DB 31 − 1
(1976. 11. 11)

SPIRESの2つの特徴的な機能
— action BNF と変形 B-tree —

国井 利泰 (東京大学)
穂鷹 良介 (SSL)

SPIRES (Stanford Public Information REtrieval System) は，スタンフォード大学によって開発された汎用のオンライン データベース管理システムで1972年から稼動している。 ここではSPIRESが備えている多くの機能のうち，特徴的な2つの機能，action BNF と B-tree を若干変形した木構造の蓄積構造とについて紹介する。

1 Action BNF

BNFはプログラミング言語のシンタックスを記述する方法としてよく知られているが，セマンティクスを記述するものではない。 action BNFはこのBNFの欠点を補って主に端末とコンピュータとの間の会話形コマンドのシンタックスとセマンティクスについても記述するものである。

図1にaction BNFの例を示す。

```
|COMMAND LANGUAGE|  (0,MASTER LANGUAGE)<LOGOFF>
<MASTER LANGUAGE>  <BUILD-COMMAND> (BUILD_LANGUAGE)
                   |LOGOFF|
                   <EXTRA COMMANDS>
<LOGOFF>  <1> LOG(OFF) (SP) <4>
<OFF>  OFF
<BUILD-COMMAND>  <1> BUILD (SP) <4>
<BUILD_LANGUAGE>  <400> (2,BUILD_LANGUAGE2)
                  <401>
<BUILD_LANGUAGE2>  <402>
<EXTRA COMMANDS>  <1> SET <SP> UPLOW (SP)<4><8>
                  <5>
(SP)  0,1,1,40
```

図1 action BNFの例

上記では |COMMAND LANGUAGE| ::= (0,MASTER LANGUAGE)<LOGOFF> とでも書くべき所を ::= を省略した形になっている。 文法の生成ルールは ::= を補って考えると

左辺 ::= 右辺

という一般規則に従って書かれる。 右辺がいくつかの選択項 (alternate term) を持つときには行を分けて

左辺 ::= 右辺-1
　　右辺-2
　　右辺-3

のように書かれている。

action BNF は項を言語の要素とするプログラミング言語と考えることができる。右辺は閉じたサブルーチン群の呼出しで左辺はサブルーチンの名前と考えられる。左辺と右辺の項はいくつかの種類に分類されるがこれらはサブルーチンの呼び出される方式，サブルーチン実行の結果の成功または不成功がその後のサブルーチンの呼出しにどのように影響を与えるかといった事柄と関係する。

左辺の種類

左辺（文法上 production と呼ばれている）には次の4種類のものがある。

comment. action BNF の文の間に挿入されるコメント。＊が特定位置にある場合コメントとなる。

standard production. ＜name＞ のような形をした項。name はいずれかの右辺の中に現れなくてはならない。parsing の時には ＜name＞ の右辺の第1項がまず用いられ，それが不成功に終ったら次の alternate term が用いられる。このような操作を続けてすべての alternate term が不成功のときはこの左辺を生成に用いることが不成功になったものとされる。

terminal production. （name） のような形をした項。256 個の8ビット文字列の部分集合を定義する。

basic production. |name| のような形をした項。pasing時には standard production と同じように扱われる。ただし，右辺の項としてまったく現れなくてもよい。

右辺の項の種類

右辺の項（文法上 production call と呼ばれている）には次の8種類がある。

required link. ＜name＞ のような形をした項。この項が存在すると左辺が ＜name＞ であるような production を呼び出せ という意味になる。呼ばれた production が parsing に失敗すると右辺の呼び出し側の production call は失敗したことになり，次の alternate part が試みられることになる。もしもこの required link が右辺の最後の alternate part であるならば右辺を呼び出した production が不成功であったことになる。

production ＜name＞ が parsing に成功すると parser はこの required link の次の項を試みることになる。もしもそのような項が存在しなければこの右辺を呼び出した production は成功であったことになる。

lookahead link. |name| のような形をした項。図1で説明するならば右辺に現れた |LOGOFF| が lookahead link の一例である。production call |LOGOFF| に呼応して ＜LOGOFF＞ production が呼出される。このproductionが不成功のときには次の production call 即ち ＜EXTRA COMMANDS＞ の parsing に進む。逆に成功のときには |LOGOFF| を含む production 即ち＜MASTER LANGUAGE＞ が不成功とされる。なお lookahead link の次には必ず1個以上の alternate term がなくてはならない。

character string. 図1の例でいえば LOG あるいは BUILD のような単なる文字列の項。右辺に現れたのと同じ文字列が parsing 時に現れると production call は成功であり，さもなくば不成功である。

semantic link. ＜n＞（ここでnは単なる自然数）のような形をした項。

図1の例でいうならば<1>, <4>などがその例である。それぞれセマンティクスの処理プロセス1あるいは4を呼出すことを示す。action BNFでセマンティクスを記述するものはsemantic linkによって呼出されるプログラムなのである。

optional link, standard. (1,name)あるいはこれを略して書いた(name)のような形をした項。parserがこの項を認識すると<name>あるいは(name)あるいは|name|のproductionを呼出す。parserはそのproduction callの成功不成功に拘らず次のproduction callに進む。もしも次のproduction callが存在しなければこのoptional linkを含むproductionは成功となる。

optional link, call until failure (CUF) (0,name)のような形をした項。(name),<name>あるいは|name|で示されるproductionを不成功になるまで繰返し呼出す。不成功になったときこのoptional linkを含むalternate part中の次の項のparsingに進む。次の項がその時点で無ければ右辺を呼出したproductionは成功である。図1の(0,MASTER LANGUAGE)ではコマンド列を不定個受付けることをこのoptional linkを用いて表現している。

continuation link. (2,name)のような形をした項。単に右辺の項を延長するために用いる。nameによって呼出されたproductionの右辺がこのcontinuation linkの右に続けて書かれたかの如くparserは扱う。

class scan terms. max,min,like,hexstring,charstringのような形をした項。terminal productionの右辺にのみ現れる。たとえば0,1,0,40は最大長が40で最小長1, hexa decimal表現で40というものではない文字列を表現する。likeが0は以下の文字列のいずれでもよいことを意味し, 1のときには以下の文字列のいずれかであればよいということを示している。

SPIRESのシステム生成

SPIRESはこのaction BNFを用いてシステムのコマンド処理機能の定義を行なっている。システム生成は次の図2に示す手順で行なわれる。

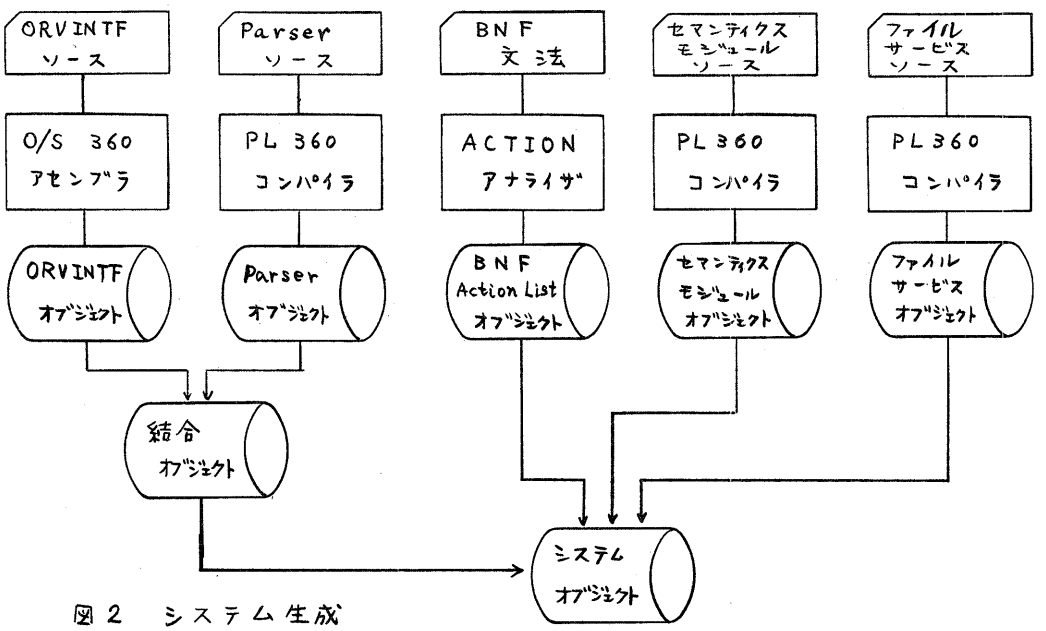


図2 システム生成

2 変形B－Tree

2.1 SPIRESの物理ファイル構造

SPIRESでユーザが用いる物理ファイルには tree-structured, slot-structured, non-record-type の3つの data set があり，これらのファイルのレコードはそれぞれ node, slot, entry と呼ばれている。

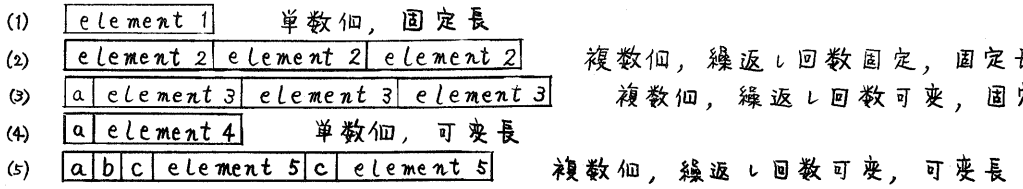
tree-structured data set はB-tree と似通った木構造のファイル構造でこれについては以下にやや詳細に紹介する。

slot-structured data set はレコードに1対1に対応する整数値をキーとしたファイルで，原則として固定長である。しかし後に述べる residual data set に可変長部分を蓄積してそれを固定長部分から point しておき，論理レコードとしては固定長，可変長両部分を合併したものを考えることはできる。この data set の利点はキーの値を与えることにより1アクセスで固定長部分を取出すことができる所にある。

non-record-type の data set の中にはシステム管理のための諸ファイルのほか residual data set が存在する。物理ファイル構造としてはこの residual data set が興味深いので2.3で再述する。

Element representation

1レコードの中のフィールドに対応するものを element という。element の表現形式として次の5種類がある。



a：全長（2バイト） b：繰返し回数（2バイト） c：element長（2バイト）

図3 Element Representation

Record representation

1レコードの中には図4に示すように各 element が分類されて蓄積される。

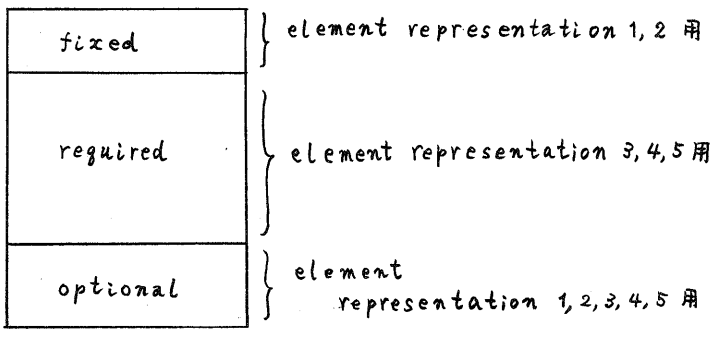


図4 Record representation

レコードを識別するキー element はそれが固定長の場合には fixed 部分の最初に，可変長の場合には required 部分の最初に置かれる。

optional部分には各レコードごとに必ず存在するとは限らない element が置かれる。どの element が optional 部分に置かれたかどうかの表示

は，required部分の（もし存在すれば）キーelementの次にbit maskの形で置かれる。

2.2 Tree-structured Data Set

tree-structured data setでは各node（レコード）を一意に識別するキーelementがあり，この値を指定することによっていわゆるディレクトリ（インデックス）によって効率よくnodeを検索することができる。　この意味でtree-structured data setは従来のISAMファイルと異るものではない。　しかしtreeのレベルをすくなくする手段，可変長レコードの扱いなどに工夫がされ，しかもロジックが公開されているので参考になる。

File block format

すべてのレコードは2048バイトの固定長file blockに適当に分散して蓄積される。　図5にfile blockの概要を示す。

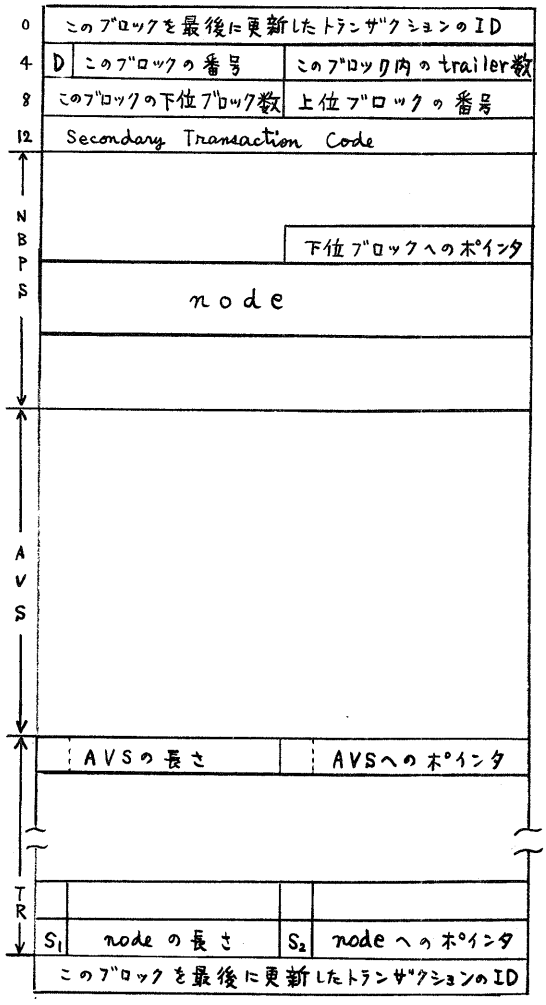


図5　file block format

[説明]　D：ブロックが破壊されているかどうかの表示

Secondary Transaction Code：エラー回復の際に用いる情報

NBPS：Nodes and branch pointer space

AVS：Available Space，ブロックの上からnode及び下位のブロックへのポインタが詰められ，下からtrailerが詰められるが，その間にはさまれている空き領域

TR：Trailers，nodeはキーelementの値と無関係にブロック内に配置されるが，それがブロック内のどこにありどれだけの大きさかなどを示す。trailerはキーelementのアルファベット順にブロックの下から配置される。　trailerのうちで最後のもの（nodeに一番近いもの）は特別で，AVSの長さとその先頭アドレスとを示す。

S_1：nodeが破壊されているかどうかの情報，論理的に削除されたかどうかの情報

S_2：レコードの他のブロックへのオーバーフロー情報

Treeの構成

図6にtree-structured data setの例を示す。 #0, #9, #20, #15などはブロックを示す。 各ブロックには下位のブロックへのポインタとnodeの対が格納されている。 たとえば#0のブロックで1,9,15,20は下位のブロックへのポインタである。 ABC, GHI, PQRはnodeである。

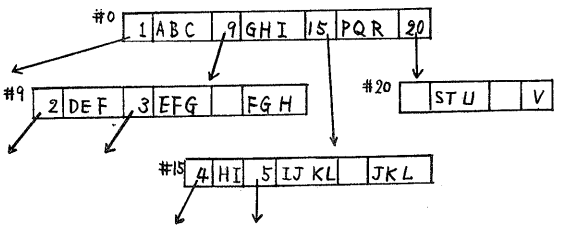


図6 treeの構成

下位のブロックへのポインタの後のnodeをnon terminalといい, ポインタが先行していないnodeをterminalという。 SPIRESのtreeではブロック内のterminal nodeの前にnon terminal nodeが配置されることはない。 この性質はtreeの更新時に利用され, 保たれる。 nodeは可変長なのでブロック内のnodeの個数もまた可変である。 高速にアクセスするためには後に述べるresidual data setを利用するなどしてキーelementを含む最小のデータをtree-structured data setに入れるようにしてブロック内のnodeの数を多くすることが考えられる。

Treeの更新

nodeを追加, 削除, 変更することによってtreeの更新が行なわれる。 nodeの削除は図5に説明したtrailerのS1情報のセットによってなされる。 変更は基本的には削除と追加とで可能であるのでtreeの更新として問題となるのは追加のみである。 以下これについてSPIRESのロジックを説明する。

なお以下の図で□はブロック内の空きスペース, ▤はnon terminal nodeの列, ▦はterminal nodeの列, ▨は追加するnode, ⇩は追加するnodeの挿入位置, 矢印は下位ブロックへのポインタを示す。

(1) 追加nodeを挿入すべきブロック(old block)に空きスペースがある場合, そこにnodeを挿入する。

(2) old blockの隣接ブロックに空きスペースがある場合, 図7のようにnodeを入れる。

(3) old blockの全terminal nodeはgrowth space(将来の拡張用のスペース)込みで新しいブロックに入る場合, 図8のように更新する。

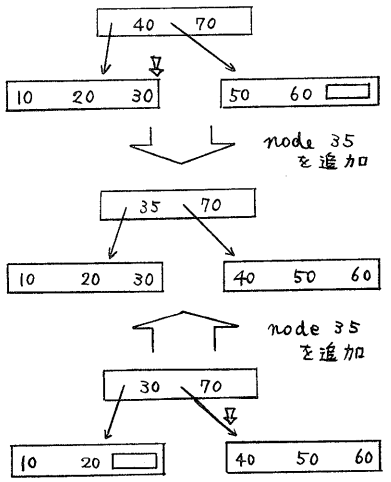


図7 ケース(2)

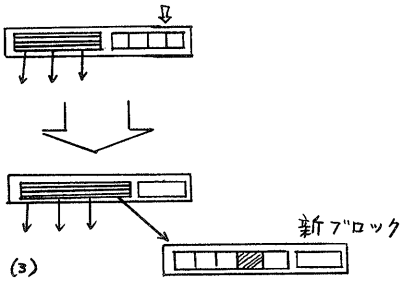


図8 ケース(3)

以上のいずれのケースでもない場合，更に以下の3つのケースがある。

(4) old block の上位ブロックに空きスペースがある場合，図9のような更新が行なわれる。

(5) old block の上位ブロックに空きスペースがない。しかし空きスペースを作り出すことができる場合。この場合にはまず図10のように上位ブロックの terminal node を利用して上位ブロックに空スペースを作り出す。その後の処理は従ってケース(4)の更新手続きが適用される。

(6) old block の上位ブロックに空きスペースがなく，しかも(5)のようにして空きスペースを作り出すこともできない場合。図11に示したような更新手続きとなる。なお追加 node の挿入位置は必ず terminal node の前か後であって，non terminal node の前になることがないということを注意しておこう。

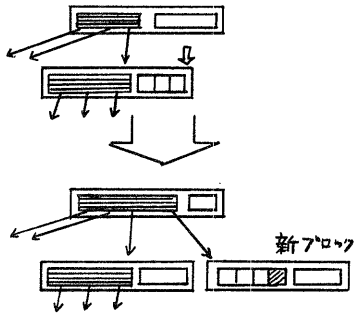


図9　ケース(4)

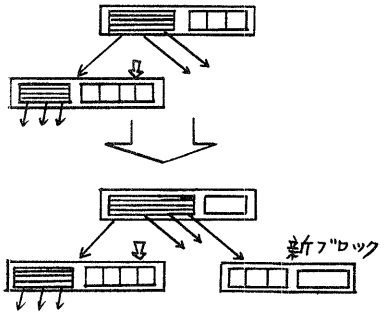


図10　ケース(5)の途中

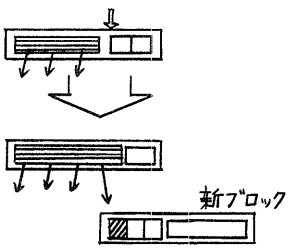


図11　ケース(6)

2.3 Residual Data Set

あるレコードを tree-structured data set に蓄積しようとしたとき，そのレコードの全 element が蓄積しようとして選ばれたブロックの中に入り切らない場合がある。また入れる気になれば入る場合でもあまり頻繁にアクセスしない element などは効率上キー element とも入るブロックに蓄積しない方が得策の場合もある。

このような場合レコードの一部分を元のブロックに置き，残りの部分を複数個に分けてチェインで縦人で別に用意した data set に蓄積する。このための data set を residual data set という。residual data set は tree-structured data set のオーバーフローデータだけでなく slot-structured data set からの分割データも蓄積できる。residual data set へのポインタは overflow segment pointer と呼ばれているがその内容は residual data set の何番目のブロックかということを示す overflow block # とそのブロック内のどのデータかを示す trailer # とから成っており，もとのブロックの node の最後の4バイトにおかれる。residual data set 内のブロックで再びデータがオーバーフローすることも考えられるがこのときの扱い方も同様である。

図12は residual data set 内の available space の管理方式を説明している。residual data set に対して n バイトのスペースが要求されると8の倍数で n を下回らない最小の長さのスペースを取るようにする。available space は8の倍数バイトの大きさごとに管理されておりそれぞれ大きさごとに分類されてそのサイズの available space を持つブロックが available space table から一方向

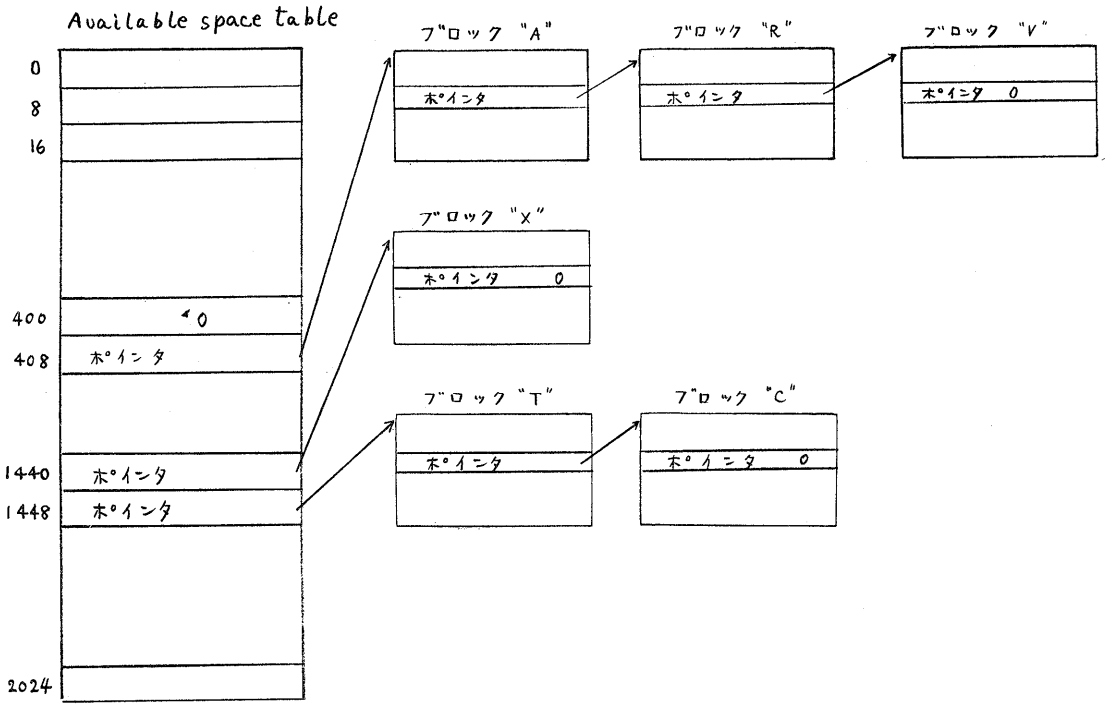


図12 Available space 管理メカニズム

のリストとして結ばれる。 スペースの要求が出されるとそのサイズに適合したavailable space tableのポインタからサイズが大きくなる方向に向って探索が行なわれる。 ポインタの値が0ということはそのサイズのavailable spaceを持つブロックが登録されていないことを意味する。 0以外の値を持つポインタが見つかるとブロックのチェインの最初がアクセスされる。 そのブロックが実際に管理表に登録されている通りのavailable spaceを持っていたならばそのブロックが割当てられ利用される。 後に述べる理由により管理表からチェインされているブロックの実際のavailable spaceが管理表から推測されるサイズと異っているケースが生じるが, このときには正しい管理表の位置につなぎかえ, 次のブロックのチェックに進む。

注意すべきことは, このようにして利用されるブロックが定まって実際に使用されるときでも依然としてそのavailable spaceのブロックのチェインの中に残したままにしておくことである。 これはチェインの途中のブロックをつなぎ変えるためには双方向のポインタがないと時間がかかり, しかも双方向のチェインの保守に手数がかかるからである。 またこのブロックの空きスペースを利用し終ったユーザが再びその空きスペースをだれも気付かぬうちに再び返却したときには, 上記のavailable spaceの管理表がそのまま再び使えるという利点もあるものと思われる。 これに反してavailable space tableから直接にポイントされている次のブロックだけは双方向のポインタがなくても自由に繋ぎかえることができる。

3 おわりに

ＳＰＩＲＥＳは大学で開発されたデータベースシステムではあるが長年にわたって実用に供されてきたシステムである。その意味でここに紹介した２つの技能も証明済みのものであるといって差支えない。たとえば2.2で紹介したnodeの追加方法を採用した結果，年間10万件のデータの成長があったようなデータベースに対してもファイルの再編成をした3年間存続したと報告されている。

本稿を作成するに当り，日頃活発な討議に参加させて頂いている日本ユニバック総合研究所，電子技術総合研究所の皆様に感謝の意を表したい。

参照文献

[1] J. R. Schroeder, W. C. Kiefer, R. L. Guertin, W. J. Berman, Stanford's Generalized Database System, Proceedings of the International Conference on Very Large Data Bases, pp. 120-143.

[2] J. R. Schroeder and SPIRES staff, Design of SPIRES II Vol I 2nd ed. SCIP, Stanford Univ. (July 1973)

[3] J. R. Schroeder and SPIRES staff, Design of SPIRES II Vol II 2nd ed. SCIP, Stanford Univ. (July 1973)