DEIM Forum 2013 F10-1

一般化されたZ-Orderを用いた多次元索引の提案

西村　祥治†　　宮崎　昇幸††

† 日本電気株式会社 〒211–8666 神奈川県川崎市中原区下沼部1753
†† 京都大学大学院　〒606–8501 京都市左京区吉田本町
E-mail: †s-nishimura@bk.jp.nec.com, ††tmiyazaki@db.soc.i.kyoto-u.ac.jp

あらまし　近年，Webサイトのログデータ，センサ情報など複数の属性を持った大量の多次元データを解析し，より有用な情報を抽出することが試みられている．この増え続けるデータを格納する先として，従来からあるリレーショナルデータベースに代わり，容易にシステムをスケール可能なキーバリューストア（KVS）が注目されている．しかし，多次元データ解析をする際，効率的な多次元検索や集約演算が提供されることが望ましいが，KVSは単純なデータ操作しか提供していない．本論文では，KVSに適用可能な一般化されたZ-Orderを用いた多次元索引を提案する．本手法により索引の設計空間を拡張し，問い合わせパターンに応じて最適化可能なスケーラブル多次元データストアを実現する．

キーワード　多次元インデックス, キーバリューストア, スケーラブル, 多次元データベース, 空間充填曲線, Z-Order

1. はじめに

近年，Webサイトのログデータ，センサ情報など複数の属性を持った大量の多次元データを解析し，より有用な情報を抽出することが試みられている．このような増え続けるデータを格納する先として，容易にシステムをスケール可能なキーバリューストア (Key-Value Store, KVS) が注目されている．このようなデータストアの例としては，HBase, Cassandraといったオープンソースプロジェクトを挙げることができる．KVSは，keyとvalueの組みとしてレコードを表現し，key通じてレコードにアクセスするデータストアである．KVSは，keyによってデータを分割することができるため，容易にシステムをスケール（拡大）させることが可能である．KVSは，データの格納方法により，ハッシュ型とkey順序型に分けられる．前者は，ハッシュ関数によりkeyを格納する場所を決めるものである．後者は，keyを辞書式順でレコードを整列して格納するものである．つまり，key順序型KVSは，keyの範囲でデータを分割して格納する．

データ解析のためには，ある属性がある範囲内にあるレコードの取得（範囲検索）や集約（クロス集計）をするが必要である．複数の属性を持っている多次元データを対象としているため，複数の属性の範囲を指定した多次元範囲検索や，集約演算を効率的に実行することができる必要がある．ところが，KVSは単純なレコードアクセス操作しか提供しない．すなわち，keyとvalueのペアを引数にしたPut操作とkeyを引数にvalueを取得するGet操作が基本操作である．key順序型KVSであれば，開始，終了となるkeyを指定してその間にあるレコードセットを取得するScan操作があるが，これは，keyという単一属性に対する範囲検索を提供するだけである．このため，多次元データ解析に欠かせない多次元範囲検索を実現するためには工夫が必要である．

このようなデータストア上で，多次元データを表現する方法として，空間充填曲線による多次元空間から一次元空間への写像を用いる方法がある．空間充填曲線とは，多次元空間上のすべての点をもれなく一筆書きでなぞりつくす曲線で，例としてZ-orderやヒルベルト曲線が知られている．この方法では，インデックスを張りたい属性集合の多次元空間を考え，それを空間充填曲線によりその空間内にあるすべての点，すなわち各属性の属性値の組を一列化し，その並びでデータストアに格納する．

多次元空間を一次元空間に写像するため，多次元空間上では近い2点が写像先の一次元空間では遠くに配置されたり，逆に，遠い2点が近くに配置されたりする．このため，従来研究では，多次元範囲検索を写像された空間上で実現する際，いかに効率的に必要なScan区間を見つけ，いかに無駄なScanを減らすかに主眼があった．このため，多くの研究では，問題を単純にするため，すべての属性は同じ定義域をもち，検索の際，各属性の範囲はほぼ同じという仮定が明示的に，あるいは，暗黙的に置かれている．

しかし，現実のデータは，属性ごとに定義域の幅は異なり，かつ，検索されるときに指定される範囲の幅も異なる．そこで、本論文では、このような状況で検索効率がよいインデッ

クスキーを設計，生成する方法について提案する．ただし、本論文では、データ解析をする際に指定される範囲は，ある程度事前に推測できることがあるという仮定をおく。これは，例えば，位置情報の場合，１ｋｍ四方や５ｋｍ四方，時刻の場合，日ごと，週ごと，月ごと，商品情報の場合，商品カテゴリごとのように，集約したい単位が，設計時にある程度推測可能と考えられるからである．このとき，検索範囲の幅は，毎日の午前 0 時 00 分から翌午前 0 時 00 分までの 24 時間というように，範囲の幅だけでなく始点と終点も固定する場合（離散化可能な場合）だけではなく，今から直近の 24 時間のように始点と終点が固定されていないが，24 時間という範囲の幅だけが事前に分かっている場合も対象とする．

本論文の貢献は以下のとおりである．

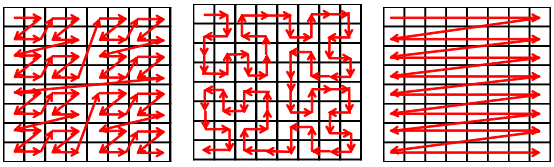
- インデックスキーの設計と多次元範囲検索の効率を明らかにするコストモデルを提案
- コストモデルに基づいて，多次元範囲検索の効率を達成するインデックスキーの設計方針を提案
- Z-order を一般化することでインデックスキーの設計空間を拡張し，上記の設計方針を実現する多次元インデックスキーを提案
- 評価実験により，提案手法の有効性を報告

本論文の構成は以下のとおりである．2 章では，関連研究について紹介する．3 章では，本論文で扱う多次元範囲検索の効率に関する問題について説明する．4 章では，インデックスキーによる多次元範囲検索のコストモデルを提案し，それに基づいて，インデックスキーの設計方針を提案する．5 章では，評価実験をし，提案手法が有効であることを示す．6 章では，全体と今後の課題についてまとめる．

2. 関 連 研 究

多次元空間を空間充填曲線により一次元空間に写像し，多次元データを一次元データとして扱う手法は従来よりよく研究されてきた．例えば，文献 [1] は，空間充填曲線をもちいたインデックス手法を網羅的に紹介している．これらのうち，代表的な空間充填曲線に関して，関連研究を紹介する．

最も使われる空間充填曲線として，Z-order がある．Z-order を用いた手法は，インデックス対象の属性値の組をビット列の組と見たとき，それぞれのビット列を１ビットずつ交互に混ぜ合わせること（bit-interleaving）でインデックスキーを構成する．すなわち，属性値の組として，４ビットのビット列の組 (abcd, WXYZ) が与えられたとき，aWbXcYdZ のように構成する．これは，つまり図 1(a) で示すような曲線として表現される．Z-order の特徴としては，任意の個数のビット列の組からインデックスキーとなるビット列を容易に構成することができる点と，一次元空間に写像された後も，元の空間における多次元データ点同士の近傍性がいくぶんか保存されている点である．このため，位置情報といった空間インデックスに用いられてきた．Z-order を用いたインデックス手法としては，UB-tree [2], zkd-B-tree [3], MD-HBase [4] など多くある．Z-order は１ビットずつ順番に交互に混ぜ合わせて構成することから，インデックス対象の各属性値は，等しいビット幅を持つことを暗に仮定する．しかし，現実のデータでは，これは強い仮定である．そこで，UB-Tree の関連研究では，ビット幅が長い順に属性の優先度付けをし，先頭ビットから順に交互に混ぜ合わせ，ある属性のビット列がすべて混ぜ合わされたならその属性からビットを取ることをスキップする方法を提案している [5]．すなわち，ビット列を左詰めで混ぜ合わせる方法を提案している．



(a) Z-order　(b) ヒルベルト曲線　(c) 複合インデックス

図 1　空間充填曲線

次によく使われる空間充填曲線としては，ヒルベルト曲線が挙げられる．ヒルベルト曲線は，図 1(b) で示すような曲線である．Z-order に比べて複雑であるが，ヒルベルト曲線上で一つ隣のデータ点は，元の多次元空間においても必ず（どこかの次元において）一つ隣のデータ点であることから，Z-order よりも高く元の空間の近傍性を保存する．この性質から，範囲検索性能の効率が高まることが期待される．このことから，Lawder らは，ヒルベルト曲線を用いた手法を提案している [6]．ヒルベルト曲線は図 1(b) から明らかに各次元に対して対称的な構造をしているため，インデックス対象の属性の定義域が異なる場合，工夫が必要である．単純には，0 でパディングして属性のビット列長を一致させる方法がある．Hamilton らは，ゼロパディングによるインデックスキー長を増加を抑えるために，ビット長が短い属性の次元の上位ビットをスキップさせるように構成するヒルベルト曲線を提案している [7]．これは，本質的には，属性値を右詰めしながらインデックスキーを構成する方法といえる．

最後に，リレーショナルデータベースでよく使われる複合インデックスも図 1(c) のような空間充填曲線により多次元空間を一次元空間へ写像した方法とみることができる．すなわち，４ビットの属性値の組 (abcd, WXYZ) が与えられたとき，abcdWXYZ のように，それぞれのビット列を連結すること (bit-concatenation) でインデックスキーを構成する方法である．複合インデックスは，単に各属性のビット列を連結するだけなので，属性ごとのビット長が異なっていても特に構成上の問題は起きない．また，複合インデックスは，どの順番で属性を並べるかで，検索性能が大きく変わることが知られている．一般的には，カーディナリティが小さい属性，

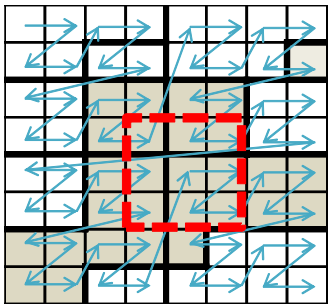


図 2　範囲検索の例

必ず指定される属性を優先する（インデックスキーにおいて上位ビット側に配置する）とよい．しかし，上記の空間充填曲線とくらべ，元の空間の近傍性が保存されにくいため，多次元範囲検索の効率は一般的によくない．

以上のように，属性の定義域が異なる場合のインデックスキーの構成方法についての研究はいくつかある．しかし，検索時に属性の範囲指定が異なる場合にこれらを考慮して検索効率を向上させるようなインデックスキーの構成に言及した研究は，筆者が知る限りない．

3. 問題設定

本章では，まず，空間充填曲線を用いた多次元データベース上での多次元範囲検索について概説する．そして，多次元範囲検索を効率化には，ヒットページ内のレコードヒット率を上げることの重要性を示す．最後に，本論文でどのような仮定のもと，このヒット率の改善をするかを説明する．

3.1 空間充填曲線と多次元範囲検索

多次元空間を空間充填曲線上に写像した多次元データベースにおいて，多次元範囲検索はおおよそ次のように実現される．

各レコードは，インデックスを張りたい属性値の組から空間充填曲線上に対応する一次元の値をもとめ，これをインデックスキーとしてデータベース上に保持される．この時，インデックスキーの値の順に従ってレコードが整列される．レコード数が多い場合，適当なインデックスキーの値でレコードを分割し，ページとしてまとめて保持する．図 2 で具体例を示す．この例では 2 つの属性に対してインデックスを張った例であり，この図の各マス目が 2 つの属性値の組を表す．図中の矢印線が 2 次元空間を一次元に写像する空間充填曲線であり，この例では Z-order を表している．太枠で囲まれた領域が，ページがカバーする範囲である．

このように格納されたレコードに対して，多次元範囲検索を実行する．図 2 では，点線の枠で囲った部分が検索領域である．多次元範囲検索は，大きく以下の 2 つのステップからなる．まず，検索範囲と重なるページの抽出を抽出する．この例では，影が付けられた太枠の領域が検索範囲と重なるページである．これらのページのことを，ここではヒットページと呼ぶ．次に，ヒットページ内を走査する．ヒットページがカバーしている領域は必ずしも検索領域とは重ならない領域も含まれる．そこで，ヒットページ内のレコードを検査し，条件を満たすレコードだけを結果として返す．

多次元範囲検索の実行時間は，ヒットページ数に依存する．なぜなら，ページ内を走査する回数は，ヒットページ数分必要だからである．図 2 で示すように，ヒットページ内のすべてのレコードが検索範囲内に存在するとは限らない．このため，ヒットページ内にあるレコードのヒット率を上げて，できる限り少ないヒットページ数で検索範囲をカバーできることが望ましい．

すべての範囲検索のパターンに対して，最小のヒットページ数を達成するようなレコード配置は不可能である．そこで，本論文では，インデックスキーを設計する時，各属性に対する支配的な検索範囲の幅がわかっていると仮定する．そして，そのような範囲検索のパターンに対してヒットページ数がなるべく少なくなるようなインデックスキーを設計する問題と設定する．

この仮定は，例えば，時間軸だと日，週，月という単位，地理情報だと 100m 四方といったマス目の単位など，ある程度推測できることが多いのではないかという観測に基づく．ただし，各属性の支配的な検索範囲の幅がわかっていることだけを仮定し，その範囲の起点や終点は任意とする．

4. 提案手法

本章では，インデックスキーの設計と範囲検索の効率の関係をヒットページ数の観点でコストモデルを構築する．そして，このコストモデルを元に，インデックスキーの設計方針について説明する．次に，この方針を実現するため，Z-order を一般化し，それをインデックスキーとする手法を提案する．そして，この一般化された Z-order を効率的に生成する Scatter-OR 法を提案する．

4.1 インデックスキーの設計と範囲検索のコスト

範囲検索のコストモデルを構築するにあたり，本論文では，空間充填曲線として解析が容易な Z-order を対象とする．まず，解析に必要な変数を定義する．次に，解析が容易な特殊な例でコストモデルを議論する．最後に，これを一般化し，インデックスキーの設計方針を得る．

属性 i のビット長を x_i とし，その属性の支配的な検索範囲の幅のビット長を w_i とする．するとインデックスキーのビット長 l は $\sum_i x_i$ であり，範囲検索のカーディナリティは $\prod_i 2^{w_i}$ より，そのビット長 w は，$\sum_i w_i$ である．

議論を単純にするため，データの分布は一様とし，各属性の検索範囲の起点，終点はビットアラインされているとする．つまり，属性 i に関する検索範囲は，上位の $x_i - w_i$ ビットは同じで，下位ビットが，0 が w_i 個並んだ値から，1 が w_i 個並んだ値までが指定されたものとする．この空間充填曲

線上における検索範囲の最大値と最小値の差のビット数は，高々kビットであるとする．いいかえれば，空間充填曲線上で，2^kの範囲の幅に，範囲検索でヒットするレコードが含まれているとする．検索範囲のカーディナリティのビット長がwより，$k \geqq w$である．

今，データの分布は一様と考えているので，ページはインデックスキーの全範囲をページ数で等分した区間をカバーしていると考える．この時，各ページがカバーする区間のビット長をcとする．もし，$c \geqq k$であれば，範囲検索にヒットするレコードはすべて一つのページに収まる．次に，$c < k$の場合について考える．今，インデックスキーにおいて，下位からjビット以内に含まれている属性iのビットの数を$v_i(j)$とする．すると，1つのページに含まれる範囲検索にヒットするレコード数は高々$\prod_i \min(w_i, v_i(c))$である．いいかえれば，少なくとも$\prod_i \max(w_i - v_i(c), 0)$のページに範囲検索にヒットするレコードが配置される．

以上より，データの分布が一様で，検索範囲の起点，終点の値がビットアラインされている場合，ページがカバーする区間のビット長がcの時，範囲検索の際にヒットページ数は，$\prod_i \max(w_i - v_i(c), 0)$となる．つまり，レコード数が増えてページ分割が増えるほどcが小さくなり，その結果，ヒットページ数が増える．ヒットページ数が増えるのは，cがkより小さくなった時であるため，kができる限り小さくなるようにインデックスキーを設計したほうがよい．今，$k \geqq w$という関係があるため，$k = w$となるようインデックスキーを設計するのがよい．

では，$k = w$とはどういう場合であるかを具体例を挙げて議論する．二つの6ビットの属性A,Bがあり，それぞれ$[a_0 00000, a_0 11111]$，$[b_0b_1b_2b_3 00, b_0b_1b_2b_3 11]$の範囲を検索する．この時，$w_A = 5$，$w_B = 2$であり，$w = 7$である．次に，この場合での$k$を求める．Z-order上での検索範囲の最小値と最大値は，各属性の起点同士，終点同士をbit interleavingすることで求まる．この時，$a_0b_0 0b_1 0b_2 0b_3 0000$，$a_0b_0 1b_1 1b_2 1b_3 1111$であり，これらの値の上位2ビットが共通であるから$k = 10$である．ページがカバーする区間のビット長が$c \geqq k = 10$である時，1つのページ内で検索範囲内のレコードがすべて含まれる．次に，$c = w = 7$の場合を考える．例として，$[a_0b_0 0b_1 00000000, a_0b_0 0b_1 1111111]$の区間をカバーしているページを考える．このヒットページ内には，属性Bの値が$b_0b_1b_2b_3 00$のような属性Bの検索範囲内のレコードだけでなく，例えば$b_0b_1\overline{b_2b_3}00$のような$b_2$と$b_3$がビット反転した範囲外のレコードも含みうる．つまり，このようなレコードがヒットページ内のレコードヒット率を下げる要因になる．言い換えれば，レコードヒット率を下げる要因となる，属性Bのb_2やb_3のビット位置にあるものをインデックスキーの下位7ビットの位置に配置しないようにすべきである．$w = w_A + w_B$なので，$k = w$となるようにするとは，インデックスキーの下位wビットに，属性Aの下位w_Aビット，属性Bの下位w_Bビットを配置するようにインデックスキーを設計するのがよい．

以下では，各属性の検索範囲がビットアラインされていない場合，データの分布が一般の場合について議論する．いずれの場合も上記の方針でインデックスキーを設計した方がよいという結論が得られる．

a） 各属性の検索範囲がビットアラインされていない場合

属性iの検索範囲がビットアラインされていない場合，その範囲内にあるレコードのビット列のパターンを観察する．ビットアラインされていないため，下位のw_iビットから上位ビットへ桁上がりが発生する (図3)．上位の$x_i - w_i$ビットに着目すると，桁上がりで伝播するビット長に関わらず，そのプレフィックスとなるビットのパターンは，高々2種類しかない．

以上より，属性iの検索範囲だけがビットアラインされていない場合を考えると，ヒットページ数は，最悪の場合でも，すべての属性がビットアラインされている場合に比べて高々2倍で抑えられる．そして，属性の数がdである場合，すべての属性の検索がビットアラインされていない場合のヒットページ数は，最悪$2^d \prod_i \max(w_i - v_i(c), 0)$で抑えられる．

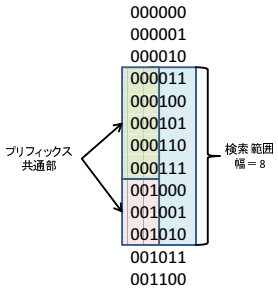


図3 ビットアラインされていない検索範囲とプレフィックスのパターン

さらに詳細に解析するため，この桁上がりで影響を受けるビット長の期待値を見積る．1ビット分だけ桁上がりする場合は全体の1/2，2ビット分の場合は1/4と桁上がりのビット長が長くなるほど指数関数的にそれが起こる事象は稀になる．つまり，その期待値は，$E_{carry,i} = \sum_{j=1}^{x_i - w_i} j/2^j \leqq 2$である．この期待値は，ビットアラインされていて桁上がりが起きない場合($j = 0$)も含んでいる．以上より，ヒットページ数の期待値は，$\prod_i \max(w_i + 2 - v_i(c), 0)$である．この場合でも，$w_i$（および桁上がり伝播のビット長の期待値分）を下位ビット側に寄せるように設計した方がよい．

b） データの分布が一般の場合

上記では，データの分布が一様の場合を議論したので，各ページがカバーするくかn区間のビット長をcと固定した．データの分布が一般の場合，このcがページごとに変化することを意味する．一般にデータが密に集まっている区間をカバーするページのcは小さく，疎である区間をカバーするページのcは大きい．個々のページに関して，cの大きさは変わるが，上記と同じ議論が成り立つ．従って，データの分布

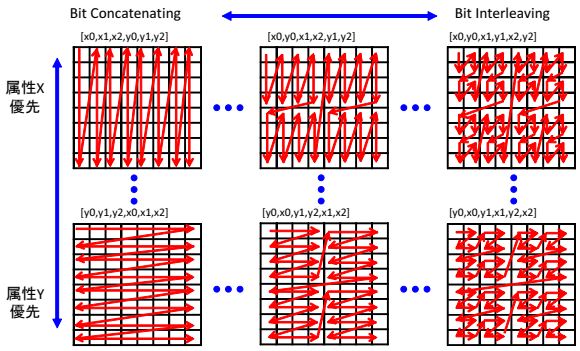


図 4　一般化された Z-order

に関わらず同様の方針でインデックスキーを設計すればよい．

4.2　一般化された Z-order

前節で，各属性に関して，支配的な検索範囲の幅に相当する下位ビットをインデックスキー上で，できる限り下位側に寄せた方がよいことを示した．この方針を実現するため，Z-order を一般化し，インデックスキーの設計空間を拡張する．

従来の Z-order を用いたインデックスキー生成手法は，1 ビットずつ交互にビット列を混ぜ合わせることで，生成していた．このビット列の混ぜ合わせのルールに自由度をもたせ，以下の条件を満たすように生成される空間充填曲線を一般化された Z-order と定義する．

［**定義** 1］（一般化された Z-order） 以下の条件を満たすように，ビット列を混ぜ合わせて生成して得られる空間充填曲線の集合を一般化された Z-order と呼ぶ．

（1） 各属性のビット列の並び順は，写像先のビット列においても入れ替えしない．

（2） 属性ごとのビットの混ぜ合わせ順は任意　□

具体例として，2 つの 4 ビットの属性の組 (`abcd`, `WXYZ`) があったとすると，この属性の組により生成されうる一般化された Z-order の例としては，`abcdWXYZ`，`abcWdXYZ`，`abWXcdYZ`，`aWbXcYdZ`，`WaXbYdZ`，`WXYZabcd` などを挙げることができる．いくつかの一般化された Z-order が描く曲線の例を図 4 に示す．一般化された Z-order の定義より，bit-interleaving も bit-concatenation も一般化された Z-order の特別な例であるし，属性ごとのビット長が異なる際にビット列を右詰め，左詰めすることも一般化された Z-order の特別な例であることは明らかである．すなわち，この一般化によりインデックスキーの設計空間を大幅に広げることができることがわかる．

また，一般化された Z-order を用いたインデックスキーは，各属性の各ビットを，インデックスキー上のどの位置にビットを配置するかを制御することを可能にする．すなわち，前節で示したような，最適化したい範囲検索にあわせたインデックスキー設計を可能とする空間充填曲線群の一つであるといえる．

4.3　一般化された Z-order の性質

Z-order を用いた多次元データストアは，Z-order がもつ性質を利用して効率的に，検索範囲と重なる範囲をカバーするページを見つける．一般化された Z-order は Z-order より自由度が高いが，上記の目的に有用な性質を保存している．このため，既存の Z-order を用いた多次元データストアに一般化された Z-order を比較的軽微な修正で導入可能であると考える．以下の節では代表的な性質を挙げる．

4.3.1　最小値と最大値

Z-order を用いた多次元データストアで，候補のページを絞り込む際，まず，検索範囲内にあるレコードが，Z-order 上のどの範囲内にあるかを調べることである．Z-order の場合，属性ごとの範囲の最小値/最大値のビット列同士を混ぜ合わせることで，Z-order 上での区間の最小値/最大値を求める．一般化された Z-order の場合も，同様のアルゴリズムで，検索範囲内にあるレコードが存在しうる一般化された Z-order 上の範囲を算出することができる．

4.3.2　2 つの矩形領域の交差領域の最小値と最大値

さらに走査範囲を絞り込むには，検索範囲とページのカバー範囲が交差する範囲を見つけることが有用である．その基本となる操作は，2 つの矩形領域が交差する領域が，空間充填曲線上でどの範囲にあたるかを見つけることである．2 つの矩形領域 `A`，`B` があり，それぞれ属性 i の始点と終点を，$[s_{Ai}, e_{Ai}]$，$[s_{Bi}, e_{Bi}]$ とする．Z-order の場合，Z-order 上におけるこの交差領域の最小値，最大値は，それぞれ，各属性について $\max(s_{Ai}, s_{Bi})$，$\min(e_{Ai}, e_{Bi})$ となる値のビット列を交互に混ぜ合わせることで得られた．一般化された Z-order の場合も，同様にして得ることができる．

結局のところ，4.3.1 節と本節の最小値，最大値に関する性質は，属性間でのビットの混ぜ合わせ順に依存しているのではなく，各属性におけるビットの並びが保存されていることに依っているからだと考えられる．

4.3.3　プレフィックスと空間上の位置

Z-order が描く曲線を注意深く観察すると，Z という形がフラクタルな入れ子構造になって描かれていることがわかる．そして，この入れ子構造の階層と Z-order のビットパターンには強い関連がある．2 次元空間の Z-order の場合，図 5 で示すように，インデックスキーを 2 ビットずつ区切った時，何番目の区切りかが，Z-order が描かれた空間において何階層目かを表し，その区切りの値が，その階層における位置を表している．すなわち，インデックスキーのプレフィックスと Z-order が描く曲線がつくる領域が一対一に対応している．

例えば，文献 [8] は，インデックスキーを上位から下位ビット方向にビットパターンを走査することで，効率的に検索範囲と重なる範囲をカバーするページを見つけるアルゴリズムを提案している．また，文献 [4] は，インデックスキーのプリフィックスのパターンを空間分割に利用している．

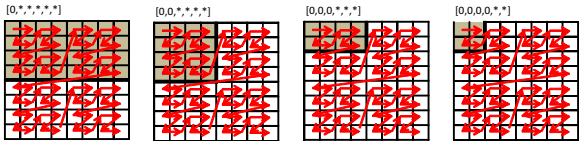


図 5　Z-order のプレフィックスと空間上の位置関係
([y0,x0,y1,x1,y2,x2] の場合)

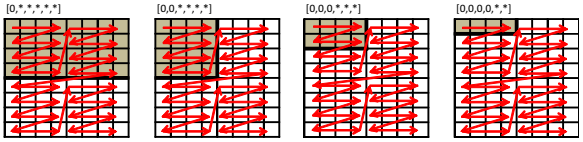


図 6　一般化された Z-order のプレフィックスと空間上の位置関係
([y0,x0,y1,y2,x1,x2] の場合)

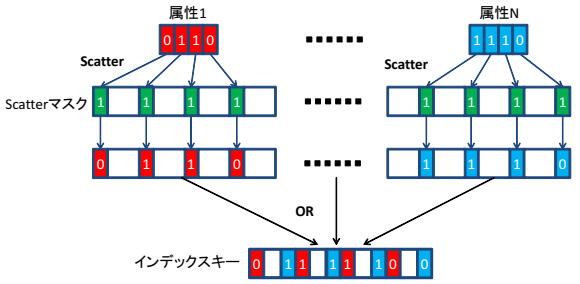


図 7　Scatter-OR 法

一般化された Z-order においても，Z-order と同様にそのプレフィックスと空間上の位置に強い関連がある (図 6)．すなわち，インデックスキーの各ビットが，どの属性のどのビット由来であるかの情報があれば，そのプレフィックスと空間上の位置を一対一に対応付けることが可能である．上記のヒットページ探索手法も空間分割手法も，本質的に，このプレフィックスと空間上の位置の対応関係を利用しているので，これらの手法を一般化された Z-order 向けに拡張可能である．

4.4　Scatter-OR 法

レコードにアクセスする操作をする際には，インデックスキーが必要となるため，インデックスキー生成にコストがあまりかからないことが望ましい．この節では，効率的にインデックスキーが生成可能な手法として，Scatter-OR 法を提案する．

Scatter-OR 法は，その名前のとおり，scatter 演算と OR 演算（論理和演算）を実行することでインデックスキーを生成する方法である．図 7 にその概略を示す．

scatter 演算とは，ビット演算の一種で，与えられた制御ビットマスク（以下，scatter マスク）に従って，与えられたビット列の各ビットを散らす（scatter）操作である．scatter マスクは，操作対象のビット列の各ビットを，結果のビット列のどこに配置するかを，ビットを立てることで表すものである．すなわち，scatter マスクにおいて，最左の 1 ビットがある位置に，対象のビット列の最上位ビットの値を，その次に最左の 1 ビットがある位置に 2 番目の最上位ビットの値を置くことを意味する．

Scatter-OR 法では，インデックスキーを構成する属性ごとに，scatter マスクを用意する．このビットマスクの組がインデックスキーの定義となる．つまり，各属性の各ビットがインデックスキーのビット列上においてどの位置に配置したいかを与えることで，インデックスキー定義を与える．

Scatter-OR 法では，まず，インデックスキー定義として与えられた scatter マスクを，対応する属性に対して scatter 演算を適用して，各属性のビット列がインデックスキーのビット幅に散らされたビット列を得る．次に，これらのビット列に対して OR 演算で 1 つのビット列に集約する．こうして得られたビット列がインデックスキーである．

scatter 演算はよく使われるビット演算であることから，効率的なアルゴリズムが知られていたり，ハードウェア命令として提供されていたりする．例えば，文献 [9] では，基本ビット演算を用いて，ビット幅 n に対して $O(\log n)$ ステップで実行できるアルゴリズムを紹介している．また，Intel(R) の Haswell マイクロアーキテクチャのプロセッサでは scatter 演算（PDEP 命令）をサポートすることが予定されている [10]．また，属性ごとの scatter 演算，OR 演算による集約は命令レベルでの並列性があるので，CPU のメニーコア化や SIMD 命令化による高速化も期待できる．

4.5　Scatter-OR 法の計算量

Scatter-OR 法の計算量を解析する．属性の数を d，属性の平均ビット長を m，インデックスキーのビット長 n とし，このとき，$n = dm$ である．そして，scatter 演算や OR 演算は，N ビット幅で実行する命令が用意されているものとし，その計算量をそれぞれ $O(1)$ とする．

scatter 演算は，インデックスキーのビット長 n を N ビットごとに分割して実行する必要があるので，$O(n/N)$ である．これを属性の数だけ適用するため，scatter 演算フェーズの計算量は $O(dn/N)$ である．

一方，OR 演算は，2 つのビット列の OR をとるのに $O(n/N)$ かかる．そして，すべての属性の scatter 結果をインデックスキーへ集約するのに $d-1$ 回適用が必要である．このため，OR 演算フェーズの計算量は $O((d-1)n/N)$ である．

以上より，Scatter-OR 法の計算量は，$O(dn/N) = O(d^2m/N)$ である．インデックスキーのビット長より，属性の数の方が計算量の影響が大きいといえる．

4.6　インデックスキー定義の検証

一般化された Z-order では，Z-order よりインデックスキー定義の自由度があるため，ユーザがインデックスキー定義をする際にミスが入り込む余地が大きくなる．このため，与えられたインデックスキー定義が妥当であることが，チェッ

クしやすいことが重要である．本提案手法は，インデックスキー定義が満たすべき条件は，比較的簡単にチェックすることが可能である．

a） scatter マスクのビット長

scatter マスクのビット長は，インデックスキーのビット長に一致する必要がある．インデックスキーのビット長は，インデックスキーを構成する各属性のビット長の和である．このことから，単純な和と比較で検証できる．

b） scatter マスクの数

scatter マスクは，属性ごとに関連付けて提供されるべきものであるため，scatter マスクの数は，インデックスキーを構成する属性の数と一致する必要がある．これも簡単な比較で検証できる．

c） scatter マスクで立っているビットの数

ある属性に関連付けられた scatter マスクは，その属性のすべてのビットが，インデックスキー上においてどの位置にあるか示すものである．このことから，scatter マスクで立っているビットの数は，その属性のビット長と一致する必要がある．これは，scatter マスクに対してビットカウントした結果と，その属性のビット長を比較すれば検証できる．

d） scatter マスクで立っているビットの網羅性と排他性

インデックスキーを生成した結果，各属性の各ビットがインデックスキー上で漏れや抜けなく展開されている必要がある．上記の検証結果に加え，2 つ以上の属性のあるビットが，インデックスキー上で同じ位置に展開されていないことを示すことができれば，scatter マスクで立っているビットの網羅性と排他性を示すことができる．これは，すべての scatter マスクを OR で 1 つのビット列に集約した時，すべてのビットが 1 になっていることを検証すればよい．

5. 評　　価

5.1 範 囲 検 索

インデックスキーの定義による範囲検索性能の影響について評価する．

本提案手法を MD-HBase 上に実装し，検索性能を評価した．評価のデータセットとしては，文献 [11] で例として挙がっている通信会社や電力会社の請求書情報に関するスタースキーマを参考に生成したデータセットを用いた．属性数は５で，それぞれ，32，32，16，16，8 ビットである．データ生成の際，属性ごとにデータ分布として単調増加，べき分布，正規分布，一様分布，正規分布を指定し，１億件生成した．

インデックスキーの定義を index A,B,C の 3 通り用意した．この時，それぞれの場合で，ページはおおよそ 76 – 77 ビットの範囲をカバーするように分割された．

ヒットページ数のコスト計算の妥当性を検証するために，ページのカバー範囲を基準に検索範囲のカーディナリティを固定し，それぞれのインデックスキーでのヒットページ件数を評価した．表 1,2 にコスト計算の結果に基づくヒットページ数と実際のページ数をまとめる．インデックスキーが A,B,C となるに従って，各属性の検索範囲の幅分のビットがインデックスキーの上位ビット側に配置されやすくなるため，ヒットページが増加する傾向にある．表 1 を見ると，検索範囲がビットアラインされている場合は，見積もったヒットページ数とほぼ同じヒットページを得られることがわかる．一方，表 2 の検索範囲がビットアラインされていない場合は，多めに見積る傾向がある．原因としては，二つ考えられる．一つめは，ビットアラインされていない時の桁上がりの効果を多めに見積もっているためである．ビットアラインされていない時の桁上がり発生によるヒットページ数の増加を，桁上がりの期待値として属性あたり 2bit を用いた．この期待値は，インデックスキーのビット長が無限とした時の極限値である．二つめの原因としては，実験に用いたデータの特性が挙げられる．この実験ではすべての属性値の分布が一様ではない．このため，ページのカバー範囲のばらつきにより少数のページで広範囲をカバーした結果，実際のヒットページ数が少なくなった可能性がある．しかしながら，ビットアラインされていない場合でも，見積もったページ数が多いほど，実際にアクセスされるページ数も多いことから，インデックスキーを設計する際の評価指標には成り得る．

表 1　インデックスキーの種類とヒットページ数の見積と実際
（検索範囲のビットアラインなし）

	見積ヒットページ数	実ヒットページ数
index A	2^0	1
index B	2^1	3
index C	2^6	55

表 2　インデックスキーの種類とヒットページ数
（検索範囲のビットアラインあり）

	見積ヒットページ数	実ヒットページ数
index A	2^2	1
index B	2^6	10
index C	2^{12}	88

5.2 インデックスキーの生成

インデックスキーを構成する属性の数や属性のビット長を変えて，Scatter-OR 法によるインデックスキー生成性能を評価する．

本論文では，Scatter-OR 法を Java で実装し，表 3 で示す環境で評価した．本評価では，属性のビット長が，8 ビット（byte 型），32 ビット（int 型）の二つを用意し，それぞれインデックスキーを構成する属性の数を変化させたときの生成時間を計測した．計測値は，Scatter-OR 法を 1 万回適用した際の実行時間を 100 回繰り返した時の平均時間とした．結果を図 8 に示す．

4.5 節で解析した通り，属性のビット長が 8 ビット，32 ビッ

表 3　実 験 環 境

CPU	Intel Core i5 1.8GHz
メモリ	8GB
OS	MacOS X 10.8.2
Java	1.6.0_37

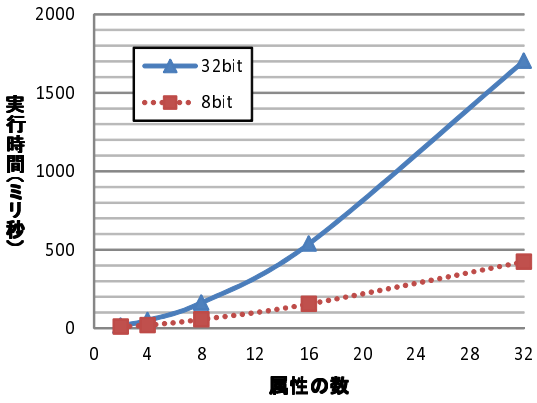


図 8　Scatter-OR 法によるインデックスキー生成性能
実行時間は，scatter-OR 法を 1 万回実行した時の平均値

トの両方の場合で，インデックスキーを構成する属性の数に比例する時間がかかっていることがわかる．また，属性のビット長が 8 ビットと 32 ビットのものを比べると，結果のインデックスキー長に比例した実行時間がかかっていることがわかる．

インデックスキー生成にかかる時間を見てみると，最も時間がかかった，ビット長が 32 ビットの属性を 32 個から生成する場合でも，1 万回実行して 1 秒程度であるため，1 回あたり，0.1 ミリ秒程度であることがわかる．範囲検索をする際，ディスクにレコードを走査する時間や，分散環境でネットワーク越しにレコードを取得する時間のオーダはこれよりも十分に大きいため，この程度の実行時間ではインデックスキー生成がボトルネックとなるとは考えにくい．

6. ま と め

増え続ける複数の属性をもった多次元レコードに対して，スケーラブルに，かつ，効率的な多次元範囲検索が可能なデータストアの実現にむけ，本論文では空間充填曲線を用いてインデックスキーを構成するデータストアに注目した．

本論文では，従来の Z-order では，属性ごとに指定される範囲の幅が均一でない場合，その範囲検索の効率が低下することを，ヒットページ数のコストモデルを構築することで示した．同時に，インデックスキーを設計する際，各属性において，支配的な検索範囲の幅が推測可能である際，その知識を用いて，よりよいインデックスキーを設計する方針も示した．そして，Z-order を一般化し，この設計方針に基づいてインデックスキーを設計する手法を提案した．最後に評価実験を通じて，提案するヒットページ数のコストモデルが，インデックスキーを設計する上での評価指標になり得ることを示した．

今後は，例えば，階層的な集約や条件として指定しない場合があるといった支配的な検索範囲の幅が複数あるような属性を含む場合へとコストモデルを拡張していきたい．また，本論文では十分に言及しなかったが，既存の Z-order を用いたインデックス手法に関連する最適化技術を一般化された Z-order を用いた手法に拡張していきたい．

文　献

[1] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[2] Volker Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*. PhD thesis, TU München, 1999.

[3] J.A. Orenstein and T.H. Merret. A class of data structures for associate searching. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 194–305, 1984.

[4] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *MDM*, pages 7–16, 2011.

[5] Volker Markl and Rudolf Bayer. Processing Relational OLAP Queries with UB-Trees and Multidimensional Hierarchical Clustering. *Proceedings of the International Workshop on Design and Management of Data Warehouses*, 2000:1–10, 2000.

[6] J. K. Lawder and P. J. H. King. Querying Multi-dimensional Data Indexed Using the Hilbert Space-Filling Curve. *SIGMOD Record*, 30(1):19–24, 2001.

[7] Chris H. Hamilton and Andrew Rau-Chaplin. Compact hilbert indices: Space-filling curves for domains with unequal side lengths. *Information Processing Letters*, 105:155–163, 2008.

[8] Tomáš Skopal, Michal Krátký, Jaroslav Pokorný, and Václav Snášel. A new range query algorithm for Universal B-trees. *Information Systems*, 31(6):489–511, September 2006.

[9] Henry S. Warren. *Hacker's Delight Second Edition*, chapter 7, pages 156–157. Addison-Wesley Professional, October 2012.

[10] Intel Corporation. *Intel(R) Architecture Instruction Set Extensions Programming Reference*, August 2012. http://software.intel.com/sites/default/files/319433-014.pdf.

[11] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: the complete guide to dimensional modeling*. Wiley Computer Publishing, second edition, 2002.