記事
Toshihiko Minamoto · 2021年5月12日 12m read

インデックスを理解する

これは、SQLインデックスに関する2部構成の記事の前半です。

第1部 - インデックスを理解する

インデックスとは？

最後に図書館に行った時のことを思い出してください。 通常そこには、分野別（そして作者順と題名順）に整理された本が並び、それぞれの棚には、本の分野を説明したコードが記載された本立てがあります。 特定の分野の本を収集する場合、すべての通路を歩いて一冊ずつ本の表紙を読む代わりに、目的の分野の本棚に直接向かって選ぶことができるでしょう。

SQLインデックスにもこれと同じ機能があります。テーブルの各行にフィールドの値へのクイック参照を提供することで、パフォーマンスを向上させています。

インデックスの設定は、最適なSQLパフォーマンスを得られるようにクラスを準備する際の主なステップの1つです。

この記事では、次のことについて説明します。

1. インデックスとは何か。いつ、なぜそれを使用するか。
2. どのようなインデックスが存在するか、どのようなシナリオに適しているのか。
3. インデックスの例
4. 作成方法

5. インデックスが存在する場合、どのように扱うのか。

この記事では、Sampleスキーマのクラスを参照します。 このスキーマは以下に示すGitHubリポジトリにあります。また、CachéとEnsembleでインストールされるSamplesネームスペースでも提供されています。

https://github.com/intersystems/Samples-Data

基本

永続プロパティと、永続データから確実に計算されるプロパティにインデックスを作成できます。

Sample.CompanyのTaxIDプロパティにインデックスを作成するとしましょう。
StudioまたはAtelierで、以下のコードをクラス定義に追加します。

```
Index TaxIDIdx On TaxID;
```

これに相当するDDL SQLステートメントは、次のようになります。

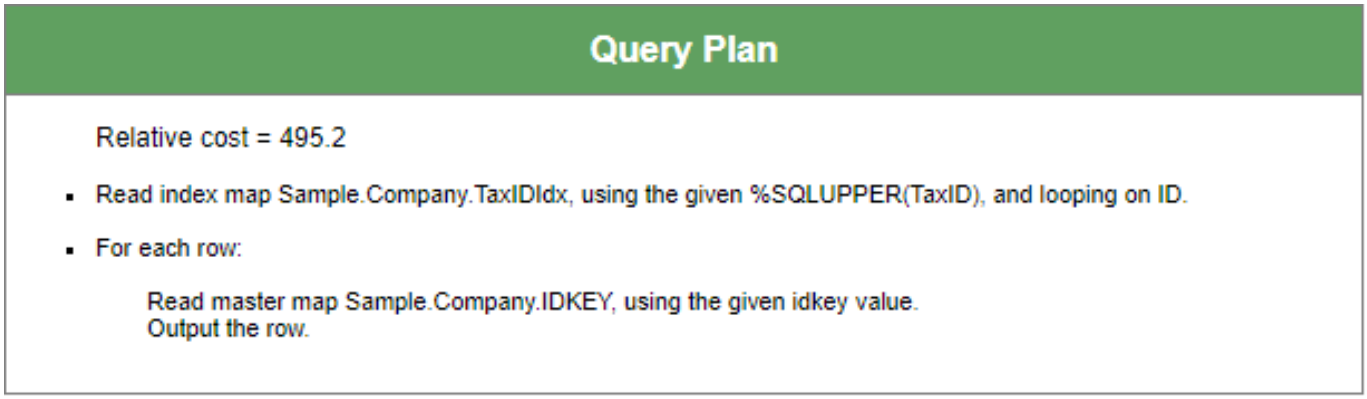
```
CREATE INDEX TaxIDIdx ON Sample.Company (TaxID);
```

デフォルトのグローバルインデックス構造は、次のようになります。

```
^Sample.CompanyI("TaxIDIdx ",<TaxIDValueAtRowID>,<RowID>) = ""
```

通常のデータグローバルのフィールドより、読み取るサブスクリプトが少ないところに注目してください。

「SELECT Name,TaxID FROM Sample.Company WHERE TaxID = 'J7349'」というクエリを見てみましょう。
論理的に単純なクエリです。このクエリを実行するためのクエリプランは、これを反映しています。



このプランは基本的に、指定されたTaxID値を持つ行のインデックスグローバルをチェックし、データグローバル（「マスターマップ」）を参照して一致する行を取得するように指定しています。

ここで、同じクエリを、TaxIDXにインデックスを*使わずに*考察してみましょう。
クエリプランの効率は、予想どおり、低下します。

インデックスがない場合、IRISの基盤のクエリ実行は、メモリを読み取って、テーブルの各行にWHERE句の条件を適用します。論理的に言って、TaxIDを共有する会社はないと思うため、この作業をたった1行のためだけに行っているのです！

もちろん、インデックスを使用するということは

、インデックスと行データがディスクにあるということですので、 条件の内容とテーブルに含まれるデータの量によっては、インデックスを作成してデータを入力する際に、それ固有の問題が生じる可能性もあります。

では、プロパティにはいつインデックスを追加すればよいのでしょうか。

一般的なケースとしては、あるプロパティを頻繁に条件とする場合が挙げられます。
例として個人のSSN（社会保障番号）または銀行口座番号といった識別情報があります。
また、生年月日や口座の資金も考慮できるでしょう。 Sample.Companyに戻ると、高収益の組織に関するデータを収集する場合は、およらくRevenueプロパティのインデックスを作成するとメリットがあるかもしれません。
逆に、条件を付ける可能性が低いプロパティは、インデックス作成にあまり適していません。会社のスローガンや説明などです。
インデックスの種類も検討する必要がある場合を除けば、「シンプル イズ ベスト」なのです！

インデックスの種類

ここでは、6つの主要なインデックスの種類について説明します。標準、ビットマップ、複合、コレクション、ビットスライス、およびデータです。 また、ストリームに基づくiFindインデックスについても簡単に説明します。
上記の例で、標準のインデックスについてすでに触れているため、内容が重複するかもしれません。

クラス定義にインデックスを作成する方法の例をいくつか紹介しますが、新しいインデックスをクラスに追加するというのは、クラス定義に行を追加するだけではありません。
その他の考慮事項については、この記事の第2部で説明します。

それでは、例としてSample.Personを使用しましょう。 PersonにはサブクラスのEmployeeがあることに注目してください。いくつかの例を理解する上で関わってくることです。 Employeeはデータグローバルストレージを Personと共有し、Personのすべてのインデックスは Employeeに継承されます。つまり、Employeeはこれらの継承されたインデックスに、Personのインデックスグローバルを使用しているということです。

これらのクラスにあまり詳しくない場合のために簡単に説明すると、Personには、SSN、DOB、Name、Home（StateとCityを含む埋め込みAddressオブジェクト）、およびOffice（Address）プロパティがあり、FavoriteColorsというリストコレクションがあります。 Employeeには、さらに私が定義したSalaryプロパティもあります。

標準

```
Index DateIDX On DOB;
```

ここでは、「標準」を大まかに使用して、プロパティのプレーンな値（2進数表現ではなく）を格納するインデックスを参照しています。 値が文字列である場合、照合順序（デフォルトではSQLUPPER）で格納されます。

ビットマップやビットスライスのインデックスに比べ、標準のインデックスは人間が読み取れる形式であり、比較的簡単に管理することができます。 グローバルノードは、テーブルの各行に1つずつあります。

以下に、DateIDXがグローバルレベルでどのように格納されるのかを示しています。

```
^Sample.PersonI("DateIDX",51274,100115)="Sample.Employee"_; Date is 05/20/81
```

インデックスの名前の後にある最初のサブスクリプトは日付値で、最後のサブスクリプトはそのDOBを持つPersonのID、そしてグローバルインデックスに格納された値は、このPersonがサブクラスSimple.Employeeのメンバーであることを示していることに注目してください。
このPersonがどのサブクラスのメンバーでもない場合、ノードの値は空の文字列になります。

この基本構造は、ほとんどの非ビットインデックスと一致します。この場合、複数のプロパティのインデックスはグローバルでサブスクリプトをさらに作成し、ノードに複数の値が格納されると、$listbuildオブジェクトが生成されます。以下はその例です。

```
 ^Package.ClassI(IndexName,IndexValue1,IndexValue2,IndexValue3,RowID) =
$lb(SubClass,DataValue1,DataValue2)
```

ビットマップ - プロパティの値に対応するIDセットのビット単位の表現。

```
Index HomeStateIDX On Home.State [ Type = bitmap];
```

ビットマップインデックスは、行ごとに格納される標準のインデックスとは対照的に、ユニーク値ごとに格納されます。

上記の例をさらに詳しく見てみましょう。ID 1のPersonがマサチューセッツ州（MA）に、ID
2がニューヨーク州（NY）に、ID 3がマサチューセッツ州（MA）に、そしてID
4がロードアイランド州（RI）に住んでいるとします。 HomeStateIDXは基本的に次のように格納されます。

(…)	0	0	0	0	-
MA	1	0	1	0	-
NY	0	1	0	0	-
RI	0	0	0	1	-
(…)	0	0	0	0	-

ニューイングランド州にする人のデータを返すクエリが必要な場合、システムはビットマップインデックスの関連する行にビット単位のORをを実行します。 少なくとも、ID
1、3、および4のPersonオブジェクトをメモリに読み込む必要があることにすぐに気づくでしょう。

ビットマップは、WHERE句のAND、RANGE、またはOR演算子に対して効率的です。

ビットマップインデックスの効率が標準インデックスより低くなる前にプロパティに指定できるユニーク値の数に、公式の上限はありませんが、一般的には、最大10,000個程度の値が経験則とされています。 そのため、ビットマップインデックスは、米国の州において効果的であっても、米国の市または郡のビットマップインデックスにはあまり有用とは言えません。

また、ストレージの効率についても、考慮する必要があります。 テーブルへの行の追加や行の削除を頻繁に行う予定であれば、ビットマップインデックスのストレージはあまり効果的ではありません。 上記の例を考察してみましょう。何らかの理由で多数の行を削除し、ワイオミング州やノースダコタ州といった人口の少ない州に住む人がテーブルから消えたとします。
つまり、ビットマップには、ゼロのみの行がいくつか存在することになります。 一方で、ビットマップストレージが大きければ、より多くのユニーク値を格納しなければならなくなるため、大型のテーブルに新しい行を作成していけば、いずれは減速していく可能性があります。

これらの例では、Sample.Personに約150,000行があります。 各グローバルノードには、最大64,000個のIDが格納されるため、ビットマップインデックスグローバルは、MAで3つの部分に分割されます。

```
^Sample.PersonI("HomeStateIDX"," MA",1)=$zwc(135,7992)$c(0,(...))

        ^Sample.PersonI("HomeStateIDX"," MA",2)=$zwc(404,7990,(…))

        ^Sample.PersonI("HomeStateIDX"," MA",3)=$zwc(132,2744)$c(0,(…))
```

特殊ケース: エクステントビットマップ

$<ClassName>とされることの多いエクステントビットマップは、クラスのIDにおけるビットマップインデックスです。IRISはこれを使って行が存在するのかをすばやく検出し、COUNTクエリまたはサブクラスのクエリに役立てています。 これらのインデックスは、ビットマップインデックスがクラスに追加される際に自動的に生成されますが、次のように、クラス定義にビットマップエクステントインデックスを手動で作成することも可能です。

```
Index Company [ Extent, SqlName = "$Company", Type = bitmap ];
```

DDLのBITMAPEXTENTキーワードを使うこともできます。

```
        CREATE BITMAPEXTENT INDEX "$Company" ON TABLE Sample.Company
```

複合 - 2つ以上のプロパティに基づくインデックス

```
Index OfficeAddrIDX On (Office.City, Office.State);
```

複合インデックスは通常、2つ以上のプロパティを条件とするクエリが頻繁に発生する場合に使用できます。

インデックスはグローバルレベルで格納されるため、複合インデックスでは、プロパティの順序が重要になります。 インデックスグローバルの最初のディスク読み取りは保存されるため、選択する頻度の高いプロパティを最初に指定すると、高いパフォーマンス効率を得ることができます。この例では、米国の州の数より都市の数の方が多いため、Office.Cityが最初に指定されています。

あまり選択しないプロパティを最初に指定すると、スペースの効率性が高くなります。 グローバル構造に焦点を当てれば、Stateを最初に指定すると、インデックスツリーのバランスがより高まります。
考えてみれば、各州には多数の市がありますが、1つの州にしか存在しない市もあるのです。

また、いずれかのプロパティのみを条件としたクエリを頻繁に実行するのかどうかを検討することもお勧めします。別のインデックスを定義する手間を省けるからです。

複合インデックスのグローバル構造の例を以下に示します。

```
^Sample.PersonI("OfficeAddrIDX"," BOSTON"," MA",100115)=" Sample.Employee"
```

余談: 複合インデックスかビットマップインデックスか

複数のプロパティで条件付けするクエリの場合、個別のビットマップインデックスを使った方が1つの複合インデックスよりも効果的かどうかを検討することもできます。

ビットマップインデックスが各プロパティに適切に適合するのであれば、2つの異なるインデックスに対してビット演算した方が効率的になる可能性があります。

複合ビットマップインデックスを作成することもできます。これらはユニーク値が、インデックスを作成している複数のプロパティの共通した値となるビットマップインデックスです。 前のセクションで示したテーブルを考察してみましょう。ただし、州の代わりに、州と市のすべての可能な組み合わせ（マサチューセッツ州ボストン、マサチューセッツ州ケンブリッジ、マサチューセッツ州ロサンゼルスなど）を用いたテーブルです。両方の値に適合する行のセルは1となります。

コレクション - コレクションプロパティに基づくインデックス

次のように定義されたFavoriteColorsプロパティがあります。

```
Property FavoriteColors As list Of %String;
```

実演の目的で、インデックスは次のように定義されています。

```
Index fcIDX1 On FavoriteColors(ELEMENTS);

Index fcIDX2 On FavoriteColors(KEYS);
```

ここでは、複数の値を含む単一セルのプロパティをより広く参照するために、「コレクション」を使用しています。 ここでは、List OfとArray Ofプロパティが重要で、必要に応じて区切り付きの文字列も指定できます。

コレクションプロパティは自動的に解析され、インデックスが構築されます。
電話番号などの区切り付きのプロパティでは、このメソッドを明示的に <PropertyName>BuildValueArray(value, .valueArray) と定義する必要があります。

上記のFavoriteColorsの例で考えると、お気に入りの色が青と白であるPersonのfcIDX1は、次のようになります。

```
^Sample.PersonI("fcIDX1"," BLUE",100115)=" Sample.Employee"

(…)

^Sample.PersonI("fcIDX1"," WHITE",100115)=" Sample.Employee"
```

そしてfcIDX2は次のようになります。

^Sample.PersonI("fcIDX2",1,100115)="Sample.Employee"

^Sample.PersonI("fcIDX2",2,100115)="Sample.Employee"

この場合、FavoriteCoslorsはListコレクションであるため、キーに基づくインデックスの有用性は、要素に基づくインデックスよりも低くなります。

コレクションプロパティのインデックスの作成と管理に関するより詳しい考慮事項については、ドキュメントをご覧ください。

https://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=GSQLOPTindices#GSQLOPTindicescollections

ビットスライス - 数値データのビット文字列表現のビットマップ表現

Index SalaryIDX On Salary [Type = bitslice]; //In Sample.Employee

ビットスライスインデックスは、どの行に特定の値が含まれるかを表すフラグを含むビットマップインデックスとは異なり、最初に数値を10進数から2進数に変換し、その後で2進数値の各桁にビットマップを作成します。

上記の例を見てみましょう。現実的に考えられるよう、Salaryを$1000単位として単純化します。つまり、従業員の給与が65であれば、65,000ドルということになります。

ID 1のEmployeeのSalaryを15、ID 2のSalaryを40、ID 3のSalaryを64、ID 4のSalaryを130とします。
この場合、対応するビット値は次のようになります。

	0	0	0	0	1	1	1	
	0	0	1	0	1	0	0	
	0	1	0	0	0	0	0	
	1	0	0	0	0	0	1	

ビット文字列は8桁を超えています。
対応するビットマップ表現（ビットスライスインデックス値）は、基本的に次ように格納されます。

^Sample.PersonI("SalaryIDX",1,1) = "1000" ; Row 1 has value in 1 ’ s place

^Sample.PersonI("SalaryIDX",2,1) = "1001" ; Rows 1 and 4 have values in 2 ’ s place

```
^Sample.PersonI("SalaryIDX",3,1) = "1000" ; Row 1 has value in 4 ’ s place

^Sample.PersonI("SalaryIDX",4,1) = "1100" ; Rows 1 and 2 have values in 8 ’ s place

^Sample.PersonI("SalaryIDX",5,1) = "0000" ; etc…

^Sample.PersonI("SalaryIDX",6,1) = "0100"

^Sample.PersonI("SalaryIDX",7,1) = "0010"

^Sample.PersonI("SalaryIDX",8,1) = "0001"
```

Sample.Employeeまたはその行の給与を変更する演算（INSERT、UPDATES、DELETE）では、これらの各グローバルノードまたはビットスライスを更新する必要があることに注意してください。 ビットスライスインデックスをテーブルの複数のプロパティまたは頻繁に変更されるプロパティに追加すると、パフォーマンスにリスクが生じる可能性があります。 一般的に、ビットスライスインデックスの管理には、標準またはビットマップインデックスの管理よりもコストがかかります。

ビットスライスインデックスは非常に特殊であるため、ユースケースも特殊であり、SUM、COUTN、またはAVGなどの集計計算を実行する必要のあるクエリで使用します。

さらに、数値に対してのみ効果を発揮するため、文字列は2進数の0に変換されます。

クエリの条件をチェックするために、インデックスではなくデータテーブルを読み取る必要がある場合、クエリの実行にビットスライスインデックスは選択されません。 Sample.PersonのNameにインデックスがないとします。 Smithという姓の従業員の平均給与を計算する場合（SELECT AVG(Salary) FROM Sample.Employee WHERE Name %STARTSWITH
'Smith,'
）、WHERE条件を適用するためにデータ行を読み取る必要があるため、ビットスライスは実際には使用されません。

行が頻繁に作成または削除されるテーブルのビットスライスとビットマップインデックスについても、同様のストレージに関する懸念があります。

データ - グローバルノードに格納されているデータのインデックス。

```
Index QuickSearchIDX On Name [ Data = (SSN, DOB, Name) ];
```

前のいくつのかの例で、「Sample.Employee」という文字列がノード自体に値として格納されていることに気づいたかもしれません。
Sample.Employeeは、Sample.Personからインデックスを継承していることを思い出してください。 特にEmployeesをクエリする場合、プロパティ条件に一致するインデックスノードの値を読み取り、そのPersonがEmployeeでもあることを確認します。

また、格納する値を明示的に定義することもできます。 インデックスグローバルノードでデータを定義すると、データグローバルの読み取りも保存できます。頻繁な選択クエリや順序付けクエリに役立ちます。

上記のインデックスを例とすると、 名前の全部または一部を指定された人物に関する識別情報を取得する場合（フロントデスクアプリケーションでクライアントの情報を検索する場合など）、「SELECT SSN, Name, DOB FROM Sample.Person WHERE Name %STARTSWITH 'Smith,J' ORDER BY
Name」というクエリを実行できます。 Nameをクエリ条件としており、取得しようとしている値はすべてQuickSearchIDXグローバルノード内に格納されているため、このクエリを実行するには、グローバルのみを読み取る必要があります。

データ値は、ビットマップまたはビットスライスインデックスと保存できないことに注意してください。

```
        ^Sample.PersonI("QuickSearchIDX"," LARSON,KIRSTEN
A.",100115)=$lb("Sample.Employee","555-55-5555",51274,"Larson,Kirsten A.")
```

iFindインデックス

このようなインデックスを聞いたことがあるでしょうか？ 私にもありません。iFindインデックスは、ストリームプロパティで使用されますが、これを使用するには、クエリにキーワードで名前を指定する必要があります。

もう少し説明することもできますが、このことについては、Kyle Baxterがすでに有用な記事を執筆しています。

フリーテキスト検索：SQL開発者が秘密にしているテキストフィールドの検索方法

［最終更新日: 2020年4月16日 - 可読性を調整。］

#SQL #インデックス付け #パフォーマンス #ベストプラクティス #Caché #InterSystems IRIS

ソースURL:
https://jp.community.intersystems.com/post/%E3%82%A4%E3%83%B3%E3%83%87%E3%83%83%E3%82%AF%E3%82%B9%E3%82%92%E7%90%86%E8%A7%A3%E3%81%99%E3%82%8B