第 4 章

インデクス

インデクス (index, 索引) は，ひとことで言うと，ある table において key を指定して record を絞り込む操作を高速に実行するための補助的なデータ構造です．インデクスがなくても，table の全 records をチェックすれば，絞り込み操作は実現できます．これを table の full scan と呼びます．例えば，1M 個 (M は million，100 万の意味) の records が存在する table の中で，たった 1 行を見付け出すために 1M 個の records を全部読んで調べるというのはいかにも効率が悪いですね．

典型的なインデクスは tree map または hash table を使って構成されています．これらは N 個の records から成る table を検索するために必要なデータアクセスが，log N 回程度で済んだり，定数回程度で済んだりするようなデータ構造です．しっかりと考える場合は最悪のケースも検討しなければなりませんが，今はそれをあまり考えずに，これらのアクセス回数の目安を O(log N)，O(1) と書いて表現します．詳しく知りたい人は，「計算量」や「O 記法」などのキーワードで調べてみてください．アルゴリズムとデータ構造の教科書には必ず載っているはずです．

4.1 Tree map を用いたインデクス

Tree map を実現するには directed rooted tree (根付き有向木) 構造を使います．Directed rooted tree は directed graph (有向グラフ) に制約を加えたものです．Directed graph を表現するためには vertex (頂点) の集合 V と edge (辺) の集合 $E \subset V \times V$ を用います．$e = (v_1, v_2) \in E$ について，v_1 から見て e を outcoming edge，v_2 から見て e を incoming edge と呼びます．また，e から見て v_1 を source， v_2 を destination と呼びます．Directed rooted tree とは，(1) 連結 (connected) しており，(2) incoming edge が存在しない root (根) と呼ばれる vertex がひとつだけ存在し，(3) root 以外の vertex は全て incoming edge が唯ひとつのみ存在する，これらの条件を全て満たす directed graph です．Outcoming edge の存在しない vertex を leaves (葉) と呼びます．各 vertex にとって，自身の outcoming edge で直接繋がっている vertex を children と呼び，incoming edge で直接繋がっている vertex を parent と呼びます．Directed rooted tree を考えるときは vertex のことを node と呼ぶことが多いのでここでもそれに倣います．

実際にメモリ上で tree map を作るとき，典型的には node を構造体として作ります．Children や parent を参照するのにはポインタを使います．(Parent へのポインタはなくても良いですが，効率的なアクセスのため用意することが多いです．その代わり，冗長な情報を持つことになるので操作時に気を使う必要が出ます．)

Tree map は key を表す型が全順序を持つことを要求します．Tree map を directed rooted tree で実現するとき，各 node は自分の担当する key の連続部分空間を children の数だけ連続部分空間に分割して保持します．ここでの連続部分空間とは，ひとつの範囲で表現できる部分空間，という意味で使っています．つまり，ある key 値を持つ record が存在するならば，root node から key 値が含まれる連続部分空間を持つ child を選ぶという操作を繰り返すことで到達した leaf node が保持していることが保証されます．つまり，木の深さ分だけ child を辿れば良いわけです．木の深さは木がバランスしている (一番深い leaf node と浅い leaf node の深さの差が高々定数倍と見做せる) とき，node 数 N に対して深さが O(log N) と見做せ，O(log N) のステップ数で record に辿りつけます．

二分木 (fanout すなわち children の数，が高々 2 の木構造) による key の型が整数である tree map の例を示します:

```
Root:
  border_key: 5
  children: N11, N12

N11:
  border_key: 3
  children: N21, N22

N12:
  border_key: 8
  children: N23, N24

N21:
  border_key: 1
  records: (1, 'aaa')

N22:
  records: (3, 'ccc'), (4, 'ddd')

N23:
  records: (6, 'fff')

N24:
  records: (8, 'hhh'), (10, 'jjj')
```

Root node には border_key として 5 が格納されています．これは，left child を辿っていった先には key < 5 の record しか存在しないことを意味します．同様に，right child を辿っていった先は $5 \leq$ key の record しか存在しないことを意味します．N12 は key < 5 の連続領域を担当しています．そして，さらに key < 3 と $3 \leq$ key < 5 の 2 つの領域に分割し，それぞれ children である N21, 22 に割り当てます．N22 は $5 \leq$ key の領域を担当し，$5 \leq$ key < 8 および $8 \leq$ key の 2 領域をそれぞれ children である N23 および N24 に割り当てます．N21, N22, N23, N24 は leaf node なので，その中に children へのポインタではなく，対応する key 値を持つ record (もしくは record へのポインタ等) が格納されています．この例では $N = 7$ ， $\log_2 N \approx 2.8$ で，深さは 3 となります．

Tree map が有効に機能するためには木がバランスしていることが必要となります．これを実現する操作方法を持つ木構造をバランス木と呼びます．二分木だと，赤黒木 (red-black tree) や AVL-tree が例として挙げられます．また，大前提として fanout は 2 以上であることが必要となります．Fanout が 1 だと，root を先頭とする連結リスト構造を意味しますので，我々が欲しい性質を持たないからです．

扱うデータの量が使えるメインメモリよりも大きいことが想定される DBMS は，メインメモリとディスク[*1]上の細かいデータのやりとりを頻繁に行う必要があるため，ページと呼ばれる (4KiB や 16KiB などの) 固定サイズ連続領域を node として扱う B+tree もしくはその亜種が良く使われています．一般に，non-leaf node は fanout - 1 個の key を格納する必要があるので，B+tree の fanout は key のサイズに依存します．

昨今研究開発が盛んなインメモリデータベースは，データが全てメインメモリに納まる前提を置きますが，必ずしもディスク上の構造とメモリ上の構造が一致する必要はないため，メモリ上の tree map の表現は，必ずしもページではなく，ポインタなどの間接参照データが使われていても問題ありません．

C++ を使っていてとりあえず tree map が欲しいときは，`std::map<Key, Value>` 型のオブジェクトを作ればそれでおしまいです．もちろん thread-safe ではないですが...

4.2 Hash table を用いたインデクス

N を自然数とし，key 空間から $\{0, 1, ..., N-1\}$ への写像 h を考えます．例えば，key 値を入力として適当な非負整数を出力する副作用のない関数を用意し，出力された非負整数を N で割って余りをとれば h は実現できます．

予め N 個のバケットと呼ばれる要素からなる配列を用意しておき，key 値から h で得た値を配列のインデクスとしてひとつのバケットにアクセスします．バケットには，複数の record が格納できるようになっており，バケット内の key が一致する record を探すことで絞り込みます．一例としてバケット内の records は連結リストを用いて管理します．同一の key 値を持つ record は他のバケットには存在しないことが保証されており，このバケット内を探すだけで済みます．

以上が典型的な hash table の設計です．たくさんの亜種があります．写像 h は hash 関数と呼ばれており，このデータ構造が hash table と呼ばれている理由となっていると思います．Hash 関数は key サイズに依存する計算量を必要とすることがほとんどですが，Key サイズは record 数を M としたとき， M に対して定数と見做せることがほとんどです．つまり key 値からバケットを特定する計算量は $O(1)$ となります．

Hash table が有効に機能するためには，各バケットに格納されている record 数が高々定数と見做せることが求められます．このとき，key 値から record に到達するための計算量が $O(1)$ と見做せるからです．そのため， h は典型的に使われる key 空間の部分集合に対して一部のバケットに偏らないような値を出力することが求められます．また，ここで説明した hash table はバケット数 N が固定であるため，N に対して record 数 M が小さすぎれば空のバケットが多くなって空間が無駄になるし，逆に大きすぎれば，バケット内の record 数が増えすぎて定数と見做せなくなってしま

[*1] HDD や SSD などの不揮発性を持つ二次記憶装置のことを通称としてディスクと呼びます．Hard Disk Drive の Disk です．

います．このような状況に対応するため，N 個の要素を持つ配列から (通常は $N < N'$ であるような) N' 個の要素を持つバケット配列にデータを移動することを rehash と呼びます．このとき，値域が異なるので，当然 hash 関数も異なります． N で割る代わりに N' で割るなどする必要があるからです．Rehash 一般に $O(M)$ かかりますので，rehash が頻繁に発生する状況は hash table が有効に機能しているとは言い難いです．

如何にごく単純な hash table の例を示します．Key 型は uint とし， h は簡単のためただの剰余としました:

```
h(key: uint) -> {0,1,2,3,4}:
  return key % 5

Array as a hash table with size 5:
  [B0,B1,B2,B3,B4]


B0: (10, 'jjj')
B1: (1, 'aaa'), (6, 'fff')
B2: (7, 'ggg'), (12, 'lll')
B3:
B4: (9, 'iii')
```

$N = 5$ の hash table です． h はただの剰余なので，5 で割った余りがそのままバケットインデクスを示しています．

C++ を使っていてとりあえず hash table が欲しいときは，`std::unordered_map<Key, Value>` 型のオブジェクトを作ればそれでおしまいです．やはり thread-safe ではありませんが...

4.3 インデクスのトレードオフ

ひとつの table について，一般に key は複数存在します．典型的にはひとつの key についてインデクスを高々ひとつ作って使います．性質の異なるインデクスを同じ key に対して作ることは考えられなくはないですが，それはかなり特殊な状況だと思います．

インデクスはその key を用いた絞り込み操作を高速化しますが，その代償として，record をひとつ write する度に，変更に関係するインデクスを全て更新する必要が生まれ，record の write コストが増えるというデメリットが発生します．どの key についてのどのようなデータ構造を用いたインデクスが必要か，または不要かについての判断は，データベースの key 分布とワークロード (どのようなアクセスがどの程度発生するか) に依存します．つまり，最適なインデクス構成を決めるのは簡単ではないということです．

また，インデクスの使用についても難しい問題があります．複数の条件で record を絞り込む場合，必ずしも全ての条件に一致するインデクスが必要というわけではありません．例えば，`age = 30 and weight = 60` という条件であれば，`(age, weight)` という key に対応するインデクスを使う方法もありますが，`(age)` という key に対応するインデクスで絞り込んだ records をさらに `(weight)` という key に対応するインデクスで絞り込む方法もありますし，`(age)` のインデクスで

絞り込んだ records を全部チェックするという方法もあります．性能の指標として重要なのは，最初もしくは最初の方でどれだけ絞り込めるかという点です．そもそも table に格納されている records 数が少ない場合は table full scan の方が高速なこともあります．

SQL をサポートする DBMS にはクエリ最適化 (query optimization)，もしくは実行計画 (execution plan) 作成と呼ぶ処理を行う機能があります．その処理の中で最も重要な仕事が最適な絞り込み方法を推定することです．Query optimization は SQL のような宣言的言語で書かれた命令を実行するときに必要になりますが，組み込み DBMS では execution plan を直接的にアプリケーションコードが書き下すことが多いです．つまり，そのような DBMS ではどのインデクスをどの順番で使って records を絞り込むかについても明示的に指定します．

どのインデクスを作るかについての判断は，どのインデクスをどの順番で使うかの判断よりも頻度が低いです．前者はデータベース物理設計，後者はクエリ最適化における判断なので．最近の商用プロダクトの一部では，ワークロードを監視して自動的にインデクスを作成したり削除したりする機構も実用化されているようなので，インデクス構成についての判断の頻度は上がっていると言えるでしょう．また，特定の key 範囲のみインデクスを作成することも有り得ます．これらの判断はすべてインデクス作成と使用のトレードオフがあるから必要となるのです．

4.4 インデクス構造のアクセス排他

シングルスレッドのプログラムであれば，インデクス構造を触っているのは自分ひとりなので，アクセス排他について何も考える必要がありません．もし，マルチスレッド/マルチプロセスのプログラムで，複数の worker が同一のインデクス構造を触るときは，アクセス排他の仕組みが必要となります．どう排他すべきかについては，また別途考える機会があるかも知れませんが，本稿ではこれ以上踏み込みません．

4.5 Table 本体の構造について

Table そのものはどのような構造をしているのかについて，2 パターン考えられます．

1 つ目は，table 本体もインデクスとして実装する場合です．この場合は primary key もしくは人工的な隠し key が定義されていて，それを使って record 本体の格納場所まで効率的に辿りつけるようになっています．この場合，table full scan はこのインデクスを強制的に使うことになりますし，その他のインデクスにおける value は primary key を保持することもあります．

2 つ目は，table 本体は独立したデータ構造として実装する場合です．この場合，メモリ上，ディスク上のレイアウトには自由度があります．Table full scan のみ実行できれば良いからです．各 record の位置は，なんらかのアドレスもしくはその組み合わせで表現することが多いと思います．

■コラム: ブロックデバイスとしての永続ストレージ

ブロックデバイスは固定サイズのデータであるブロックを要素とする巨大な配列として抽象化されたデータ保持のためのデバイスです．配列のインデクスをブロックアドレスといい，アクセスする先頭ブロックのアドレスとブロック単位のサイズ，すなわち，ひとつのアドレス範

囲を指定することで読み書きします．

ブロックデバイスという抽象が必要になったのは HDD の性能特性によるところが大きいと思います．昔の HDD はブロックサイズが 512bytes でしたが，今の多くの HDD は 4KiB です．HDD の中にはプラッタと呼ばれる磁性体が塗られた円盤が複数枚入っていて，プラッタの上を滑るように動くヘッドと呼ばれる細長い三角形のようなものが入っています．ヘッドの先がプラッタ上の任意の位置に移動して磁力の向きを検出したり変更したりできるようになっています．実際には，ヘッドはプラッタの中心から外周に向かう線上をステップモータの力で移動できるようになっていて，それに加えてプラッタが回転することによってプラッタ上の任意の極小領域にヘッドを位置合わせしてその位置に記録されているデータを読み書きすることができます．様々な技術によって微細化の努力は今でも続けられていますが，ヘッドの位置合わせ操作の高速化はとっくの昔に限界を迎えています．

プラッタもヘッダも物理的に動いて位置合わせするので，特定のブロックアドレスにアクセスするためにはミリ秒単位の時間がかかります．市販されている 3.5inch の HDD で高々 10ms 程度です．一度位置を合わせてしまえば，プラッタの回転に合わせてアクセスできるデータが変化するので，連続領域の読み書き，すなわちシーケンシャルアクセスは比較的高速です．最近の市販されている 3.5inch HDD だと 200MB/s 程度です．この連続領域をブロックの配列として捉えることで，アドレスが連続するブロックの読み書きは高々 1 回の位置合わせで実現できるように作られています．

もし HDD をバイト単位でアクセスできるようにしてしまうと，1byte アクセスする度に位置合わせが必要となってしまう状況が想定され，あまりにひどい性能となってしまうので，ソフトウェア側にブロック単位でのアクセスを強要することによって，性能が下がりすぎるのを防いでいるというわけです．

B+tree は，固定サイズの連続メモリ領域を node として扱い，node をブロックとしてそのままメモリとディスク間でやりとりするときに変換をほとんど必要としないので，ブロックデバイス上で管理するインデクス構造として広く使われています．Ext4 や xfs などのファイルシステムの多くは，ブロックデバイス上でより柔軟なバイト単位のファイルアクセスやファイルおよびディレクトリの管理を行うためのソフトウェアです．ディレクトリツリーがツリーと呼ばれるように，ファイルシステムは木構造をブロックデバイス上で管理するという点で B+tree と共通点があります．現代ではファイル抽象を大前提とするソフトウェアがほとんどであるため，OS にファイルシステムは内蔵されています．ファイル管理のためのメタデータもやはりブロック単位で管理されます．DBMS はストレージデバイスをファイルシステムを介さずに使うことのある数少ないソフトウェアのひとつです．

Flash メモリ (NAND 型) は，HDD とは別の物質，仕組みを利用した不揮発性ストレージです．具体的にはメモリセルと呼ばれる極小領域に必要な量の電子を閉じ込めることで値を書き，その電子の量を間接的に測定することによって値を読みます．半導体なので HDD と異なり機械的な位置合わせは不要なのですが，素子の寿命管理の制約などから，ブロックデバイスとして抽象化するのが典型的な使い方です．また，複数の単位を並べて並列にアクセスするストライピングによってシーケンシャルアクセスが相対的に高速となる傾向があります．

バイト単位でアクセスできる不揮発性メモリ (NVRAM) は一部の領域で昔から使われていましたが，コスト，容量，性能の面から，SRAM，DRAM，Flash メモリ，HDD，(光学メディア，テープ) で構成される典型的なコンピュータストレージ階層で採用されることはこれまでありませんでした．(実際は DRAM ですら 32bytes や 64bytes などの cache line 単位でアクセスされるのですが，512bytes や 4KiB に比べれば十分小さい単位といえるでしょう...) 2018 年に市場に投入された 3D Xpoint メモリがそこに食い込もうとチャレンジしています．低コスト，大容量，Flash メモリよりも高性能であるようなバイト単位のアクセスが可能な NVRAM が台頭すれば，ブロックデバイスに最適化されたソフトウェアは淘汰される運命ですので，ハードウェアの研究開発には DBMS 屋さんとしては目を光らせておく必要が常にあります．

■コラム: ハードウェアの進化とそれに追随するソフトウェア

ハードウェアの進化によって、効率の良いソフトウェアのアーキテクチャはときどきガラリと変わります。昨今だと、以下の点が大きな変革だと思います:

- CPU の メニーコア化 (とにかく並列に動かさないと性能が出ない時代の到来)
- メインメモリサイズの増大 (データベースがすっぽり収まるケースが増えてきた)
- HDD から Flash memory へ (シーケンシャルアクセスとランダムアクセスの性能差が縮まる、flush request の有無)
- 高速ネットワークの低価格化 (Ethernet 10G とか)
- GPGPU や FPGA の進化 (トランザクション処理にとっては今のところ影響は少ないが、分析系の処理には大きな影響がある)
- NVRAM (3D Xpoint が去年発売にこぎつけましたが、それがメモリ階層の中で不可欠な役割を担うようになるかまだ五分五分と私は見ています)

トランザクション処理という視点で見ると、今の DBMS 研究の世界で見えているものと、現実の DBMS 実装には大きな乖離があります。今の主流の実装は、1990 年代のハードウェアにおいて最適なソフトウェアから継ぎ足し継ぎ足しで進化してきているものが多いと思います。商用の SQL をサポートするようなコードベースが大きい DBMS (Oracle とか MS SQL とか IBM DB2 とか) は、この変化についていくのが大変です。それまで儲けたお金で莫大な投資をしなければ、ついていけません。MySQL、PostgreSQL などは、どこまで付いていけるか、見物ですね。現代のハードウェアに最適な構成を伴って新たに出てくる OSS や商用の DBMS に取って代わられる可能性も十分あります。それでも、DBMS はユーザから最も「枯れている」ことを要求されるソフトウェアのひとつなので、一旦市民権を確立した DBMS 実装の寿命は相対的に長いと思います。