4D v11 SQL データ構造

データ構造の知識を深めて対応力を上げる

4D デベロッパカンファレンス 2010 配付資料

目次

４Ｄのデータ構造

データ構造は特に意識する事なくデータベースをデザインする事はできます。4D v11 SQL では、過去のバージョンと比較して速度改善されていますし、現在では容量の大きなディスク装置や、高速なプロセッサを搭載したマシンが比較的安い価格で手に入るようになった現在、細かい事に目を向ける事無くデザインしても問題なく稼働するデータベースシステムは多く存在します。
しかし構造を知る事は、トラブルへの対応力を高めことにもなりますし、何よりも技術者としての探究心を満たす事になります。知っておいて損の無い情報であることは間違いありません。

レコード

レコードの構造は3つの部分から成り立っています。

ヘッダー

レコードの最初の構造はヘッダーと呼ばれる部分で、32 バイト固定の大きさを持つ構造体です。

マイクロストラクチャー

マイクロストラクチャーにはフィールドの情報が収められています。マイクロストラクチャーの大きさはフィールドの数によって変化します。フィールド一つあたり8バイトを使います。

値領域

データはヘッダーとマイクロストラクチャーの間に挟まれた形で格納されます。この値領域は収められるデータによって変化します。各データ型の占める大きさは、次のようになっています。

- ブール：1バイト
- 整数：2バイト
- 64 ビット長整数、実数、日付、時間：8バイト
- フロート：7＋Xバイト
- 文字列：4＋2×n バイト
- BLOB、ピクチャ、テキスト：4×n バイト

倍長整数型のように大きさが決まっているデータ型もありますが、テキスト型のように可変長のデータ型もあります。

フラグメンテーション

こうしたデータがディスク空間に収められるとき、ブロック単位で収められます。ブロックの並びは最初は先頭から整然と収められます。
データが変更されデータの占める大きさ小さくなる時に、元のブロックに収まります。しかしデータの占める大きさが、大きくなったときには、元のブロックには収まらなくなります。その場合、データは別のブロックに移されます。
こうして整然と収められていたレコードのデータがディスク上に断片的に散らばっていきます。これがフラグメンテーションです。
断片化したデータをディスクから読み取る時、整然と並べられたデータと違い、ディスクからの読み取りに無駄が生じてきます。この無駄はミリ秒という非常に小さな時間となって現れる訳ですが、例えば1レコードあたりの無駄が 0.5 ミリ秒であるとき、100 万レコードならロードす

るだけで 200 秒もかかってしまうことになります。実際には、ドライバが先読みを行ってこうした時間的な無駄が生じないようになっているので、そこまでひどい事にはなりません。
フラグメンテーションは構造的には必ず発生してしまいますが、少なくする技術もあります。

エアバッグ

エアバッグは、ブロック内に余裕を持たせてデータを格納する技術です。
データは複数のブロックに格納されます。これは断片化している訳ではありません。連続したレコードデータにエアバッグが仕込まれた状態なのです。整然と順番に格納されたデータであり、無秩序なデータ順になってしまった断片化とは違い、ディスク読み出し時に無駄はありません。
エアバッグを作るため、ブロックに収めるデータを最小サイズと最大サイズの間の大きさになるよう調節して格納します。この制限のために生まれた空間がエアバッグなのです。
エアバッグは未使用状態ですが、いつまでも使わないということではありません。やがて追加されたデータによって埋められていきます。しかしフリースペースという訳ではありません。連続性が保てるようにデータが格納されていくのです。例えば、レコードにフィールドが追加されたようなときに使われます。
単純なフリースペースということではなく、エアバッグはレコードに最適化されています。

テーブル

テーブルは複数のレコードを含んだデータの集合体です。レコードの情報は1024レコードを1ページにまとめた単位で管理されます。

1024 レコード以上の場合

1ページでは、1024 レコードまでしか管理できません。1024 レコード以上ある時には、どのように管理されているのでしょうか。そのために複数のページを収めたページが存在します。

ページ

ページそのものの構造は単純です。1つのヘッダー、1024のエントリー、1024 のデータ長、1024 のアドレスが整然と並んでいます。
エントリーそのものは実体はなく、ページ全体の大きさは、ヘッダーを含めて 12316 バイト、97 ブロックの大きさになります。
アドレスページには、レコードの情報あるいは他のアドレスページの情報が収められています。ページ構造はアドレスページ以外にも使われています。

プライマリアドレスページ

レコードを見つけるために、アドレスページが使われます。アドレスページには、別のアドレスページへリンクされていることもありますが、最終的に目的のレコードを探し出すことができます。
この探索の最初に置かれているアドレスページのことをプライマリアドレスページと呼びます。
では、このプライマリアドレスページは、どのようにして探しだされるのでしょうか。

DTab

DTab は、プライマリアドレステープルを見つけるためのページ構造体です。DTab は、レコードに対するアドレステーブルと同じ働きを持っています。DTab を使って探索をして、プライマリアドレステーブルを見つけることができます。
DTab は、テーブルに対してつくられるページ構造体で、ストラクチャファイル内に格納されています。
では、ストラクチャファイル内に置かれた DTab は、データファイルからどのように見つけることができるのでしょうか。

DTab の探索

DTab は、ストラクチャ内に複数存在します。データファイルから DTab を見つけ出すためには、データファイルのヘッダー情報を使います。ヘッダー情報の中には、DTad のアドレスが収められています。
ヘッダー情報を起点にして、アドレステーブルを辿ります。このアドレステーブルは、DTab への道筋を示すためのものです。

レコードの探索
結局レコードを見つけ出すためには、最初にデータファイルのヘッダー情報から始めることになります。ヘッダー情報からアドレステーブルを経由して、DTab を探します。そして DTab からアドレステーブルを経由してレコードにたどり着くのです。

BLOB 族

BLOB 族は BLOB 型データを含む次の3つのデータ型のことを言います。

- BLOB
- ピクチャ
- テキスト

テキストがバイナリの仲間であることに抵抗があるかもしれません。テキストもテキストという形式のフォーマットで表現されたバイナリとして考えると、この3つは BLOB 族に含まれることが理解できると思います。
BLOB 族の最大の特徴は、レコードの中に収められないことにあります。
他のデータ型は、先に説明したレコード構造の値領域に格納されますが、BLOB 族は値領域には格納されません。その代わりに。BLOB 番号と呼ばれる倍長整数型のデータが値領域に格納されます。

BLOB 族の格納

BLOB 族は、レコードと同じようにデータファイルに格納されます。
BLOB 族データを見つけ出すためには、データファイルのヘッダー情報から始めることになります。ヘッダー情報からアドレステーブルを経由して、DTab を探します。そして DTab からアドレステーブルを経由して BLOB 族データにたどり着くのです。
BLOB 族データがデータファイルと別ファイルになっていると、よく勘違いされますが、格納場所はデータファイルの内部になります。構造的にレコードの外側に保存されているのです。

インデックス

インデックスとは、データを検索しやすくするために、これまで説明したデータ構造の他に用意された構造体です。実際、4D v11 SQL では、データファイルとは別のインデックスファイルとして生成されます。

単純なインデックス

インデックスの考え方はレコードの並び順とは別に、値を探し出しやすいように表を作ることです。実際のレコードを並び替えるのではなく、対応表を作ります。
単純にインデックスを考えてみます。数値が収められたレコードを例にとります。
ある数値を見つけやすくするためには、数値を小さい順から並べかえして整理した表を作れば良いのです。つまり、ソートした対応表を作ることで、目的の値を持ったレコードを素早く見つけ出すことができるようになります。
しかし、単純に表の上から検索を行うと、何万ものレコードがあるような時には、表の下の方にたどり着くまでに、相当な時間が掛かるのは明白です。
実際のインデックスは、このような単純な構造ではないということです。

4D v11 SQL で利用できるインデックスの種類

4D v11 SQL で利用できるインデックスには次のようなものがあります。

- B-tree
- Cluster
- 複合インデックス
- キーワード

さらに、インデックスが設定できるフィールド型で分類するなら、非常に多くの種類のインデックスの種類があることになります。
しかし、インデックス構造は B-tree と Cluster の2つになります。

B-tree

B-tree もページ構造体です。
整数型のインデックスページを例にとりますと、ページには次のデータが収められています。

- キー値
- レコード番号
- サブページ番号

サブページとは、他のインデックスページのことです。
実際に具体的な例をあげて、これらのデータが、どのように利用されるのかを説明します。

B-tree 探索

例えば、14 という数値を持ったレコードを検索したとします。
最初のインデックスページは、数値 8 を持ったレコード情報で終わっています。14 を探すには、次のインデックスページつまりサブページを探さなくてはなりません。8 よりも大きな数値が集められているサブページに移動して探すことになります。そこにはサブページ番号2と記録されていますので、サブページ2番に移動します。
サブページ2番では、3 の次は 15 となって 14 は見つかりません。そこで 13 よりも大きな数値が集められているサブページ番号を見ますと、16 と記録されています。そこでサブページ 16 番に移動します。
サブページ 16 番でようやく 14 という数値をみつけることができました。レコード番号は 600 と記録されています。そして次のサブページ番号はありません。これでレコード 600 をロードして作業終了です。

B-tree インデックスページ構造

仕組みが判ったところで、構造について説明します。
インデックスページの構造にはキーとなる値を収めなくてはなりません。そのためデータ型にインデックスページのサイズは異なります。ここでは倍長整数型の場合を例にとり説明させていただきます。
インデックスページには 256 キーが収められます。キーの大きさは4バイト。レコード番号は4バイト。サブページ番号が4バイト。合計12バイトが、キー1つあたり必要な大きさです。ですから表として 3072 バイトの大きさを持ちます。
これにヘッダの 28 バイトと拡張領域の 16 バイトを合わせるとインデックスページ全体で、3116 バイトになります。これを収めるために 128kb のブロックが 25 個必要になります。

インデックス追加

例をとってインデックスの追加について説明します。

1. 255 キーが登録済み
2. レコード#60 を追加
3. レコード#70 を追加
4. #70 を挿入する余地なし(256 登録済) …
5. …ページを追加するしかない
6. 最初のページを 128 キーづつのページに2分割
7. 中間のキー51 レコード#13 のページを作成して入れ替え
8. 新しいページにサブテーブル番号を格納

ページの数

ページの連鎖レベルが深くなるほど、指数的に収められるキーの数が増えます。

インデックス探索

インデックスの探索の方法は、これまで説明した他のページ構造と基本的に同じです。

データタイプによるページの大きさ

ページの大きさは、データ型やそこに収められるキーにより変化します。

- スカラーデータ
  - 整数
  - 倍長整数
  - 64 ビット整数
  - 実数
  - ブール
  - 時間
- 非スカラーデータ
  - 文字型
  - フロート
  - 複合インデックス

Cluster

Cluster インデックスの構造について説明します。
Cluster インデックスは、キーとなる値にある程度制限があるときに有効に働きます。
例えば、色の名称をフィールドに収めるとして説明を進めます。
フィールドには、青、赤、黄の3色の名称が収められています。このとき、インデックスページには、青、赤、黄のキーが格納され、さらに各キーが収められているレコードの位置を表したデータにリンクしています。この場合、レコード位置はビットテーブルで表現されています。
ビットテーブル以外にも、表現方法はあります。レコード番号を収めた倍長整数配列でも表現することがあります。
ビットテーブルを使うか、倍長整数配列を使うかは、どちらのほうがスペースを使わないかで決定されます。より小さなスペースで済む方式を使います。

Cluster 探索

Cluster インデックスの探索の方法は、これまで説明した他のページ構造と基本的に同じです。