記事
Toshihiko Minamoto · 2021年4月12日 24m read

アトミックでない属性のインデックス作成

(1NF/2NF/3NF)[露] からの引用

> 行と列で特定される位置には、それぞれアプリケーションドメインの値が 1 つだけあります
> (それ以外は何もない)。
> その目的によって、同じ値がアトミックであったり、なかったりします。 例えば、「4286」という値は、
>
> - 「クレジットカードの PIN コード」を意味するのであれば、**アトミック**となります
>   (破損している場合や並び替えられている場合は、使用できません)。
> - 単に「連続する番号」であれば、**非アトミック**となります
>   (いくつかに分割されていたり、並び替えられていても、値は意味を成します)。

この記事では、文字列や日付、($LB 形式の) 単純なリスト、「list of 」、「array of
」といったフィールドの型を伴う SQL クエリのパフォーマンスを向上させる標準的な方法にして検証します。

はじめに

それでは、お馴染みの「電話番号の一覧」から見てみましょう。 以下のように、テストデータを作成します。

```
create table cl_phones(tname varchar2(100), phone varchar2(30));
insert into cl_phones(tname,phone) values ('Andrew','867-843-25');
insert into cl_phones(tname,phone) values ('Andrew','830-044-35');
insert into cl_phones(tname,phone) values ('Andrew','530-055-35');
insert into cl_phones(tname,phone) values ('Max','530-055-35');
insert into cl_phones(tname,phone) values ('Max','555-011-35');
insert into cl_phones(tname,phone) values ('Josh','530-055-31');
insert into cl_phones(tname,phone) values ('Josh','531-051-32');
insert into cl_phones(tname,phone) values ('Josh','532-052-33');
insert into cl_phones(tname,phone) values ('Josh','533-053-35');
```

では、各人物が持つ電話番号をコンマで区切ってまとめたリストを表示します。

```
SELECT
   %exact(tname) tname,
   LIST(phone) phonestr
FROM cl_phones
GROUP BY tname
```

または以下のコードを使います。

```
SELECT
   distinct %exact(tname) tname,
   LIST(phone %foreach(tname)) phonestr
FROM cl_phones
```

tname	phonestr
Andrew	867-843-25,830-044-35,530-055-35
Josh	530-055-31,531-051-32,532-052-33,533-053-35
Max	530-055-35,555-011-35

インデックスを*電話番号*別に作成すれば、特定の番号を使って素早く検索することができます。 このソリューションの唯一の欠点は、名前が重複する可能性があるということです。リストの要素が多くなれば、その分データベースも大きくなります。

従い、同じフィールドに複数の値 (電話番号の一覧やその一部だけを集めた一覧、パスワードなど)
をコンマで区切った文字列として格納しておくと、値別に素早く検索できるので、重宝することがあります。 もちろん、そのようなフィールドに対し普通のインデックスを作成し、長い文字列の中の部分文字列を検索することは可能です。しかし、第 1
にそのような要素が数多く含まれる可能性があり、またインデックスも非常に長くなり得るということ、そして第 2 に、そのようなインデックスを使用しても検索処理をまったく加速化できないという点を考慮する必要があります。

そうでは、どうすれば良いのでしょう。

このように、コレクションを持つフィールドを扱う状況に対処するために、インデックスの特殊な型が導入されています。
コレクションは「実在するもの」(組み込みの *list of* および *array of*) でも「仮想のもの」でも構いません。

組み込みコレクションの場合、それ専用にこのようなインデックスを作成するのはシステムの仕事であり、開発者はそのプロセスに干渉できません。
但し、仮想コレクションであれば、インデックスの作成は開発者が担います。

そのような「仮想」コレクションには、シンプルなコンマ区切りの文字列、日付、単純なリストなどがあります。

コレクションのインデックスには、以下の構文を用います。

```
INDEX idx1 ON (MyField(ELEMENTS));
```

もしくは

```
INDEX idx1 ON (MyField(KEYS));
```

インデックスは、***propertyname****BuildValueArray*
メソッドを使って作成しますが、これは開発者が自分で実装しなければいけません。

通常、このメソッドのシグネチャは以下のようになります。

```
ClassMethod propertynameBuildValueArray(value, ByRef valueArray) As %Status
```

引数の意味

- *value* – 複数の要素に分割されるフィールド値
- *valueArray* – 個別の要素を含む結果の配列。

配列は、キー / 値の組み合わせで、以下の形式が使われます。
array(key1)=value1
array(key2)=value2
などなど。

先ほども触れましたが、このメソッドはシステムにより組み込みコレクション用に自動的に生成されるものであり、属性が **[Final]** であるため、開発者は再定義できません。

それでは、これらのインデックスを作成し、SQL での活用方法を見てみましょう。

注意:
前の例で作られた構造が残らないよう、新しいインデックスを作成する前にグローバルやクラスのストレージスキーマを空にしておくことをおすすめします。

コンマ区切りの文字列

では、次のクラスを作成しましょう。

```
Class demo.test Extends %Persistent
{

Index iPhones On Phones(ELEMENTS);

Property Phones As %String;

ClassMethod PhonesBuildValueArray(
  value,
  ByRef array) As %Status
{
  i value="" {
    s array(0)=value
  }else{
    s list=$lfs(value,","),ptr=0
    while $listnext(list,ptr,item){
      s array(ptr)=item
    }
  }
  q $$$OK
}

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(phones)
    select null union all
    select 'a' union all
    select 'b,a' union all
    select 'b,b' union all
    select 'a,c,b' union all
    select ',,'
  )
  zw ^demo.testD,^demo.testI
}
}
```

ターミナルで **Fill**() メソッドを呼び出します。

```
USER>d ##class(demo.test).Fill()
^demo.testD=6
^demo.testD(1)=$lb("","")
^demo.testD(2)=$lb("","a")
^demo.testD(3)=$lb("","b,a")
^demo.testD(4)=$lb("","b,b")
^demo.testD(5)=$lb("","a,c,b")
^demo.testD(6)=$lb("",",,")
^demo.testI("iPhones"," ",1)=""
^demo.testI("iPhones"," ",6)=""
^demo.testI("iPhones"," A",2)=""
^demo.testI("iPhones"," A",3)=""
^demo.testI("iPhones"," A",5)=""
^demo.testI("iPhones"," B",3)=""
^demo.testI("iPhones"," B",4)=""
^demo.testI("iPhones"," B",5)=""
^demo.testI("iPhones"," C",5)=""
```

ご覧のとおり、インデックスには文字列全体ではなく、その特定の部分だけが含まれています。
従って開発者は好きなように長い文字列を複数の部分文字列に分割できます。
コンマ区切りの文字列以外にも、XML や JSON ファイルなども使えます。

ID	電話番号
1	(null)
2	a
3	b,a
4	b,b
5	a,c,b
6	,,

それでは、「a」を含む部分文字列をすべて見つけましょう。 このためには、**'%xxx%'** や **['xxx'**
といった述語を使うことがルールになっています。

```
select * from demo.test where Phones [ 'a'
select * from demo.test where Phones like '%a%'
```

しかし、この場合、**iPhones** インデックスは使用されません。
使用するには、特殊な述語を使う必要があります。

```
FOR SOME %ELEMENT(ourfield) (%VALUE = elementvalue)
```

これを考慮して、以下のようなクエリを使います。

```
select * from demo.test where for some %element(Phones) (%value = 'a')
```

特殊なインデックスを使ったおかげで、このクエリの処理速度は最初のバージョンを使った場合よりも大幅に速くなります。

もちろん、以下のようにもっと複雑な条件を使うこともできます。

```
(%Value %STARTSWITH 'a')
(%Value [ 'a' and %Value [ 'b')
(%Value in ('c','d'))
```

(%Value is null)

ここでさっと魔法をかけます。。。

機密情報を隠す

通常、**BuildValueArray** メソッドでは、*value* を使って *array* 配列を作成します。

でも、このルールに従わなかったらどうなるのでしょう？

次の例を試してみましょう。

```
Class demo.test Extends %Persistent
{

Index iLogin On Login(ELEMENTS);

Property Login As %String;

ClassMethod LoginBuildValueArray(
  value,
  ByRef array) As %Status
{
  i value="Jack" {
    s array(0)="test1"
    s array(1)="test2"
    s array(2)="test3"
  }elseif value="Pete" {
    s array("-")="111"
    s array("5.4")="222"
    s array("fg")="333"
  }else{
    s array("key")="value"
  }
  q $$$OK
}

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(login)
    select 'Jack' union all
    select 'Jack' union all
    select 'Pete' union all
    select 'Pete' union all
    select 'John' union all
    select 'John'
  )
  zw ^demo.testD,^demo.testI
}
}
```

ID	Login
1	Jack

ID	Login
2	Jack
3	Pete
4	Pete
5	John
6	John

そして、肝心なのはここです！ 次のクエリを実行してみましょう。

```
select * from demo.test where for some %element(Login) (%value = '111')
```

ID	Login
3	Pete
4	Pete

結果、一部のデータだけがテーブルに表示されています。また、インデックスでは非表示になったデータがいくつかありますが、それでも検索はできます。 これはどのように使えばいいのでしょう？

例えば、特定のユーザーがアクセスできる一連のパスワードを、インデックスから (1 つだけではなく)
すべて非表示にすることができます (通常 1 つだけ非表示にすることはできません)。 また、SQL
を使って開かれては困るという機密情報があれば、それをすべて非表示にすることもできます。
もちろん、これは、GRANT column-privilege を実行するなど、他の方法でも行えます。
しかし、その場合は、protected フィールドにアクセスするためのストアドプロシージャを使う必要があります。

機密情報を隠す(続き)

私たちのテーブルのデータとインデックスを含むグローバルを見れば、「5.4」や「fg」といったキーの値が表示されていないのが分かります。

```
^demo.testD=6
^demo.testD(1)=$lb("","Jack")
^demo.testD(2)=$lb("","Jack")
^demo.testD(3)=$lb("","Pete")
^demo.testD(4)=$lb("","Pete")
^demo.testD(5)=$lb("","John")
^demo.testD(6)=$lb("","John")
^demo.testI("iLogin"," 111",3)=""
^demo.testI("iLogin"," 111",4)=""
^demo.testI("iLogin"," 222",3)=""
^demo.testI("iLogin"," 222",4)=""
^demo.testI("iLogin"," 333",3)=""
^demo.testI("iLogin"," 333",4)=""
^demo.testI("iLogin"," TEST1",1)=""
^demo.testI("iLogin"," TEST1",2)=""
^demo.testI("iLogin"," TEST2",1)=""
^demo.testI("iLogin"," TEST2",2)=""
^demo.testI("iLogin"," TEST3",1)=""
^demo.testI("iLogin"," TEST3",2)=""
^demo.testI("iLogin"," VALUE",5)=""
^demo.testI("iLogin"," VALUE",6)=""
```

そもそも、なぜこれらを定義しているのか？

その疑問に答えるには、インデックスを少し変更して、テーブルを作成しなおします。

```
Index iLogin On (Login(KEYS), Login(ELEMENTS));
```

これで、グローバルは見た目が変わります (インデックス付きのグローバルのみ表示しています)。

```
^demo.testI("iLogin"," -"," 111",3)=""
^demo.testI("iLogin"," -"," 111",4)=""
^demo.testI("iLogin"," 0"," TEST1",1)=""
^demo.testI("iLogin"," 0"," TEST1",2)=""
^demo.testI("iLogin"," 1"," TEST2",1)=""
^demo.testI("iLogin"," 1"," TEST2",2)=""
^demo.testI("iLogin"," 2"," TEST3",1)=""
^demo.testI("iLogin"," 2"," TEST3",2)=""
^demo.testI("iLogin"," 5.4"," 222",3)=""
^demo.testI("iLogin"," 5.4"," 222",4)=""
^demo.testI("iLogin"," FG"," 333",3)=""
^demo.testI("iLogin"," FG"," 333",4)=""
^demo.testI("iLogin"," KEY"," VALUE",5)=""
^demo.testI("iLogin"," KEY"," VALUE",6)=""
```

これで、見事にキーの値と要素の値の両方が格納されています。
これは、今後どのように活用できるのでしょう？

例えば、先ほどのパスワードを使った例では、パスワードをその有効期限や他の情報と一緒に保存することができます。 クエリでは、次のようにして行います。

```
select * from demo.test where for some %element(Login) (%key='-' and %value = '111')
```

データをどこに保存するかは、あなた次第です。但し、キーは一意ですが、値はそうでないことを覚えておきましょう。

また、「コレクション」インデックスは、普通のインデックスと同様、追加データを保存するのに使用できます。

```
Index iLogin On (Login(KEYS), Login(ELEMENTS)) [ Data = (Login, Login(ELEMENTS)) ];
```

この場合、上のクエリはデータにアクセスせずに、インデックスからすべてのデータを取得してくれるので、時間を節約できます。

日付 (時刻など)

日付はコレクションにどう関連しているのか、と疑問に思う方がいると思います。 答えはズバリ「ダイレクトに関連している」です。それは、日付や月、年だけによって検索する必要のあることが頻繁にあるためです。
通常の検索では効果はありません。ここで必要なのは、まさに「コレクションベース」の検索なのです。

ここで、次の例を見てみましょう。

```
Class demo.test Extends %Persistent
```

```
{

Index iBirthDay On (BirthDay(KEYS), BirthDay(ELEMENTS));

Property BirthDay As %Date;

ClassMethod BirthDayBuildValueArray(
  value,
  ByRef array) As %Status
{
  i value="" {
    s array(0)=value
  }else{
    s d=$zd(value,3)
    s array("yy")=+$p(d,"-",1)
    s array("mm")=+$p(d,"-",2)
    s array("dd")=+$p(d,"-",3)
  }
  q $$$OK
}

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(birthday)
    select {d '2000-01-01'} union all
    select {d '2000-01-02'} union all
    select {d '2000-02-01'} union all
    select {d '2001-01-01'} union all
    select {d '2001-01-02'} union all
    select {d '2001-02-01'}
  )
  zw ^demo.testD,^demo.testI
}
}
```

ID	BirthDay
1	01.01.2000
2	02.01.2000
3	01.02.2000
4	01.01.2001
5	02.01.2001
6	01.02.2001

これで、簡単に、とてもスピーディに 日付の特定の部分を検索できます。 例えば、以下のようにすれば 2 月生まれの人を全員選択できます。

```
select * from demo.test where for some %element(BirthDay) (%key='mm' and %value = 2)
```

ID	BirthDay
3	01.02.2000
6	01.02.2001

単純なリスト

Caché DBMS では、単純なリストを対象とした特殊なデータ型 (**%List**) が備えられています。どの区切り記号を使うかを決められなくて困っている開発者は、文字列の代わりに使えるので便利です。

このフィールドの使い方は、文字列の使い方にとても似ています。

それでは、簡単な例を見てみましょう。

```
Class demo.test Extends %Persistent
{

Index iList On List(ELEMENTS);

Property List As %List;

ClassMethod ListBuildValueArray(
  value,
  ByRef array) As %Status
{
  i value="" {
    s array(0)=value
  }else{
    s ptr=0
    while $listnext(value,ptr,item){
      s array(ptr)=item
    }
  }
  q $$$OK
}

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(list)
    select null union all
    select $LISTBUILD('a') union all
    select $LISTBUILD('b','a') union all
    select $LISTBUILD('b','b') union all
    select $LISTBUILD('a','c','b') union all
    select $LISTBUILD('a,,',null,null)
  )
  zw ^demo.testD,^demo.testI
}
}
```

ID	List
1	(null)
2	a
3	b,a
4	b,b
5	a,c,b
6	"a,,",,

注意
Caché では、論理、ODBC、ディスプレイモード (まとめてデータディスプレイオプション) の 3 種類のデータプレゼンテーションモードがサポートされています。

ここでは、要素の区切り記号が使われていないため、要素の中では好きな文字を使うことができます。

ODBC モードで **%List** 型のフィールドを表示するときは、ODBCDELIMITER パラメーターが区切り記号として使用されます (デフォルトで「,」と同じ)。

例えば、そのようなフィールドを表示する場合

```
Property List As %List(ODBCDELIMITER = "^");
```

ID	List
1	(null)
2	a
3	b^a
4	b^b
5	a^c^b
6	a,,^^

要素の検索は、コンマ区切りの文字列を検索する場合と同じです。

```
select * from demo.test where for some %element(List) (%value = 'a,,')
```

ID	List
6	a,,^^

%INLIST のオプションは、未だ「コレクション」インデックスを使用できないため、上に紹介した例よりも処理速度が遅くなります。

```
select * from demo.test where 'a,,' %inlist List
```

コレクション

それでは、上の例を書き直しましょう。単純なリストの代わりにコレクションのリストを使います。

```
Class demo.test Extends %Persistent
{

Index iListStr On ListStr(ELEMENTS);

Property ListStr As list Of %String;

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(liststr)
    select null union all
    select $LISTBUILD('a') union all
    select $LISTBUILD('b','a') union all
    select $LISTBUILD('b','b') union all
```

```
    select $LISTBUILD('a','c','b') union all
    select $LISTBUILD('a,,',null,null)
  )
  zw ^demo.testD,^demo.testI
}
}
```

この例は、中身はほぼ同じですが、微妙に違う点がいくつかあります。 以下にご注意ください。

- フィールドの COLLATION 値や *array* のキーおよびインデックス値は、グローバルに保存される前に適切な形に変換されます。両方の例のグローバルインデックスの値、特に、NULL 値の表記を見比べてください。
- **BuildValueArray** メソッドがないため、要素の値しか使えません (キーは使用できない)。
- フィールドの型に特殊なコレクションクラス (**%ListOfDataTypes**) が使われている。

配列コレクション

上述のとおり、リストではキーを使用できませんが、 配列を使えば、この欠点を解決できます。

では、次のクラスを作成しましょう。

```
Class demo.test Extends %Persistent
{

Index iArrayStr On (ArrayStr(KEYS), ArrayStr(ELEMENTS));

Property str As %String;

Property ArrayStr As array Of %String;

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(str)
  select null union all
  select 'aaa' union all
  select 'bbb' union all
  select 'bbb' union all
  select 'ccc' union all
  select null
  )
  &sql(insert into demo.test_ArrayStr(test,element_key,arraystr)
    select 1,'0','test1' union all
    select 1,'1','test2' union all
    select 1,'2','test3' union all
    select 2,'0','test1' union all
    select 2,'1','test2' union all
    select 2,'2','test3' union all
    select 3,'-','111' union all
    select 3,'5.4','222' union all
    select 3,'fg','333' union all
    select 4,'-','111' union all
    select 4,'5.4','222' union all
    select 4,'fg','333' union all
    select 5,'key','value' union all
    select 6,'key','value'
  )
```

```
  zw ^demo.testD,^demo.testI
}
}
```

説明が必要と思われる点を列挙しておきます。

- データは、以前と同じように *class**D*** の *^name* (データ) と *class**I*** の *^name* (インデックス) という 2 つのグローバルに格納されています。
- クラスは 1 つですが、テーブルは既に 2 つあります (お馴染みの demo.test と追加でもう 1 つ demo.testArrayStr)。
- demo.testArrayStr では、SQL を使って簡単に配列データにアクセスできるほか、以下のフィールドが設けられています。それぞれの名前は一部事前定義されています。
  elementkey – キーの値 (事前定義されているフィールド名)。
  ArrayStr – 要素の値。
  test – 親テーブル demo.test へのリンク。
  ID – test||elementkey 形式のサービスプライマリキー (事前定義されているフィールド名)。
- フィールドの型には特殊なコレクションクラス (**%ArrayOfDataTypes**) が使われています。

以上を踏まえ、**Fill**() メソッドを実行すると、テーブルの中身は以下のとおりになります。

ID	str
1	(null)
2	aaa
3	bbb
4	bbb
5	ccc
6	(null)

ID	test	elementkey	ArrayStr
1||0	1	0	test1
1||1	1	1	test2
1||2	1	2	test3
2||0	2	0	test1
2||1	2	1	test2
2||2	2	2	test3
3||5.4	3	5.4	222
3||-	3	-	111
3||fg	3	fg	333
4||5.4	4	5.4	222
4||-	4	-	111
4||fg	4	fg	333
5||key	5	key	value
6||key	6	key	value

テーブルが 1 つから 2 になった結果、テーブル間で JOIN
を使うことを余儀なくされているように思えますが、そうではありません。

Caché DBMS が提供する SQL のオブジェクト拡張機能について考慮すると、

demo.test テーブルの **str** フィールドにある文字列のうち、キーが「-」で要素の値が「111」であるものを表すテストクエリは、以下のようになります。

```
select test ID,test->str from demo.test_ArrayStr where element_key='-' and
arraystr='111'
```

もしくはこちら

```
select %ID, str from demo.test where test_ArrayStr->element_key='-' and
test_ArrayStr->arraystr='111'
```

ID	str
3	bbb
4	bbb

ご覧のとおり、ここは至って単純で、**JOIN** も使われていません。これは、すべてのデータが 1 つのグローバルに格納され、Caché がこれらのテーブルの「関係」を把握しているためです。

これらのフィールドを両方のテーブルから参照できるのは、まさにこれが理由です。 実際、demo.test テーブルに **testArrayStr** フィールドはありませんが、それを使って関連するテーブルにアクセスすることができます。

結論

この記事で説明したインデックス作成メカニズムは、一部のシステムクラスで広範に使用されています。それには、テキストストリームのインデックスを作成し、SQL による検索を可能にする %Stream.GlobalCharacterSearchable クラスなどが含まれます。
クラスコレクションのインデックス作成に関するトピックは、非常に多岐にわたるため、意図的に割愛しています (組み込み、格納、ストリーム、ユーザー定義、コレクションのコレクションなど)。 また、SQL による操作が不便な場合もあります。これを踏まえ、著者はそのようなコレクションが必要になることは、一部の極めて稀な状況を除き、ほぼないであろうと判断しました。 フルテキスト検索もまた別のトピックであり、SQL を使う場合には独自のインデックスと操作法を伴うものであるため、本記事ではカバーしておりません。
最後に、著者は、SqlListType や SqlListDelimiter といったプロパティパラメーターの使用例も省いておりますが、好奇心旺盛の方は実際に試す方法を見い出していただけると思います。

役に立つリンク

- コレクションクラス
- コレクションのインデックス作成
- コレクションのインデックス作成、SQL を使いコレクションに対してクエリを実行する方法
- 一部の %ELEMENT について
- オブジェクト / リレーショナル接続
- SQL の拡張機能
- 特別な機能

これは、こちらの記事を英訳したものです。 [@Evgeny Shvarov]、翻訳作業にご協力いただきありがとうございました。

この記事は、Habrahabr露 でもお読みいただけます。

#ObjectScript #SQL #インデックス付け #オブジェクトデータモデル #パフォーマンス #Caché

ソースURL:
https://jp.community.intersystems.com/post/%E3%82%A2%E3%83%88%E3%83%9F%E3%83%83%E3%82%AF%E3%81%A7%E3%81%AA%E3%81%84%E5%B1%9E%E6%80%A7%E3%81%AE%E3%82%A4%E3%83%B3%E3%83%87%E3%83%83%E3%82%AF%E3%82%B9%E4%BD%9C%E6%88%90