷率を考慮して n 次元空間を分割していく最も単純な空間インデックス構造の一つである。本稿ではこのデータ構造を基盤とし、本章ではグリッドファイルについて述べる。

4.1　データ表現とデータ操作

まず各次元のキーに対応したグリッドを各軸の情報より得る。そのグリッドよりアクセスすべきバケットが得られ、そのバケットに対して各種操作を行う。

図 4 では各グリッドが所定の領域を管理しており、実際には各グリッド領域に対応したバケットでデータ管理を行っている。ここで $R = (45, 53)$ となるデータ R の挿入を考える。R は $25 \leqq R_x \leqq 50$、$50 \leqq R_y \leqq 100$ のグリッドの管理するバケット内に入れるべきである。従ってこのグリッドの指すバケット D を取り出し挿入を行う。

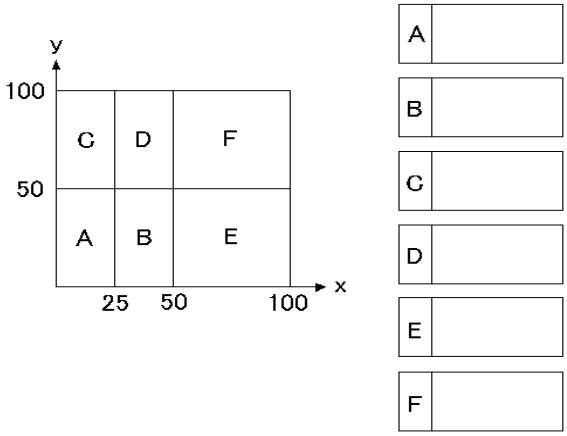

図 4　グリッドファイル

4.2　データ更新による構造変化

バケットにおいてあふれが生じた場合、拡張操作が行われる。あふれの生じたバケットを指すグリッドに対して、グリッドができる限り立方体になる軸を選択して領域を 2 等分割する。分割された片方のグリッドは分割前に指していたバケットを、もう片方のグリッドは新しいバケットを指すように設定する。

4.3　特　　徴

利点はキーに対応するグリッドが得られれば各種操作は効率良く行われ、完全一致型検索や挿入操作では 1 回のファイルアクセス回数となる。問題点はグリッド空間の分割が進むにつれて、キーに対応するグリッドを検索するためにかかる参照回数が非常に大きくなることである。これを解決するため、本稿では拡張ハッシュを適応させる。

5.　拡張可能グリッドファイル

ここでは、前々章で述べた拡張ハッシュを前章で述べたグリッドファイルに適応させた拡張可能グリッドファイルの構造について論ずる。本試作では 2 次元データを扱い、1 レコードは (整数, 整数, 文字列) で表現されるものとする。ディレクトリの総数は $2^{P_x+P_y}$ となり、P 及び p も 2 次元となる。また、範囲検索などでデータ順を考慮しなければならないため、ハッシュは使用しない。データは K 桁の二進数表記とする。

5.1　データ表現とデータ操作

まず各次元の値を n 進法で表現しそれぞれの上位 p_x, p_y 桁

を取り出す。切り出された数の組み合わせがバケットへのインデックスとなる。

そして、そのインデックスよりディレクトリが指定され、ディレクトリからアクセスすべきバケットを取り出し各種操作を行う。また、値から切り出される桁数 P_x, P_y はバケット数に応じて伸縮する。

図 5 は、x 軸, y 軸共に値の上位 1 桁をディレクトリへのインデックスとしている。ディレクトリはそれぞれ (0,0) はバケット A、(0,1) はバケット B、(1,0),(1,1) はバケット C へのポインタとなる。ここで $R = (0110011, 1110101)$ となるデータ R の挿入を考える。いま $P_x = P_y = 1$ であるから R_x, R_y の上位 1 桁を取り出しディレクトリへのインデックスとする。(0,1) はバケット B を指しているのでバケット B を取り出し挿入を行う。

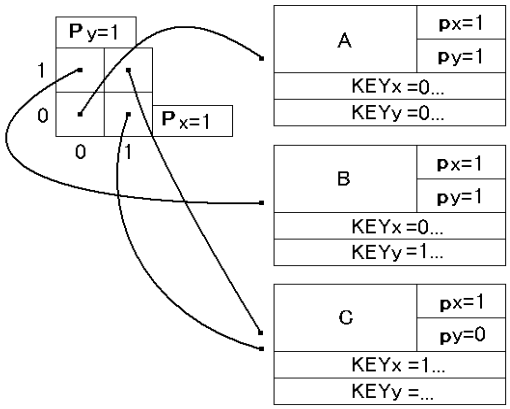

図 5　拡張可能グリッドファイル

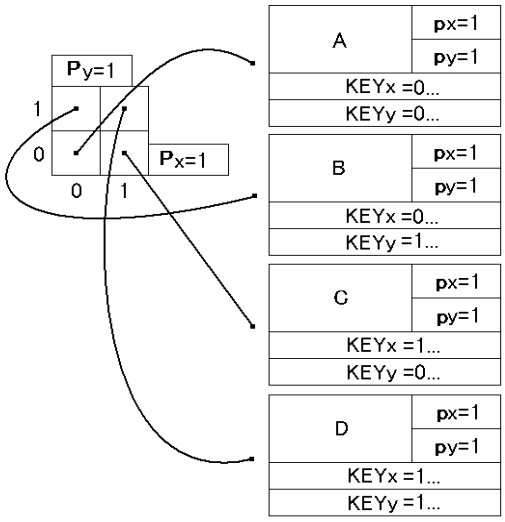

図 6　バケット C 分割後のキー分布

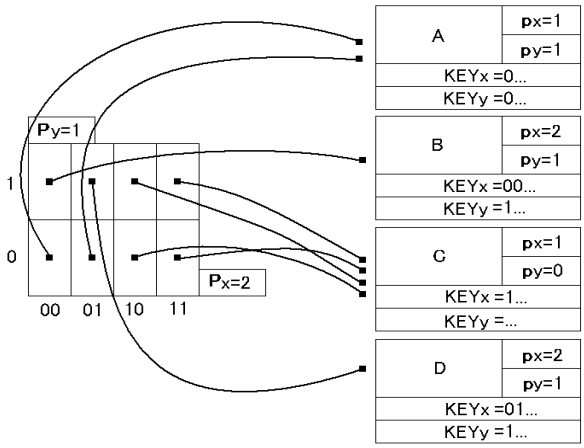

図 7　ディレクトリ拡張後のキー分布

5.2　データ更新による構造変化

バケットにおいてデータ挿入によってあふれが生じる場合、$P_x = p_x$ 且つ $P_y = p_y$ となるバケットのあふれ発生時であればディレクトリ拡張操作が、それ以外のバケットのあふれ発生時ならばバケット分割操作が行われる。手順は 1 次元と同じであるが、2 次元の場合分割する軸の選択が行われる。本論文では、バケット分割時は $P_x - p_x = P_y - p_y$ ならばデータ分布の大きいほうを、それ以外なら $P - p$ の値の小さいほうを分割軸とし、ディレクトリ拡張時は $P_x = P_y$ ならば x 軸を、それ以外なら P の値の小さいほうを分割軸とする。

図 5 のバケット C があふれた場合バケット分割操作が行われ図 6 の状態となり、バケット B があふれた場合ディレクトリ拡張操作が行われ図 7 の状態となる。

5.3　一括生成操作

大規模データにおける一括挿入の手法を論ずる。本研究での一括投入は、あらかじめデータ分布に適する P の値とバケットを作っておくというものである。まずデータに対してデータ数/($B \times$ (1〜1.5)) 程度のデータの代表点を抽出する。次にその代表点で B=1 の拡張可能グリッドファイルをメモリ内で構築し、得られる d の値とバケットを引き継いで本データで挿入を行う。代表点の抽出について、本論文では K-mean クラスタリング手法を使用して抽出する。

5.4　検索の手法

5.4.1　完全一致型検索

検索点のそれぞれ上位 P_x, P_y 桁を取り出しディレクトリへのインデックスとする。これによりディレクトリが指定され、そのディレクトリの指すバケットにアクセスし、バケット内のすべてのデータに対して比較を行う。ディレクトリファイルへのアクセスを入れるとファイルアクセス回数は 2 回となる。

5.4.2　範 囲 検 索

検索方形領域の頂点すべての上位 P_x, P_y 桁を取り出しディレクトリへのインデックスとし、グリッド空間に射影させた範囲にあるディレクトリの指すバケットすべてにアクセスし、バケット内のデータに対して実際の判定を行う。

5.4.3　最近点検索

検索点のそれぞれ上位 P_x, P_y 桁を取り出しディレクトリへのインデックスとし、ディレクトリの指定するバケット内のデータで距離計算を行い仮の最近点 Q を算出する。指定するバケット内にデータがない場合、指定されたディレクトリの周囲のディレクトリの指すバケットにアクセスし、データが存在する

バケットにアクセスするまでディレクトリの検索範囲を拡げていく。Q 算出後、質問点と Q との距離を L とし、質問点を中心とした辺長 $2L$ の正方形領域に対して範囲検索を行い得られたデータより真の最近点 Q' を算出する。

6. 性能評価

前出の特性の有無について調査を行い、拡張可能グリッドファイルの有用性を考慮する。

6.1 実験環境

本稿で行う実験の環境は、OS に FreeBSD4.6.2-RELEASE をインストールし、CPU が Pentium4 1.8GHz、メモリが 256MByte の計算機を使用する。

使用する点データ 119,898 点は、各レコードが 46 バイト長 (2 つの 7 バイト long integer 型項目および 32 バイト character 型からなる) のニューヨーク、ボストン、フィラデルフィア周辺の郵便アドレス 119,989 件である。100000+19898 点、60000+59898 点の二通りに分割し静的、動的投入を行うものとする．また、質問点 10000 点は乱数で、範囲検索時は質問点を中心とした辺長 10000 の方形領域を範囲とする。$K = 20$、$B = 4, 8, 16, 32, 64$Kbyte として実験を行う。

6.2 実験 1

点データ 100000 点と 60000 点で一括生成を行い各検索を実行し、各操作のファイルアクセス回数と実行時間の計測を行う。ファイルアクセス回数については SR 木との比較を行う。実験結果を表 8,9 に示し、グラフを付録の図 8,9 に示す。

		100000		60000	
com	B	SR 木	EGF	SR 木	EGF
LO	4	7827	207479	4625	124811
	8	3737	203737	2245	122542
	16	1863	201683	1121	121258
	32	929	200930	559	120613
	64	467	200437	281	133250
EQ	4	24712	20002	16973	20002
	8	19274	20002	18324	20002
	16	20267	20002	19819	20002
	32	14129	20002	13405	20002
	64	14267	20002	13455	20002
RQ	4	35532	28382	23553	25815
	8	25911	25677	22471	24040
	16	24740	23909	22736	22926
	32	17041	22879	15243	22170
	64	16199	22156	14756	25155
NNQ	4	47269	356193	39914	467395
	8	35709	197358	37610	280873
	16	34395	135099	35839	170746
	32	22789	102531	24295	127651
	64	22365	89645	23297	449776

LO:一括生成 EQ:完全一致型検索
RQ:範囲検索 NNQ:最近点検索
表 1　実験 1 ファイルアクセス回数

		100000			
com	B	ut[s]	st[s]	rt[s]	I/O
LO	4	6.491	4.641	31.86	738
	8	2.922	4.196	14.98	1385
	16	2.337	3.792	9.64	2530
	32	2.467	3.569	7.71	5293
	64	1.892	3.809	6.76	10478
EQ	4	0.265	0.474	10.16	1693
	8	0.524	0.339	6.09	1047
	16	0.889	0.383	4.16	570
	32	1.783	0.306	4.35	461
	64	3.213	0.478	4.85	373
RQ	4	1.026	0.69	13.86	2345
	8	1.217	0.541	7.76	1189
	16	2.079	0.345	5.46	586
	32	3.244	0.53	5.97	462
	64	5.804	0.427	7.56	373
NNQ	4	86.93	7.465	106.55	2328
	8	25.158	3.318	34.11	1186
	16	17.681	2.34	22.9	585
	32	17.284	1.939	21.22	462
	64	22.444	1.835	25.39	373
		60000			
LO	4	5.322	2.589	20.7	6655
	8	1.952	2.422	9.29	3517
	16	1.411	2.335	5.74	1773
	32	1.407	2.16	4.58	927
	64	32.189	3.873	44.61	8730
EQ	4	0.24	0.403	6.43	1127
	8	0.422	0.312	4.21	703
	16	0.713	0.28	2.86	387
	32	1.432	0.352	2.99	300
	64	0.236	0.455	7.1	1217
RQ	4	0.717	0.634	8.88	1509
	8	0.874	0.448	5.01	779
	16	1.424	0.315	3.58	393
	32	2.544	0.397	4.24	300
	64	1.094	1.032	8.86	1453
NNQ	4	111.547	9.787	128.73	1499
	8	38.205	4.718	46.62	774
	16	24.98	2.576	29.33	392
	32	27.17	2.334	30.76	300
	64	420.728	29.57	458.76	1357

ut:ユーザー時間 st:システム時間
rt:実行時間
表 2　実験 1 実行時間及び I/O 回数

6.3 実験 2

実験 1 で構築した一括生成ファイルに対し、それぞれ残りの点データ 19898 点と 59898 点を投入後各検索を実行し、各操作のファイルアクセス回数と実行時間の計測を行う。ファイルアクセス回数については SR 木との比較を行う。実験結果を表 10,11 に示し、グラフを付録の図 10,11 に示す。

また、実験 1，2 の検索時におけるディレクトリ参照回数を $B = 4, 32$ で算出しており、実験結果を表 12 に示し、グラフを

		19898		59898	
com	B	SR 木	EGF	SR 木	EGF
AD	4	484561	41934	1315884	127597
	8	447589	40849	1138817	123578
	16	551723	40366	1291733	121654
	32	574524	40030	1167804	120695
	64	441585	39876	1026931	127688
EQ	4	28404	20002	28734	20002
	8	20838	20002	21945	20002
	16	22810	20002	24124	20002
	32	23324	20002	26381	20002
	64	15380	20002	15660	20002
RQ	4	42068	29120	43319	29051
	8	29653	26111	31142	26050
	16	29022	24221	30544	24226
	32	27580	23059	30722	23077
	64	18164	22259	18529	28453
NNQ	4	53149	357986	55001	356296
	8	37802	195968	38535	191690
	16	38463	135232	41572	134644
	32	37322	101624	38716	104399
	64	23195	89267	23630	372394

AD:動的挿入

表 3　実験 2 ファイルアクセス回数

付録の図 12 に示す。

6.4 実　験　3

実験 1，2 での最低の $B = 4$ と最高の $B = 32$ を除いた B の値の平均値に最も近い $B = 16$ に対して人工データ 119,898 点で各操作を行う。実験結果を表 13,14 に示し、グラフを付録の図 13,14 に示す。

6.5 考　　察

各実験についての考察を行う。

実験 1 では各結果より各操作の一定以上の速度の確保が確認できた。ファイルアクセス回数では、SR 木に対して $B = 4$ の 100000 点の完全一致型検索で 23 %の性能向上が見られる。しかし、最近点検索では SR 木に対して性能劣化が見られる。これは、最近点検索では n 次検索バケットが空の場合指定ディレクトリの周囲のディレクトリの指すバケットにアクセスするため、$n+1$ 次検索でのアクセスバケット回数は最大で n 次検索より $8n$ 回増加する。またその後の範囲検索でも検索範囲が広くなるためと思われる。実際の I/O 回数はファイルアクセス回数より少なく、最大は $B = 4$ の 60000 点の最近点検索で 0.0032 倍のアクセス回数で行われている。OS 内のシステムバッファの働きと思われる。

実験 2 では各結果より高い動的特性の保持を確認できた。ファイルアクセス回数では、動的挿入で SR 木に対してすべての結果で性能向上が見られ、最大 $B = 32$ の 19898 点で 1435 %の性能向上が見られる。また、各種検索においても SR 木に対して最近点検索以外では 4〜45 %の性能向上が見られる。実行時間でも動的投入において一定以上の速度が確保され、さらに実験 1 の実行時間及び I/O 回数での各検索の計測結果と大

		19898			
com	B	ut[s]	st[s]	rt[s]	I/O
AD	4	0.608	0.884	5.44	2452
	8	0.484	0.819	3.51	1234
	16	0.498	0.767	2.24	676
	32	0.444	0.781	1.81	339
	64	0.422	0.719	1.51	192
EQ	4	0.361	0.416	10.93	1862
	8	0.517	0.413	7.00	1151
	16	1.01	0.347	4.52	641
	32	2.001	0.345	4.85	535
	64	3.829	0.42	5.62	429
RQ	4	1.102	0.765	15.74	2635
	8	1.264	0.686	8.91	1337
	16	2.209	0.46	6.08	666
	32	3.858	0.449	6.87	536
	64	6.722	0.557	8.64	429
NNQ	4	89.431	7.465	110.5	2623
	8	26.285	3.302	36.05	1336
	16	19.055	2.232	24.57	666
	32	18.848	1.673	22.91	536
	64	26.754	1.607	29.66	429
		59898			
AD	4	2.679	2.998	16.52	8335
	8	1.904	2.541	9.22	4127
	16	1.733	2.3	6.48	2076
	32	1.381	2.521	5.41	1088
	64	3.156	3.311	18.34	8567
EQ	4	0.363	0.416	11.03	1862
	8	0.544	0.396	6.76	1143
	16	0.985	0.333	4.4	637
	32	1.817	0.466	4.67	538
	64	0.394	0.433	11.86	1965
RQ	4	1.168	0.7	15.61	2600
	8	1.487	0.473	8.52	1319
	16	2.115	0.493	5.78	659
	32	3.701	0.528	6.68	539
	64	1.46	1.218	15.92	2559
NNQ	4	90.029	7.345	110.59	2593
	8	25.713	3.259	35.33	1319
	16	18.9	2.077	24.28	659
	32	19.292	1.959	23.57	539
	64	336.589	21.954	372.41	2478

表 4　実験 2 実行時間及び I/O 回数

差がなく、最大でも $B = 4$ の 59898 点の範囲検索で 11 %の性能劣化が見られるのみとなっている。

また、ディレクトリ参照計測よりバケット容量 B に最適解があることが確認できた。最近点検索において SR 木に対して、B=32 では 0.32 倍の参照回数であるが、B=4 では 79 倍の参照回数となった。バケット容量が小さいとディレクトリ数が増加しその結果ディレクトリ参照回数も増加するためであり、バケット容量が十分な値であると SR 木より性能向上すると考えられる。

		100000+19898		
	B	L-EQ	L-RQ	L-NNQ
SR 木	4	602763	922155	1444423
	32	5521588	6150510	7390050
EGF	4	20002	362557	73398005
	32	20002	48395	2281265
		A-EQ	A-RQ	A-NNQ
SR 木	4	861271	1235611	1797134
	32	3904547	4680340	7218990
EGF	4	20002	363295	71163357
	32	20002	48575	2208231
		60000+59898		
		L-EQ	L-RQ	L-NNQ
SR 木	4	710218	908286	1621786
	32	3513776	3910280	5854090
EGF	4	20002	359790	127428576
	32	20002	47686	4217504
		A-EQ	A-RQ	A-NNQ
SR 木	4	812704	1183911	1704823
	32	4757160	5565700	7378890
EGF	4	20002	363226	71533048
	32	20002	48593	2305733

表 5　実験 1,2 ディレクトリ参照回数

	100000+19898			
	LO	EQ	RQ	NNQ
SR 木	1863	29676	36203	31921
EGF	201847	20002	25381	61923
	AD	EQ	RQ	NNQ
SR 木	535039	33744	42879	36917
EGF	40001	20002	25436	61735
	60000+59898			
	LO	EQ	RQ	NNQ
SR 木	1121	29642	34684	31861
EGF	120723	20002	24817	78459
	AD	EQ	RQ	NNQ
SR 木	1369467	32958	41994	36172
EGF	121240	20002	25001	61429

表 6　実験 3 ファイルアクセス回数

実験 3 ではデータへの非依存性の保持を確認できた。最近点検索以外では実験 1，2 と大差はないが、最近点検索では 2 倍程度の時間短縮が行われた。これは人工データの一様分布に起因し、空バケットが少なくなることにより検索時間が短縮したと思われる。しかし、データによる誤差の範囲内であり、データへの非依存性の保持を確認できたといえる範囲内である。

7. 結　　び

本論文では拡張可能グリッドファイルの手法を提案した．そしてその性能を各実験により評価し，その有用性を示した．今後の課題として、膨大なデータ量を想定しひとつの計算機のみに特化せず、複数の計算機による分散環境下での拡張可能グリッドファイルの構築などが挙げられる。

	com	ut[s]	st[s]	rt[s]	I/O
100000+198098	LO	2.591	5.089	32.65	3008
	EQ	1.803	0.458	6.08	754
	RQ	3.592	0.76	8.24	774
	NNQ	5.601	1.117	10.72	764
	AD	0.391	1.03	17.27	768
	EQ	2.119	0.493	6.39	754
	RQ	4.439	0.598	8.73	774
	NNQ	6.73	0.999	11.62	764
600000+598098	LO	1.239	2.857	9.03	1729
	EQ	1.97	0.367	5.03	510
	RQ	3.528	0.5	6.47	538
	NNQ	6.037	0.887	9.4	529
	AD	1.524	3.134	35.33	1812
	EQ	2.221	0.407	6.32	689
	RQ	4.248	0.729	8.6	717
	NNQ	6.779	1.023	11.55	706

表 7　実験 3 実行時間及び I/O 回数

謝　　辞

本研究の一部は文部科学省科学研究費補助金 (課題番号 14580392) の支援による．

文　　献

[1] A. Guttman: "R-trees: A dynamic index structure for spatial searching" , proc. Int. Conf. ACM SIGMOD '84, pp.47-57,1984
[2] T.R. ハーブロン, 遠山元道 (訳): "ファイルシステム" , 啓学出版,1992
[3] J. Nievergelt , H, Hinterberger: "The Grid File: An Adaptable, Symmetric Multikey File Structure" , ACM TODS. vol. 9, pp.38-71,Mar.1984
[4] 坂内正夫, 大沢裕: "画像データベース" , 昭晃堂,1987
[5] 片山紀夫, 佐藤真一: "マルチメディア情報の大規模処理に向けた多次元インデクシング手法の応用" , 電子情報通信学会論文誌 (D-II),vol.J82-D-II,no.10,pp.1606-1616,Oct,1999
[6] 大沢裕, 坂内正夫: " 2 種類の補助情報により検索と管理性能の向上を図った多次元データ構造の提案" , 電子情報通信学会論文誌 (D-I),Vol.J74-D-I,No.8,pp.467-475,1991

付　　録

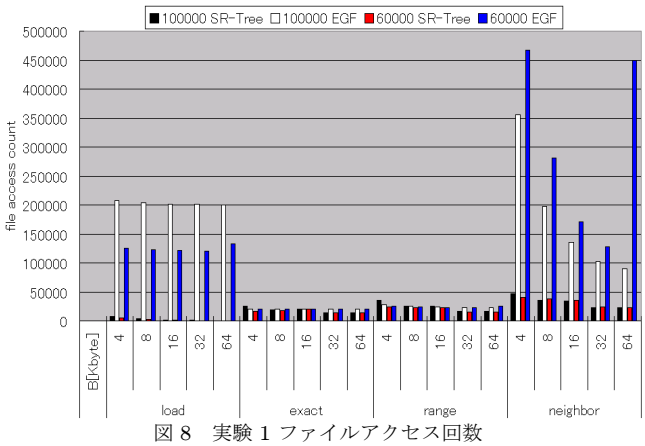

図 8　実験 1 ファイルアクセス回数

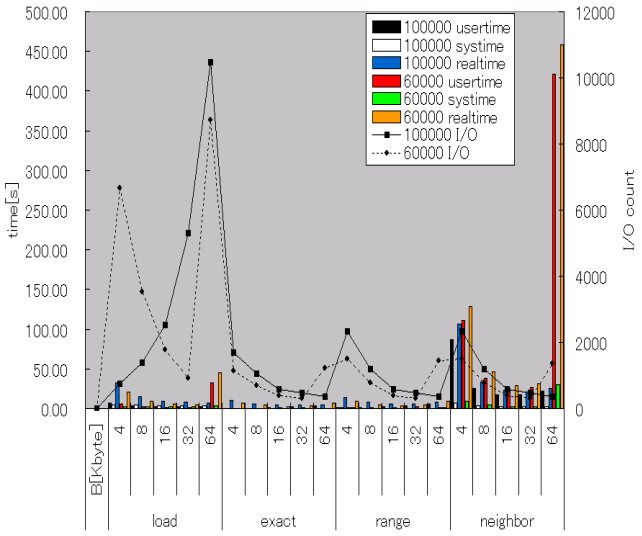

図 9　実験 1 実行時間及び I/O 回

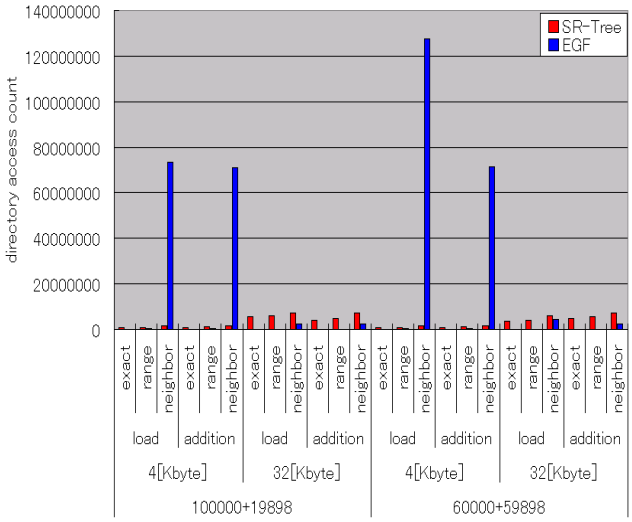

図 12　実験 1,2 ディレクトリ参照回数

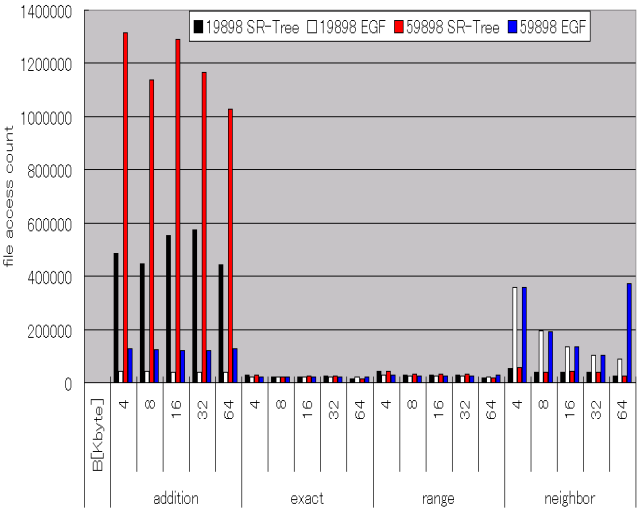

図 10　実験 2 ファイルアクセス回数

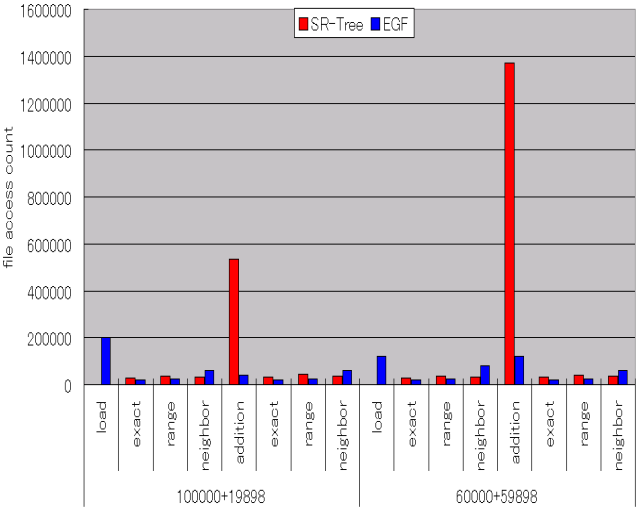

図 13　実験 3 ファイルアクセス回数

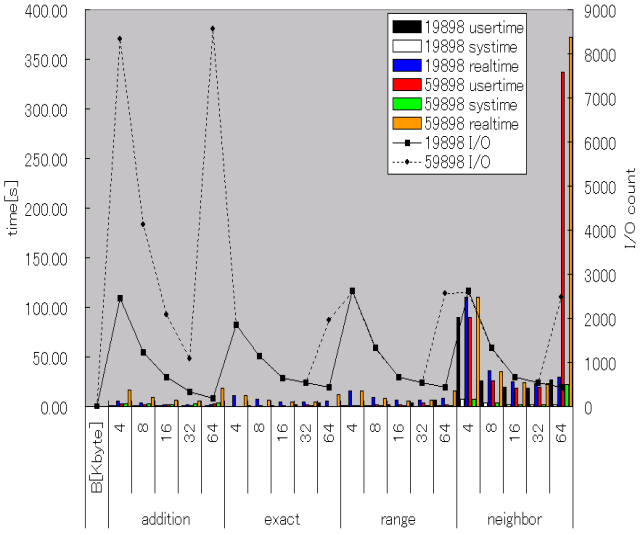

図 11　実験 2 実行時間及び I/O 回数

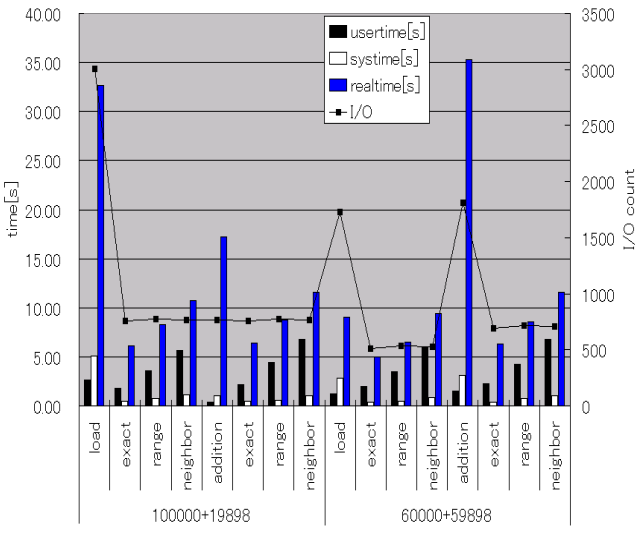

図 14　実験 3 実行時間及び I/O 回数