他のレコードを更新せずに順序を記録する手法

新居　雅行

〒336-0922 さいたま市緑区大牧 1495-3
E-mail: nii@msyk.net

あらまし　レコードの順序をバイナリ値で記録する方法とその改良について提案する。データベースのレコードに並べ替えの手がかりになるフィールドがない場合に、レコードの順序を記録する用途を想定する。VLEI コードを利用したバイナリ値を使用することで、他のレコードの変更を伴わずに順序を記録でき、2つのバイナリ値の中間値は必ず存在する。一度生成した順序を示すバイナリ値は不変である。ただし、VLEI コードは同一のレコードの直前や直後、先頭あるいは末尾へのレコードの連続挿入によってバイナリ値の桁数が増大するが、空間をあらかじめ用意する手法によって最大桁数を抑える手法を提案する。

キーワード　順序、バイナリ、データベース、2 分木

Order Preserving Binary Code for RDB

Masayuki Nii

1495-3 Omaki, Midori-ku, Saitama-city, SAITAMA, 336-0922, Japan
E-mail: nii@msyk.net

Abstract　Some kinds of application need to determine the order of each record not using the field data, for example, a user arranges photographs as he/she wants. In that case, VLEI code is available to express the order of each record, and labels generated by VLEI code are immutable for inserting records to any positions. Though VLEI code produces many digit binary for continuous inserting to the top or end of the record sequence, the improved method to reduce digits in case of continuous inserting is proposed.

Keyword　Order, Binary, Database, Binary Tree

1. はじめに

リレーショナルデータベースでレコードを並べ替える場合、並べ替える手がかりとなるフィールドを指定して並べ替える。例えば、「得点」や「申込日」といった数値情報のフィールドをもとに並べ替えを行う。こうした手がかりとなるフィールドがない場合に、何らかの基準でレコードの順序を記録したい場合がある。レコードに記録されたデータとは独立して順序を記録したい場合、リンクリスト形式を用いるか、番号を記録するフィールドを別途設けるといった手法でアプリケーションが作られる事が一般的である。リンクリストの場合は最初のレコードへのリンクを保持しつつ、レコードの追加において他のレコードの修正が必要になる。番号を振る場合、レコードの挿入などで番号の振り直しが必要になる。他のレコードに対して影響することなく、一度振ったラベルを修正する必要がなく、また全体的な振り直しも不要な手法があれば、データベースへの記録等で有用である。本稿では、それを実現する手法をもとに、データベースアプリケーションでの実用が可能な手法を提案する。

順序を記録するための手法に利用できる手法として有理数をバイナリで表現して辞書順にエンコードする LCF という手法が提案されている[1][2]。また、バイナリ列を利用することで順序を記録する VLEI コードが提案されている[3]。数値を利用する方法では中間値を求める場合に数値計算が必要となるため、桁数の増加を考えれば実装上の困難さがある。一方、バイナリを利用する方法では中間値を求める手法の方が単純なため実装をしやすい。しかしながら、同一のレコードの直後にレコードを追加するような場合や、先頭や末尾に連続してレコードを追加した場合、最悪の場合はレコード数と同数の桁数増加が発生し多大な記憶領域を使う点が挙げられている[4]。本稿では、VLEI コードが 2 分木として表現可能な点に注目し、ある2つのラベルの中間値となるラベルを求める方法を求める。さらに桁数が増大する傾向のあるようなレコード挿入に対する手法を提案する。

2. 順序記録を必要とする用途

順序を記憶するためのラベルを利用する事で、構築が容

易になる例として次のようなものがある。1 つは、機械的に順序が決まらないようなリストである。多くの場合はデータから求められるのではなく、ユーザが順序を判断しそれを記録したいような場合である。例えば、短いメモを記録したとき、利用者が任意の順序で並べておきたい場合や、写真をユーザが意図した順序で並べるような場合がある。別の例として、一連のデータが同時に存在しない場合や、同列に得られないような場合で、まとめて並べ替えが難しい場合や処理時間がかかる場合である。一部のデータを残す必要はあるが、データが発生した時点でラベルをつける事によって、ソートに必要なラベルをデータに付与することが可能である。また、データが複数のユーザや複数のプロセスで共有されている場合、変更が不要なラベルの利点は大きい。もし、順番を整数で記録して時々振り直すということをした場合、すでに別ユーザや別プロセスでデータをロードしていれば、再度のロード等複雑な処理が必要になる。変更不要なものの場合はそうした点を考慮しなくても良く、新たなレコードの発生や削除で、それら変更のあるレコードの更新だけで済む。この点から、O/R マッピングを行っているような状況での順序の記録方法として、複雑な処理を避ける事ができる。

順序を記録できることを利用すること、階層関係にあるレコードを 1 つのテーブルにフラットに記録する手法が実現する[5]。この方法により、たとえば、メッセージにコメントを階層的に付加するような掲示板システムや、あるいは電子メールの引用先が下位の項目となるようなデータの記録が可能となる。順序と階層を記録するため、それぞれフィールドを確保すれば良い。例えば、数多くのメールをデータベース化するような場合には、順序を記録する手法として、前記のような一度設定すれば変更しないでも済むラベルは有効であると同時に、桁数が極端に増大しないことでの処理時間の平均化はシステム設計上望まれる機能となる。

3. バイナリ値の生成手法

3.1. VLEI コードと 2 分木表現

順序を記録するために、レコードにラベルを追加する。このラベルはバイナリ形式で記述する。集合 $B=\{0, \phi, 1\}$ があり、順序関係として、$0<\phi<1$ という順序関係が成り立っているとする。ϕ は空を意味する。そして、この集合 B の直積である $B\times B\times\cdots\times B=B^n$ によって順序を記録する。このとき、辞書順（lexicographic）で直積の各要素の順序を評価するものとする。すなわち、$x, y\in B^n$ における順序関係を次のように定義する。

$\exists m\leqq n, x_i = y_i\ (i=1,\ldots,m-1), x_m\neq y_m$ の場合
$x_m>y_m$ であれば、$x>y$
$x_m<y_m$ であれば、$x<y$

ここで示したコード化の手法は VLEI コードとして提唱されている[3]。

記録可能なバイナリは 0 ないしは 1 が連続するものであり、ϕ は記録できない。ϕ の存在が可能なのは、バイナリの末尾に存在するとみなす場合である。ゆえに、B^n のすべての要素が記録できる訳ではない。B^2 の場合は 9 つの要素（00、0ϕ、01、$\phi 0$、$\phi\phi$、$\phi 1$、10、1ϕ、11）を生成できるが、このうちそのままをバイナリとして記録可能なものは、00、01、10、11 である。また、ϕ で終わるものについては、空白であるとみなせば、$0\phi=0$、$1\phi=1$、$\phi\phi=\phi$ の 3 つも記録可能である。これらを定義した辞書順に並べれば、00、0、01、ϕ、10、1、11 となる。そして、同様な手法で桁を拡張して、バイナリ値を生成する。結果的に ϕ に続く要素は存在し得ないので、1 つの要素から、0 ないしは 1 を続けた新たな要素が B^n においてバイナリ値として存在可能なものとなる。

集合 B を順序関係に基づいたハッセ図（上下方向が順序[6]）で記述すると図 1(i) のようになる。B×B を構成する場合、直積は図 1(ii)のように表現できる。ϕ に続く要素は存在し得ないのであれば、最初の桁が 0 および 1 に対して B の要素が続くバイナリを生成できる。結果的に B を図 2(i)のように表現し、それに対する直積の図を描くことで図 2(ii)のように B×B は高さ 2 の 2 分木として表現できる。

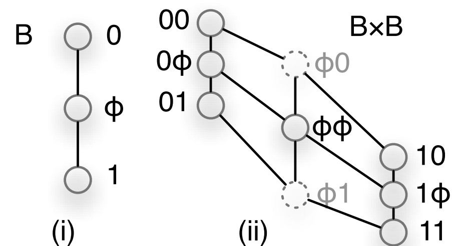

図 1 集合 B と B × B のグラフ表現

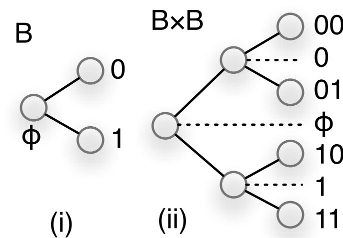

図 2 2 分木への転換

図 3 には高さ 3 の 2 分木を記述した。順序を示すためのラベルは、2 分木中のそれぞれの節点となり、それぞれバイナリ値として表現可能である。なお、ハッセ図上での記述では順序の上で隣接するもの同士を線で結ぶので、図 3(ii)が正しい図となるが、取り得る値の生成過程を考える場合には 2 分木での表現は全体像を捉えるには分かりやすい表現と言える。

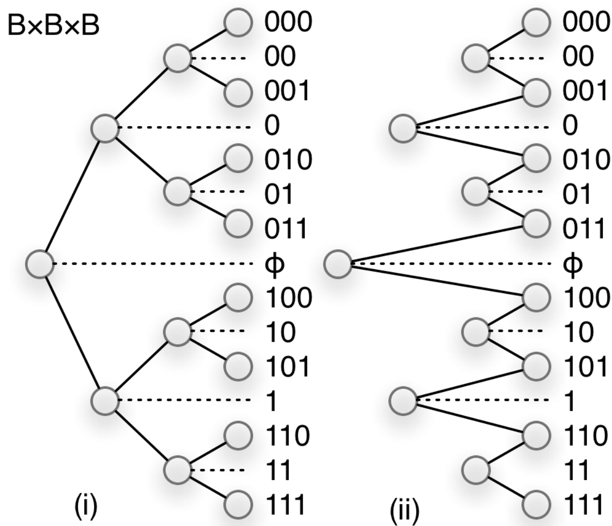

図 3 B×B×B の表現と順序に応じたグラフ

3.2. バイナリ値の中間点が必ず存在する

ある一連のレコード $R=\{R_1,\ldots, R_n\}$ に対して VLEI コードに基づく順序を示すバイナリ値 B_i $(i=1,\ldots, n)$ が割り当てられている場合を考える。あるレコード R_x を任意の順序の位置に追加したいとする。そのレコードに与える順序を示すバイナリ値 B_x を求める。

順序の上でレコード R_m $(1\leqq m<n)$ の直後にレコードを挿入する場合、R_m と R_{m+1} の間の順序となるように、$B_m=(x_1,\ldots,x_j)$ と $B_{m+1}=(y_1,\ldots,y_k)$をもとに新たなバイナリ値 $B_x=(z_1,\ldots,z_l)$を求める必要がある。j と k は同一とは限らない。

x_i と y_i を最初から順番に比較して、最初に異なる値となる桁数を p とする。桁が 0 でも 1 でもない場合は ϕ と考えられる。なお、B_m と B_{m+1} は $B_m<B_{m+1}$ の関係があり、$B_m\neq B_{m+1}$ である。このとき、x_p と y_p については、

(i) $x_p=0$ かつ $y_p=1$

(ii) $x_p=\phi$ かつ $y_p=1$

(iii) $x_p=0$ かつ $y_p=\phi$

の 3 通りしか存在しない。他の組み合わせについては $B_m<B_{m+1}$ という前提が成り立たなくなる。

(i)の場合は $z_1,\ldots,z_{p-1}$ には $x_i(=y_i)$を、z_p には集合 B の定義より 0 より後で 1 より前となる ϕ を採用する。こうして B_x を求めることができるため、B_x は存在する。

(ii)と(iii)については、B_m と B_{m+1} を 2 分木上の節点として考えれば、(ii)の場合は B_m、(iii)の場合は B_{m+1} の子を根とする部分木の中にあるいずれかの節点が、(ii)の場合の B_{m+1}、(iii)の場合の B_m に相当する(図 4)。(ii)と(iii)は、2 分木の上か下かの違いがあるだけで対称的であり結果は同じとみなし、(ii)の場合のみを検討する。

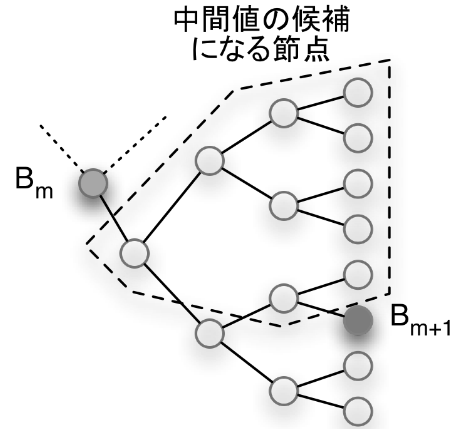

図 4 中間値は節点からの選択

定義より、j<k となり、k-j の高さの部分 2 分木にのみ注目すれば良い。この部分 2 分木の範囲内に $B_m<B_x<B_{m+1}$ を満たす B_x に相当する節点が(a)ある場合と(b)ない場合が考えられる(図 4)。

手順としては、この部分 2 分木に間順走査を行い、得られた節点に対して、$B_m<B_x<B_{m+1}$ を満たしているかどうかを調べれば良い。

(a)の「節点がある場合」はそれを中間値として採用できる。(i)は結果的に(ii)および(iii)の(a)の「節点がある場合」と同じ手続きで中間値を探す事ができる。p 桁目を根とした部分木の中にある節点から、$B_m<B_x<B_{m+1}$ を満たすものは中間値として採用できる。複数の節点がある場合にどの節点を採用するのが適切なのかという問題があるが、ここでは長さが最小のものを選択することにする。この問題はセクション 4 で議論する。

(b)の「節点がない場合」については、さらに部分木の高さを高くする事で、中間点は存在する。j<k の場合は、

$$B_x = (y_1,\ldots,y_k, 0)$$

が中間値となる。なぜなら、B_x は B_{m+1} と同一の部分を持つので、$B_m<B_x$ となる。そして、最後に 0 を付加することで、$B_x<B_{m+1}$ となる。結果的に $B_m<B_x<B_{m+1}$ が成り立つ。

j>k の場合は、以下のような B_x が中間値となる。証明は省略する。

$$B_x = (x_1,\ldots,x_k, 1)$$

結果的に、桁数の長い方に、0 ないしは 1 を追加することで、中間のバイナリ値が得られることを意味する。

3.3. 先頭と末尾について

リストの先頭や末尾にレコードを追加する場合のバイナリ値を求める場合は次のような手法で可能である。先頭に追

加する場合、先頭のレコードのバイナリ値 B_1 が q 桁であるとすると、0 が q 桁並ぶバイナリ値(仮の最小値)との中間値を求める。同様に末尾に追加する場合、最後のレコードのバイナリ値 B_n が r 桁の場合、1 が r 桁並ぶバイナリ値(仮の最大値)との中間値を求める。このとき、B_1 と仮の最小値、あるいは B_n と仮の最大値の間に節点が存在すれば、それらは新たなレコードのバイナリ値として利用できる。節点が存在しない場合は、B_1 に 0 を付加するか、B_n に 1 を付加する。

なお、B_1 がすべて 0、あるいは B_n がすべて 1 から構成される場合は「節点が存在しない」として、B_1 ないしは B_n に対して、0 ないしは 1 を続ける。

例として、000101 の前の値は 00010 などが割り当てられる。ところが、000 の前となると 0000 となり、その前のレコードは 00000 と、先頭や末尾に連続的にレコードが追加された場合のバイナリ値はレコード数の増加と桁数の増加が同一になってしまうという性質がある。この解決方法については後述する。

まったくレコードがないテーブルに最初のレコードを追加する場合、バイナリ値は φ を採用すれば良い。

以上の手続きで求められるバイナリ値は順序関係を持ち、新たにレコードを追加する場合でも、既存のレコードのバイナリ値を変更する必要はない。レコードの削除の場合でも、順序関係がある集合の要素を削除することと同等と考えれば、順序関係は削除後も存在する。

4. バイナリ値の格納方法

バイナリ値を直接記録することも処理系によっては可能であるが、バイナリ値を ASCII コードの文字でエンコードし直す方が、データベース上で扱いやすい場合もある。バイナリ値の性格と、一般的なデータベースでの処理より、以下の手順で作った文字列をデータベース上に記録することを提案する。

1. 比較に利用するバイナリ値の最大桁数 s を求める
2. バイナリ値を最初から 4 桁ずつ、s 桁まで取り出す
3. 存在しない桁は φ であると見なす
4. 1 つの桁を 3 進数の 1 桁と見なす
5. 0, φ, 1 をそれぞれ 3 進数の 0, 1, 2 とみなす
6. 4 桁のバイナリ (x_1, x_2, x_3, x_4) について、
   $T=t_1 \times 3^3+t_2 \times 3^2+t_3 \times 3+t_4$
   を求める
7. T の値に応じて数字ないしはアルファベットを割り当てる(表 1 が対応表で T 列の数値に対して C 列の文字を割り当てることを意味する)

表 1 バイナリ値と文字の対応表

T	C	T	C	T	C
0	0	22	B	62	M
1	1	24	C	67	N
2	2	25	D	72	O
4	3	26	E	73	P
6	4	40	F	74	Q
7	5	54	G	76	R
8	6	55	H	78	S
13	7	56	I	79	T
18	8	58	J	80	U
19	9	60	K		
20	A	61	L		

表 1 の T の値は連続していないが、これはたとえば φ1φ0 (T=48) のようなバイナリが存在しないからである。結果的に 4 桁のバイナリで取り得る値は $2^4+2^3+2^2+2^1+2^0=31$ 個となるので、数字とアルファベットの大文字でまかなえる。

バイナリ値から上記のルールに沿った文字列への変換によって順序関係は保持される。この文字列をフィールドに記録し、このフィールドで並べ替えを行う事で、順序を文字列で記録することができる。数字とアルファベットであれば、実際に使用するシステムの上での並べ替えや比較の上での問題は少ないと考えられる。

単にバイナリ値をエンコードするのではなく、最大桁数を求める必要がある。なぜなら、バイナリ値での φ は、生成した文字列での空には対応しないからである。たとえば、000001 は 00001φφ とし、4 桁ごとにアルファベットに変換すると「0A」となる。0000 は「0」となるが、バイナリ値として比較すると 000001<0000 である。一方、文字列として比較すると「0」<「0A」となり結果が違ってしまう。2 つのバイナリ値を比較する場合は、桁数の長い方の長さになるまで φ を続ける必要がある。ここでは 0000 ではなく、0000φφφφ として、「0F」という文字列を求めて比較をしなければならない。なお、末尾に φ を続けても、順序関係には影響しない。なぜなら、φ が通常は顕在化していないだけと考えることができるからだ。結果的に複数のレコードをソートする場合は少なくとも、バイナリ値の最大桁となるまで φ で埋めた表現で文字列にしたものを並べ替えの対象にする必要がある。

実用上は、毎回最大桁を求める必要があるとも言えるが、最大桁以上であればいいので、あらかじめ想定される桁数より多めの桁数で文字列に変換しておくという手段も利用できる。また、比較する桁が増えたとしても、それに満たない場合には末尾に「F」を付けるだけで良い。

5. 本手法の改良と評価

5.1. バイナリ値の生成に関わる問題点

ここまでに、順序を示すバイナリ値の生成が可能であり、2 つのバイナリ値の中間が常に存在することが明らかになった。新たなレコードを追加するとき、挿入する場所に限らずバイ

ナリ値を生成でき、他のレコードのバイナリ値を変更する必要はない。また、バイナリ値を文字列化することで並べ替えの手がかりになるので、データベースの文字型フィールドに対して適用可能である。ここで、問題点は次の通りである。

(A) 中間値に対して複数の候補がある場合にどれを採用すべきか
(B) 同一のレコードや、先頭、末尾に連続してレコードを追加する場合にビット数が増大する

このうち(A)の問題、すなわち桁数を増大させなくても中間値の候補が存在する場合のノードの選択方法としては次のものを検討する。他の条件設定も可能だが、5.2 項以降で説明する手法で利用するものとして、以下のものを利用する。

①. 最小の桁のものを求める
②. 長い方の節点に近い桁数が最大のもの
③. 長い方の節点に順序として近接したもの

最初や最後のバイナリ値の前や後のバイナリ値を求める場合は、これらのルールの「長い方」を「存在するバイナリ値」と読み替えればよい。

①については、2 つのバイナリ値について、前から走査をして異なる部分が発生した位置で次のルールによって中間値は求めることができる。前後のバイナリ値は表 2 の 3 つの組み合わせ以外にはならない。表にないものは、前後関係が逆になり定義と異なる。

表 2 最小桁を求める手段

前のバイナリ値の桁	後のバイナリ値の桁	中間値
0	1	それぞれのバイナリの共通部分
0	ϕ	前のバイナリ値に 1 を続ける
ϕ	1	後のバイナリ値に 0 を続ける

②については、前のバイナリ値と後のバイナリ値との異なる部分木について間順走査を行えば良い。そのため、桁数が増えると走査に時間がかかる傾向がある。見つからない場合は長い方のバイナリ値に 0 あるいは 1 を追加して延長したものを中間値にする

③の方法については、長い方のバイナリ値について、末尾から桁を 1 つずつ取り除き、順序の要件を満たすものを探す。近接した節点は、直接の祖先のいずれかになる(コードの順序の定義より、ある節点 A より大きい節点以下の部分木の節点は必ず節点 A より大きいことを逆に考えれば成り立つ)。手順としては、2 つのバイナリの共通しない部分で、桁を取り除くようにする。これで見つからない場合は長い方のバイナリ値に 0 あるいは 1 を追加して延長したものを中間値にするといった手法で高速に中間値が求められる。

これらの違いによるバイナリ値の生成の違いに付いても検討は必要だが、ランダムな位置へのレコード挿入実験では大きな違いは見られなかった。これらの手法は(B)の問題を解決するための手法と組み合わせて検討する事にする。

③については、厳密には桁を減らしたもので該当するバイナリ値が見つからない場合には間順走査をして求めるべきではあるが、同一レコードに対する追加の場合は、一方のバイナリ値がもう一方のバイナリ値の一部分になるケースが多数であることから、後で示す実験では厳密な手法は取らないで、処理速度を優先させた手順を採用した。

5.2. 桁数の増大を防ぐ方法

一連のレコードの途中あるいは先頭や末尾も、いずれも隣接する節点に対応するバイナリ値の中間値を求める場合、2 分木上での空間がないため、桁数を増やすことで中間値を求めるのがここまでの結論であった。図 5(i)のように、節点 A と B のバイナリ値の中間値を求める場合、C1 となる。そして、A と C1 の中間は C2 となり、「A の次のレコード」が次々に増えるとレコード数に比例してビット数が増大する。しかしながら、A と B の中間値として取り得る値は、桁数をさらに増大することによって、図 5 の(ii)のように数多くの候補がある。

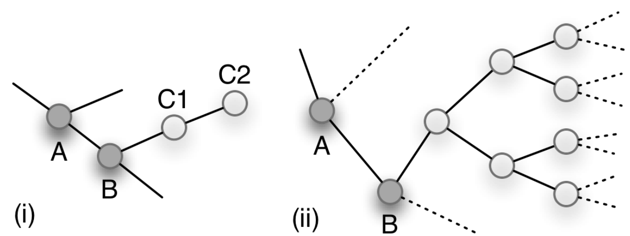

図 5 隣接する節点の中間の節点

ここで、同一レコードの次や前という追加が数多くある場合でのビット数の増大を避ける方法として、図 6 のような手法が考えられる。つまり、隣接する節点にある 2 つのバイナリ値 $A = (a_1,\ldots,a_s)$と $B= (b_1,\ldots,b_t)$の中間 C を求める場合、中間値のバイナリ値 C は、A と B が仮の最小値あるいは仮の最大値ではない場合、

s>t の場合 : C=A 1 0 0 … 0
t>s の場合 : C=B 0 1 1 … 1

として求める事ができる。ここで、末尾に続く 0 ないしは 1 の数 e は任意に指定ができる。

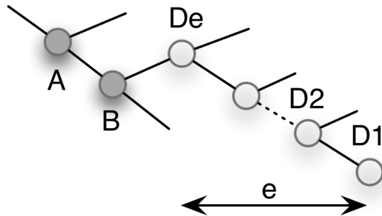

図 6 空間を作って節点を追加

先頭あるいは末尾に追加する場合、先頭のレコードのバイナリ値 S と、末尾のレコードのバイナリ値 E であるとする。S の要素がすべて 0 ではない場合や、E の要素がすべて 1 でない場合は、セクション 2 の方法で中間値を求めるとする。

S がすべて 0、あるいは E がすべて 1 で、S より前あるいは E より後にレコードを追加する場合のバイナリ値は次の通りである。

S より前：S 0 1 1 … 1
E より後：E 1 0 0 … 0

このルールに加えて、2 つのバイナリ値の中間に節点が存在する場合、長い方の節点に近い桁数が最大の節点を採用する（前述の②）というルール、あるいは長い方の節点の直近にある節点を採用するというルール（前述の③）を加える。先頭や末尾に加える場合は、先頭ないしは末尾のレコードのバイナリ値に一番近い節点を選ぶものとする。

表 3 は、e=4 として、常に末尾にレコードを追加した場合のバイナリ値を求めたものである。

表 3 改良手法によるコード付けの例

VLEI コード	改良手法（e=4）	
	②の手法	③の手法
φ	φ	φ
1	10000	10000
11	10001	1000
111	10010	100
1111	10011	10
11111	10100	1
111111	10101	110000
1111111	10110	11000
11111111	10111	1100
111111111	11000	110
1111111111	11001	11
11111111111	11010	1110000
111111111111	11011	111000
1111111111111	11100	11100
11111111111111	11101	1110
111111111111111	11110	111
1111111111111111	1111	1110000
11111111111111111	111110000	111000

改良手法では、最初のレコードは φ、次のレコードは、仮の最大値である 1 と比較する結果として、新たなレコードのバイナリ値は 10000 となる。②の手法では以後 11111 との中間で既存の節点に近く最長のものを採用して、10001、1010 と 4 桁のバイナリ値が続く。③の手法では以降は直近の節点なので 1000、100、10、1 と 0 が 1 つずつ減る。1 と仮の最大値との中間値はルールにより 110000 となる。

③の手法では e+1 レコードごとに 1 ビット増えることになる。n レコードを連続して挿入する場合、これまでの方法では、最大 n 桁のバイナリ値となっていた。これに対して n/(e+1)+e 桁になる。1 の桁数は e+1 レコードごとに 1 桁増え、最大でそれよりも 0 の数つまり e 桁だけ多いビット数となる。つまり、e=4 の場合は 15 ビットだったものが 15÷(4+1)+4=7 ビットとなり、ビット数は半分になった。しかしながら、たとえば VLEI コードで 1〜5 桁の場合でも、最大桁数は 5 桁となっているなど、連続する数値が少ない場合はかえってビット数を増やしてしまうことにもなる。

5.3. 想定される問題点と一般化

実際にこの手法でレコードを追加した場合は、セクション 2 で説明した方法よりも、同一のレコードの前後に連続してレコードを追加する場合には桁数の増加を抑えることができる。一方、図 6 では A と B の中間で D1 を採用したときに、A と D1 には節点が存在するが、D1 と B の間には桁数を増やさずに中間の節点を追加することはできない。この点は不利になるとも考えられる。

そこで、A と B のいずれに対しても空間を確保した中間値として、図 7 のような位置の節点を採用する。e 個の節点を増やす事で、A と D1 の間に節点を確保すると同時に、B と D1 の間にも f 個の節点を確保する。「B の直前のレコード」が次々と発生しても、中間にある節点を順番に使用する範囲では、最大桁数の増大はなくなる。一方、中間値がなくなると、e+f 個の桁数が増えることにもなるため、連続して追加されるレコード数が少ない場合にはさらに不利になる。

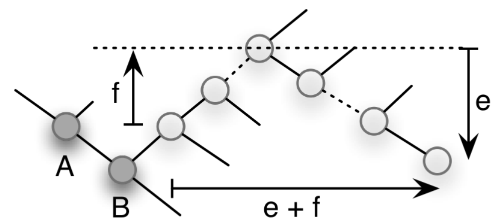

図 7 前後に空間を作って節点を追加

5.4. 実験結果

最適な e や f を検討するために、同一のレコードの直後に連続的にレコードが追加される状況での平均桁数と平均最大桁数を比較する実験を行った。実験では、30 レコードのランダムな位置への挿入の後に、特定のレコードに対して 20 レコードを直後に挿入する。この特定のレコードは、その

ときに存在するレコードからランダムに選択した。この作業を10回繰り返して500レコードを作成する実験を行った。1つの条件に対して20回の実験を行い、平均桁数と平均最大桁数を求めた(図9)。

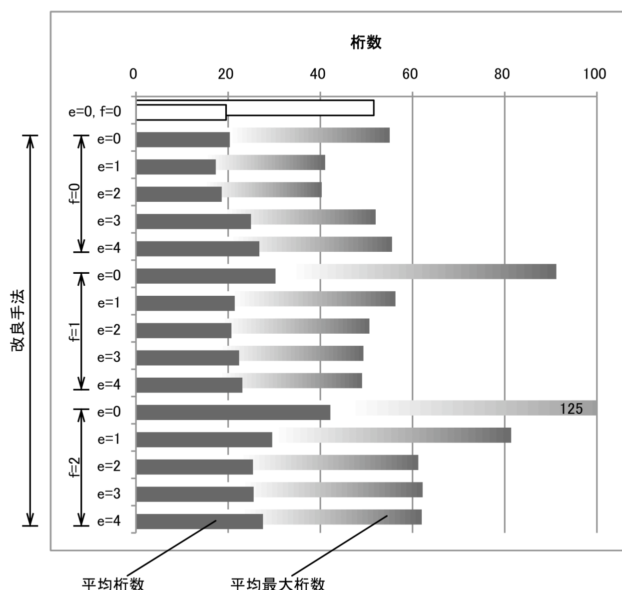

図 9 連続追加を伴うレコード追加実験の結果

図9のグラフの最初のe=0, f=0の実験値は、5.1項の①の手法で求め、他の場合は③の手法で求めた。

最初のe=0, f=0の手法に比べて、f=0でe=1あるいはe=2の場合に、平均桁数が少なく、最大平均桁数も少ない。e=1, f=0の場合は平均桁数も平均最大桁数も有意水準1%での有意差があった。e=2, f=0は最大平均桁数のみ、1%での有意差があった。このことから、eの値を1以上にするという改良手法は有効であると言える。一方で、fの値が1の場合には目立った改善がなく、fの値を1以上にすることに対する効果はこの実験結果には見られない。

この実験では、全体の40%のレコードは、連続した順序の位置に挿入されるものであり、eの値を増やす効果のみが結果として現れたものと推測できる。eやfを増やすことで連続して挿入する場合に対処はされるが、逆にランダムな場合には1桁しか増えないところがe+f桁増えてしまい、かえって桁数を増大する効果になる。結果的にはレコードの生成過程に応じたパラメータや動作の調整が必要になることが予測される。

今回の実験では、②の手法では桁数の増加による処理時間が大きくなり過ぎるため、実験には至らなかったが、課題として処理を効率化して実験可能にする必要がある。

実験に使用した中間のバイナリ値を求めるプログラムは、以下のアドレスで公開している。

http://msyk.net/study

6. まとめ

リレーショナルデータベースで順序を記録するためのラベルとしてVLEIコードを利用することを想定し、バイナリ値の桁数がデータ数に比例して増加する場合の対処方法を検討した。VLEIコードによって生成されるバイナリ値が2分木で記述できる事を手がかりにして、2つのバイナリ値の中間値を求める場合に空間を確保して追加する方法を用いる事で、最大桁数抑える傾向があることが分かった。

7. 謝辞

本稿に関して有益なヒントをいただいた筑波大学システム情報工学研究科コンピュータサイエンス専攻長の北川博之教授にお礼を申し上げます。また、慶応義塾大学図書館にもお礼を申し上げます。

8. 参考文献

[1] D. W. Matula and P. Kornerup., An Order Preserving Finite Binary Encoding of the Rationals, Proc, 6th IEEE Symposium on Computer Arithmetic, pp201-209, 1983.

[2] P. Kornerup, D. W. Matula, LCF: A Lexicographic Binary Representation of the Rationals, Journal of Universal Computer Science, vol. 1, no. 7, pp484-503, 1995.

[3] Kazuhito Kobayashi, Wenxin Liang, Dai Kobayashi, Akitsugu Watanabe, and Haruo Yokota. VLEI code: An Efficient Labeling Method for Handling XML Documents in an RDB. In ICDE, 2005.

[4] 高橋昭裕, 梁文新, 横田治夫, "要素挿入に強いXMLラベルの構造情報抽出手法の提案と評価", DEWS2008 C8-4, Mar. 2008.

[5] 新居雅行, "コメントチェーンのデータベース化", http://msyk.net/fmp/relationalbook/ch6/ch6-4.html

[6] B.A. Davey, H.A.Priestley, Introduction to Lattices and Order Second Edition, pp11-13, Cambridge University Press, 2002.