高性能スキーマの定義

Version 2024.1
2024-06-03

InterSystems Corporation　1 Memorial Drive　Cambridge MA 02142　www.intersystems.com

高性能スキーマの定義
InterSystems IRIS Data Platform　Version 2024.1　2024-06-03

インターシステムズ社の製品に関するサポートやご質問は、以下にお問い合わせください：

InterSystems Worldwide Response Center (WRC)
Tel:　+1-617-621-0700
Tel:　+44 (0) 844 854 2917
Email:　support@InterSystems.com

目次

図一覧

テーブル一覧

1

クエリ・オプティマイザで使用するテーブル統計

InterSystems SQL テーブルの最大限のパフォーマンスを実現するには、代表的なデータ・メトリックまたは予測したデータ・メトリックを作成して使用します。最適化は、このテーブルに対するクエリの実行に大きな効果をもたらす可能性があります。この章では、以下に示すパフォーマンス最適化の考慮事項について説明します。

- ExtentSize、Selectivity、および BlockCount では、テーブルにデータを移入する前に、テーブル・データの見積もりを指定します。このメタデータは、今後のクエリを最適化するために使用されます。
- テーブルのチューニングの実行では、データが移入されているテーブル内の代表的なテーブル・データを分析します。これにより生成されたメタデータは、今後のクエリを最適化するために使用されます。
- テーブル・チューニングの計算値として、エクステント・サイズ、選択性、異常値の選択性、平均フィールド・サイズ、ブロック・カウントなどがあります。
- テーブル・チューニング統計のエクスポートと再インポート

1.1 ExtentSize、Selectivity、および BlockCount

特定の SQL クエリを実行する最も効率的な方法をクエリ・オプティマイザが決定するときに、考慮される項目が 3 つあります。

- ExtentSize は、クエリ内で使用されるテーブルごとの行数です。
- Selectivity は、クエリで使用される列ごとに計算された個別値の割合です。
- BlockCount は、クエリで使用される SQL マップごとのカウント数です。

クエリ・オプティマイザが正しい決定を下すためには、これらの値が正しく設定されていることが重要です。

- これらの統計は、テーブルにデータを移入する前に、クラス（テーブル）の定義を行うときに明示的に設定できます。
- テーブルに代表的なデータを移入したら、テーブル・チューニングを実行してこれらの統計を計算できます。
- テーブル・チューニングを実行した後、明示的な値を指定して、計算された統計をオーバーライドすることもできます。

明示的に設定した統計と、テーブルのチューニングで生成された結果を比較できます。テーブルのチューニングで行った想定により、クエリ・オプティマイザの結果よりも適切でない結果が得られた場合は、テーブルのチューニングで生成された統計ではなく、明示的に設定した統計を使用してください。

スタジオのクラス編集ウィンドウに、クラスのソース・コードが表示されます。ソース・コードの末尾にはストレージ定義が表示されますが、この定義にはプロパティごとに ExtentSize クラスと Selectivity クラス (該当する場合は OutlierSelectivity クラスも) が含まれています。

1.1.1 ExtentSize

テーブルの ExtentSize 値は、そのテーブルに保存されている行数 (概数) を表します。

開発時には、ExtentSize の初期値を指定できます。ExtentSize を指定しない場合、既定値は 100,000 です。

通常、データを移入したときに予想されるこのテーブルのサイズを大まかに見積もります。正確な値である必要はありません。この値は、さまざまなテーブルのスキャンの相対コストを比較するときに使用します。関連するテーブル間の ExtentSize の相対値が正確な比率を表すようにすることが最も重要です (つまり、小さいテーブルの値は小さく、大きいテーブルの値は大きくする必要があります)。

- CREATE TABLE には、以下の例に示すように、テーブル内の行数の予想値を指定するための %EXTENTSIZE パラメータ・キーワードが用意されています。

  SQL

  ```
  CREATE TABLE Sample.DaysInAYear (%EXTENTSIZE 366,
                                   MonthName VARCHAR(24),Day INTEGER,
                                   Holiday VARCHAR(24),ZodiacSign VARCHAR(24))
  ```

- テーブルの永続クラス定義では、ストレージ定義内で ExtentSize パラメータを指定できます。

  ```
  <Storage name="Default">
  <Data name="MyClassDefaultData">
  ...
  <ExtentSize>200</ExtentSize>
  ...
  </Storage>
  ```

  この例では、MyClass クラスのストレージ定義を示しています。ここでは、ExtentSize に値 200 を指定しています。

テーブルが実際の (現実的な) データを持つ場合は、管理ポータルのテーブルのチューニング機能により、自動的に ExtentSize 値を計算して設定できます。詳細は、後述の "テーブルのチューニング" セクションを参照してください。

1.1.2 Selectivity (選択性)

InterSystems SQL テーブル (クラス) では、各列 (プロパティ) に Selectivity 値が関連付けられています。列の Selectivity 値は、標準的な列の値を検索するクエリの結果として返される、テーブル内の行の割合を示します。異常値があるかどうかによって、InterSystems SQL で Selectivity 値がどのように解釈されるかが変化することがあります。

Selectivity は、ほぼ同じ数量の個別値に基づきます。例えば、テーブルに、値が “heads” と “tails” にほぼ均等に分散している CoinFlip 列があるとします。CoinFlip の Selectivity の値は、50% になります。State などのさらに特有なプロパティに対する Selectivity 値は、通常、小さい割合になります。

すべての値が同じであるフィールドの Selectivity は 100% です。これを判断するため、オプティマイザは最初に少数のレコードをテストします。すべてのレコードのフィールド値が同じである場合、インデックスが付けられていないフィールドの値がすべて同じであるという仮定を裏付けるために、ランダムに選択した最大 100,000 個のレコードをテストします。ランダムに選択した 100,000 個のレコードのテストでフィールドの他の値が検出されない可能性がある場合は、Selectivity を手動で設定する必要があります。

一意として定義された (すべての値が異なる) フィールドの Selectivity は 1 です (1.0000% の Selectivity と混同しないでください)。例えば、RowID の Selectivity が 1 であるとします。

開発時には、テーブルのクラス定義の一部であるストレージ定義内に Selectivity パラメータを定義して、この値を設定することができます。

```
<Storage name="Default">
<Data name="MyClassDefaultData">
...
<Property name="CoinFlip">
<Selectivity>50%</Selectivity>
</Property>
...
</Storage>
```

通常は、アプリケーションで使用される場合に予測される Selectivity 値の概数を指定します。ExtentSize と同様に、正確な値である必要はありません。InterSystems IRIS の多くのデータ型クラスでは、Selectivity に妥当な既定値が指定されます。

また、SetFieldSelectivity() メソッドを使用して、特定のフィールド (プロパティ) の Selectivity 値を設定することもできます。

テーブルに実際の (現実的な) データが含まれている場合は、管理ポータルのテーブル・チューニング機能を使用して、その Selectivity 値を自動的に計算し、設定できます。テーブル・チューニングでは、他のどの値よりもはるかに多く存在する値である異常値がフィールドにあるかどうかが確認されます。異常値がある場合、テーブル・チューニングでは、別の Outlier Selectivity の割合が計算され、その異常値の割合に基づいて Selectivity が計算されます。異常値の割合によって、Selectivity 値が著しく変わることがあります。

Selectivity はクエリの最適化に使用されます。SELECT クエリで指定されたフィールド、およびビューの SELECT 節で指定された同じフィールドには、同じ Selectivity 値が使用されます。ビューの行の分散がソース・テーブルとは異なる場合があることに注意してください。これはビュー・フィールドの選択性の正確性に影響する可能性があります。

1.1.3 BlockCount

永続クラスをコンパイルする際、クラス・コンパイラは、ExtentSize とプロパティ定義に基づいて、各 SQL マップで使用されるマップ・ブロックの概数を計算します。これらの BlockCount 値は、テーブル・チューニング機能の **[マップ・ブロック・カウント]** タブで表示できます。この BlockCount は、テーブル・チューニングでは `Estimated by class compiler` として識別されます。ExtentSize を変更した場合、その変更が BlockCount 値に反映されていることを確認するには、SQL の [テーブルチューニング] ウィンドウを閉じてから、開き直す必要があります。

テーブル・チューニングを実行すると、SQL マップごとの実際のブロック・カウントが測定されます。特別な指定がない場合、テーブルのチューニングの測定値により、クラス・コンパイラの概算値が置き換えられます。テーブル・チューニングで測定されたこれらの値は、指定した BlockCount 値と区別するために、クラス定義では負の整数として表されます。詳細は、以下の例を参照してください。

```
<SQLMap name="IDKEY">
 <BlockCount>-4</BlockCount>
</SQLMap>
```

テーブル・チューニングで測定された値は、テーブル・チューニングでは正の整数として表され、`Measured by TuneTable` として識別されます。

クラス定義内では、明示的に BlockCount の値を定義できます。以下の例に示すように、ブロック・カウントを明示的に正の整数として指定できます。

```
<SQLMap name="IDKEY">
 <BlockCount>12</BlockCount>
</SQLMap>
```

クラスを定義するときには、マップについての BlockCount の定義を省略するか、BlockCount を明示的に正の整数として指定するか、BlockCount を明示的に NULL として定義できます。

- BlockCount を指定しない場合、または BlockCount を 0 として指定した場合は、クラス・コンパイラによってブロック・カウントが試算されます。テーブル・チューニングを実行すると、クラス・コンパイラによって試算された値が置換されます。
- 正の整数の BlockCount を明示的に指定した場合、テーブル・チューニングを実行しても、その明示的な BlockCount 値は置換されません。明示的なクラス定義のブロック・カウント値は、テーブル・チューニングでは正の整数として表され、`Defined in class definition` として識別されます。これらのブロック・カウント値は、テーブル・チューニングをその後実行しても変更されません。
- NULL の BlockCount を明示的に指定した場合、SQL マップは、クラス・コンパイラによって試算された BlockCount 値を使用します。BlockCount はクラス定義で “定義” されているので、テーブル・チューニングを実行しても、試算されたこの BlockCount 値は置換されません。

すべての InterSystems SQL マップ・ブロックのサイズは 2048 バイト (2KB) です。

以下の状況では、テーブルのチューニングによる BlockCount の測定は行われません。

- テーブルが、配列コレクションまたはリスト・コレクションで投影された子テーブルの場合。このような子テーブルに対する BlockCount 値は、親テーブルのデータ・マップに対する BlockCount と同じになります。
- グローバル・マップが、リモート・グローバル (別のネームスペース内のグローバル) の場合。その代わりに、クラス・コンパイル時に使用された見積もり BlockCount が使用されます。

1.2 テーブルのチューニング

テーブル・チューニングは、テーブル内のデータを検証し、ExtentSize (テーブル内の行の数)、各フィールドの個別値の相対的な分散、および平均フィールド・サイズ (各フィールドの値の平均長) に関する統計を返すユーティリティです。また、SQL マップごとの BlockCount も生成します。テーブルのチューニングがこの情報を使用して、テーブルとその各フィールドに関連付けられているメタデータを更新することを指定できます。その後、クエリ・オプティマイザは、これらの統計を使用して、クエリの最も効率的な実行プランを特定できます。

外部テーブルでテーブル・チューニングを使用する場合は、ExtentSize のみが計算されます。テーブル・チューニングでは、外部テーブルについてフィールドの Selectivity 値、平均フィールド・サイズ、またはマップの BlockCount 値を計算することはできません。

1.2.1 テーブル・チューニングの実行時

各テーブルに対してテーブル・チューニングを実行する前に、そのテーブルに代表的な数量の実データを移入しておく必要があります。一般に、テーブル・チューニングは、アプリケーション開発の最終段階として、データを “実動環境” に移す前の 1 回しか実行する必要はありません。テーブル・チューニングはメンテナンス・ユーティリティではありません。実動データで定期的には実行しないでください。ただし、挿入、更新、または削除によりテーブルの構造が大きく変更された場合は、テーブル・チューニングを再実行する必要があります。一般的に、変更の種類に関係なく、テーブルの 20% が変更された場合は、以下の特性が変化した可能性が高いため、テーブル・チューニングを再実行する必要があります。

- 相対的なテーブル・サイズ:テーブル・チューニングは、データの代表的なサブセットを分析することを前提にしています。代表的なサブセットであれば、このサブセットはデータ・セット全体のごく一部でかまいません。結合またはその他のリレーションシップに関係するテーブルの ExtentSize が相対的にほぼ同等なサイズを維持している場合は、テーブル内の行数が変わっても、テーブル・チューニングでは引き続き妥当な結果が得られます。結合テーブル間の比の桁数が変わる場合は、ExtentSize を更新する必要があります。SQL オプティマイザは、テーブル結合順序を最適化するときに ExtentSize を使用するため、これは JOIN 文で重要です。原則として、クエリで指定された結合順序に関係なく、小さいテーブルは大きいテーブルより前に結合されます。

- 均等な値の分散:テーブルのチューニングは、各データ値が均等に分散することを前提にしています。異常値を検出した場合は、異常値以外の各データ値が均等に分散することを前提とします。テーブルのチューニングでは、各フィールドの現在のデータ値を分析することで、Selectivity を確定します。実データにおける均等な分散は常に概算です。個別データ値の数とそれらの相対的な分散の変動が通常の範囲であれば、テーブル・チューニングを再実行する必要はありません。ただし、フィールドで使用される可能性がある値の数(レコードに対する個別値の比率)や、1 つのフィールド値が全体に占める割合が大幅に変わった場合は、Selectivity が正確でなくなる可能性があります。1 つのフィールド値を持つレコードの割合が著しく変わると、テーブル・チューニングで異常値が指定されたり、異常値の指定が解除されて、計算された Selectivity が大幅に変わることがあります。1 つのフィールドの Selectivity が、実際のデータ値の分散を反映しなくなった場合には、テーブルのチューニングを再実行する必要があります。
- InterSystems IRIS の大幅なアップグレード、または新規サイトでのインストールによって、テーブル・チューニングの再実行が必要になる場合があります。

効率的なクエリ処理はテーブル・チューニングの統計に依存するため、InterSystems IRIS ではテーブル・チューニングを実行します。

- 自動 最初の SELECT クエリがチューニングされていないテーブルに対して実行される場合。自動テーブル・チューニングは、非常に特定の状況でのみ実行されます。
- 手動 サポートされている任意のユーザ・インタフェースを使用してテーブル・チューニングを実行する場合。

1.2.2 自動テーブル・チューニング

InterSystems IRIS は、テーブルに対して初めて SELECT クエリが実行されると、以下の状況に従って、自動的にテーブル・チューニングを実行します。

- テーブルは、テーブル・チューニングの前の実行によっても、エクステント・サイズまたは選択性の値の設定によっても、チューニングされたことがない。一意でないすべてのフィールドで、選択性の値が "" である必要がある。
- クエリが実行されたことがない。クエリがキャッシュされたことがない。クエリの実行により、オプティマイザが呼び出され、クエリ・プランが生成される。
- テーブルは通常のテーブルである。ビュー、またはその他のテーブル以外のデータ構造ではない。
- テーブルはシャーディングされていない。
- テーブルはリンク・テーブルではない。
- クエリがビルド・ルーチンの一環として実行されていない。

1.2.3 手動テーブル・チューニング

テーブル・チューニングを実行するための手動インタフェースは 3 つあります。

- 管理ポータルの SQL インタフェースの [**アクション**] ドロップダウン・リストを使用します。テーブルのチューニングを 1 つのテーブルに対して実行することも、複数のテーブルに対して実行することもできます。
- 1 つのテーブル、または現在のネームスペース内のすべてのテーブルに対して、$SYSTEM.SQL.Stats.Table.GatherTableStats() メソッドを呼び出します。
- 1 つのテーブルに対して、SQL コマンド TUNE TABLE を発行します。

テーブル・チューニングによって、チューニング対象のテーブルを参照するクエリ・キャッシュが削除されます。TUNE TABLE コマンドにはクエリ・キャッシュのリコンパイルのオプションが用意されており、テーブル・チューニングの新しい計算値を使用してクエリ・キャッシュを再生成できます。

テーブルが読み取り専用データベースにマップされている場合は、テーブル・チューニングを実行できず、エラー・メッセージが生成されます。

テーブルのチューニング機能を実行した後、結果として得られる ExtentSize および Selectivity の値は、クラスのストレージ定義に保存されます。スタジオでストレージ定義を確認するには、**[ビュー]** メニューから **[ストレージの表示]** を選択します。すると、クラスのソース・コードの下部にストレージが表示されます。

注釈　テーブル・チューニングによって SQL のパフォーマンスが低下することはまれです。テーブル・チューニングは実動データに対して実行できますが、プロダクション・システム上ではなく、テスト・システム上で実データを使用して実行することをお勧めします。オプションの [システムモード] 構成パラメータを使用して、現在のシステムがテスト・システムと実働システムのいずれであるかを示すことができます。[システムモード] が設定されている場合、これは、管理ポータルのページの最上部に表示され、$SYSTEM.Version.SystemMode() メソッドを使用して返すことができます。

1.2.3.1 管理ポータルからのテーブル・チューニング

管理ポータルからテーブル・チューニングを実行するには、以下の手順を実行します。

1. **[システムエクスプローラ]**、**[SQL]** の順に選択します。ページ上部の現在のネームスペース名をクリックしてネームスペースを選択し、表示されたリストからネームスペースを選択します (ユーザごとに管理ポータルの既定ネームスペースを設定できます)。
2. 画面の左側にあるドロップダウン・リストから **[スキーマ]** を選択するか、**[フィルタ]** を使用します。**[スキーマ]** および **[フィルタ]** の使用方法の詳細は、このドキュメントの “管理ポータルの SQL インタフェースの使用法” の ″SQL スキーマの参照″ を参照してください。
3. 以下のいずれかを行います。
   - 1 つのテーブルのチューニング : **[テーブル]** カテゴリを展開し、リストからテーブルを選択します。テーブルを選択したら、**[アクション]** ドロップダウン・リストをクリックし、**[テーブル・チューニング情報]** を選択します。テーブルの現在のエクステント・サイズおよび選択性に関する情報が表示されます。テーブル・チューニングを実行したことがない場合は、エクステント・サイズ = 100000 が表示され、選択性、外れ値の選択性、外れ値、または平均フィールド・サイズに関する情報は表示されません (選択性が 1 の RowID 以外)。また、マップ・ブロック・カウントに関する情報は `Estimated by class compiler` として示されます。

     [選択性] タブから、**[テーブルチューニング]** ボタンを選択します。テーブルに対してテーブル・チューニングが実行され、テーブル内のデータに基づいてエクステント・サイズ、選択性、異常値の選択性、異常値、および平均フィールド・サイズの値が計算されます。マップ・ブロック・カウントに関する情報は、`Measured by Tune Table` として示されます。

     1 つのテーブルに対するテーブル・チューニングは常にバックグラウンド・プロセスとして実行され、完了するとテーブルが更新されます。これにより、タイムアウトの問題が防止されます。このバックグラウンド・プロセスが実行されている間は、「`in process`」というメッセージが表示されます。バックグラウンド・プロセスの実行中に、**[閉じる]** ボタンを利用して [テーブルチューニング] ウィンドウを閉じることができます。
   - スキーマ内のすべてのテーブルのチューニング : **[アクション]** ドロップダウン・リストをクリックし、**[スキーマ内の全テーブルをチューニング]** を選択します。[テーブルチューニング] ボックスが表示されます。**[完了]** ボタンを選択して、スキーマ内のすべてのテーブルに対してテーブル・チューニングを実行します。テーブル・チューニングが完了すると、このボックスに **[完了]** ボタンが表示されます。**[完了]** を選択して、[テーブルチューニング] ボックスを終了します。

SQL の [テーブルチューニング] ウィンドウには、**[選択性]** と **[マップ・ブロック・カウント]** という 2 つのタブがあります。これらのタブには、テーブル・チューニングによって生成された現在の値が表示されます。これらのタブを使用して、テーブル・チューニングによって生成された値とは異なる値を手動で設定することもできます。

[選択性] タブには以下のフィールドがあります。

- [現在のテーブルエクステントサイズ]。このフィールドには、異なるテーブル・エクステント・サイズを入力するための **[編集]** ボタンがあります。
- **[クラスを最新状態に保つ]** チェック・ボックス。テーブルのチューニング、[テーブルチューニング] インタフェースでユーザが入力した値、またはテーブル・チューニング・メソッドによって生成された統計に対する変更はすべて、即座にクラス定義に反映されます。
  - このチェック・ボックスにチェックが付いていない場合 (いいえ)、変更されたクラス定義の最新フラグが設定されません。これは、クラス定義が古くてリコンパイルが必要であることを示します。これが既定値です。
  - このチェック・ボックスにチェックが付いている場合 (はい)、クラス定義は、最新のものとしてフラグが設定されたままになります。これにより、テーブルのクラス定義がリコンパイルされる可能性が低くなるため、実働システムで統計を変更する場合に推奨されるオプションです。
- [フィールド名]、[選択性]、[メモ]、[異常値の選択性]、[異常値]、および [平均フィールド・サイズ] の列がある [フィールド] テーブル。[フィールド] テーブルの見出しをクリックして、その列の値でソートすることができます。[フィールド] テーブルの行をクリックして、そのフィールドの選択性、異常値の選択性、異常値、および平均フィールド・サイズの値を手動で設定できます。

[ブロックカウントのマップ] タブには以下のフィールドがあります。

- [SQL マップ名]、[ブロックカウント]、および [ブロックのカウント元]の列がある [マップ名] テーブル。インデックスの SQL マップ名は SQL インデックス名です。これは、永続クラスのインデックス・プロパティ名とは異なる場合があります。

  個々のマップ名をクリックして、そのマップ名のブロック・カウント値を手動で設定できます。

[選択性] タブから、**[テーブルチューニング]** ボタンをクリックして、このテーブルに対してテーブル・チューニングを実行できます。

1.2.3.2 メソッドを使用したテーブル・チューニング

$SYSTEM.SQL.Stats.Table.GatherTableStats() メソッドを使用して、現在のネームスペースでテーブル・チューニング機能を実行することができます。

- GatherTableStats("Sample.MyTable") は、1 つのテーブルに対してテーブル・チューニングを実行します。
- GatherSchemaStats("Sample") は、指定したスキーマのすべてのテーブルに対してテーブル・チューニングを実行します。
- GatherTableStats("*") は、現在のネームスペースですべてのテーブルに対してテーブル・チューニングを実行します。

GatherTableStats() メソッドを使用すると、以下のエラー・メッセージが生成されることがあります。

- 存在しないテーブル : `DO $SYSTEM.SQL.TuneTable("NoSuchTable",0,1)`

  `No such table 'SQLUser.NoSuchTable'`
- ビュー : `DO $SYSTEM.SQL.Stats.Table.GatherTableStats("Sample.MyView")`

  `'SQLUser.MyView'`

並列処理がサポートされている場合、GatherTableStats("*") または GatherSchemaStats("SchemaName") を実行すると、複数のプロセスを使用して、複数のテーブルが並列でチューニングされます。

1.2.4 シャード・テーブルでのテーブルのチューニングの実行

テーブルのチューニングをシャード・テーブルで実行する場合、テーブル・チューニング操作は、各シャードに転送され、そこでテーブルのシャードに対して実行されます。テーブルのチューニングは、その呼び出し元のマスタ・ネームスペースで実行されることはありません。クラス定義をシャードにエクスポートした未シャード化テーブルでテーブルのチューニングが実行されると、そのテーブルはシャード・テーブルに結合されるため、テーブルのチューニング操作は各シャードに転送され、マスタ・ネームスペースでも実行されます。

シャード・テーブルでテーブルのチューニングを実行する際には、以下のガイドラインに従う必要があります。

- シャード・ローカル・テーブルではなく、シャード・マスタ・テーブルをチューニングします。
- EXTENTSIZE および BLOCKCOUNT の値は、すべてのシャードの合計値ではなく、各シャードの値です。
- $SYSTEM.SQL.Stats.Table.Export() および $SYSTEM.SQL.Stats.Table.Import() を使用する場合、シャード・ローカル・テーブルではなく、シャード・マスタ・テーブルのテーブル統計をエクスポートまたはインポートします。
- シャード・テーブルの統計を収集する際に、RecompileCachedQueries 引数が無視され、テーブルのクエリ・キャッシュが必ず削除されます。
- シャード・テーブルの統計の収集では、シャード・マスタとシャード・ローカルの両方のクラス/テーブル定義でテーブル統計が定義されます。クラス定義のテーブル・チューニング・メタデータを手動で編集する場合は、シャード・マスタ・クラスの定義を変更した後、シャード・マスタ・クラスをリコンパイルするという手順を推奨します。シャード・マスタ・クラスをコンパイルすると、シャード・マスタのテーブル統計がクラスのシャード・ローカル・バージョンにコピーされます。

GatherTableStats() または GatherSchemaStats() で logFile パラメータを指定すると、シャード・マスタ・インスタンスのログ・ファイルには、以下のように指定テーブルのエントリが含まれます。

- シャード・テーブル：`TABLE: <tablename> Invoking TuneTable on shards for sharded table <tablename>`
- シャード化されていないテーブル：`TABLE: <tablename> Invoking TuneTable on shards for mapped non-sharded table <tablename>`

シャード・インスタンスそれぞれで、同じ名前のログ・ファイルが mgr/<shard-namespace> ディレクトリに作成され、対象シャードの対象テーブルについて、テーブルのチューニングに関する情報が記録されます。ディレクトリ・パスがこのログ・ファイルに指定されている場合、そのパスはシャードでは無視され、ファイルは常に、mgr/<shard- namespace> に格納されます。

1.3 テーブルのチューニングの計算値

テーブルのチューニング操作では、テーブル内の代表的なデータに基づいてテーブルの統計が計算および設定されます。

- エクステント・サイズ。テーブルにある実際の行数である場合とそうでない場合があります。
- テーブルにあるプロパティ (フィールド) ごとの選択性。オプションで、個々のプロパティの選択性が計算されないようにすることができます。
- 1 つの値が、他の値よりずっと一般的に出現するプロパティの異常値の選択性。効率的なクエリでは、異常値の最適化を使用できるほか、RTPC を使用し、実行時にクエリ・パラメータに基づいてプランを選択することもできます。
- 特定のプロパティ特性を識別するプロパティごとの注意事項。
- プロパティごとの平均フィールド・サイズ。

- テーブルの SQL マップ名、ブロック・カウント、およびブロック・カウントのソース。

1.3.1 エクステント・サイズと行カウント

管理ポータルからテーブル・チューニング機能を実行する場合は、ExtentSize が、現在テーブル内にある行の実際の数になります。既定では、GatherTableStats() メソッドは、ExtentSize として実際の行数も使用します。テーブルに多数の行が含まれる場合は、より少ない行数で分析を実行することをお勧めします。SQL TUNE TABLE コマンドを使用して %SAMPLE_PERCENT を指定すると、合計行数のパーセンテージについてのみ分析を実行できます。このオプションを使用すると、多数の行が含まれるテーブルに対して実行したときのパフォーマンスが向上します。この %SAMPLE_PERCENT の値は、代表的なデータのサンプリングに十分な大きさにする必要があります。ExtentSize が 1000 未満の場合、TUNE TABLE は、%SAMPLE_PERCENT の値に関係なくすべての行を分析します。

指定された ExtentSize は、実際の行数より少なくても多くてもかまいません。ただし、ExtentSize は、現在のテーブル・データの実際の行数を大幅に超えてはいけません。ExtentSize を指定すると、テーブル・チューニングは、その行数に応じた行 ID を推定して、サンプリングを実行します。ExtentSize が実際の行数を大幅に超えていると、サンプリングされる行 ID の大部分は実際の行データに対応しなくなります。この場合は、フィールドの選択性が計算できなくなります。その代わりに、テーブル・チューニングは、指定された ExtentSize を CALCULATED ExtentSize とし、より小さい数を SAMPLESIZE としてリストします。テーブル・チューニングは存在しない計算値に対して <Not Specified> を返します。

0 の ExtentSize を設定できます。これは、データは入力されないが、クエリの結合などその他の目的で使用される予定のテーブルがある場合に適切です。ExtentSize を 0 に設定すると、InterSystems IRIS は各フィールドの選択性を 100% として設定し、各フィールドの平均フィールド・サイズを 0 として設定します。

1.3.2 Selectivity と Outlier Selectivity

テーブルのチューニングでは、Selectivity (選択性) が、プロパティ (フィールド) の値ごとに割合として計算されます。これは、データをサンプリングすることで行われるため、Selectivity は常に推定値になります (正確な値ではありません)。Selectivity は、すべてのプロパティ値が均一である (または均一になる) という仮定に基づきます。これは、ほとんどのデータに対して合理的な仮定です。例えば、一般的な人口テーブルのほとんどのデータ値は典型的な値になります。特定の誕生日はデータの約 0.27% (365 分の 1) で出現し、男性と女性はおよそ半々 (50%) の存在になります。一意として定義されたフィールドは、1 の Selectivity になります (1.0000 (1%) の Selectivity と混同しないようにしてください)。Selectivity の割合は、ほとんどのプロパティに対して十分に必要を満たします。

いくつかのプロパティに対して、テーブルのチューニングでは Outlier Selectivity (異常値の選択性) も計算します。これは、サンプル内で、その他のデータ値よりも頻出する 1 つのプロパティ値の割合です。テーブルのチューニングは、あるデータ値の頻度とその他のデータ値の頻度に大幅な差があるときにのみ、Outlier Selectivity を返します。テーブル・チューニングで返される異常値は、データ値の分散に関係なく、多くても 1 つのテーブルについて 1 つです。異常値が選択されると、テーブルのチューニングはこの値を Outlier Value として表示します。NULL は <Null> として表示されます。テーブル・チューニングで異常値の選択性が返される場合、通常の選択性は、依然として全行のうちの異常でない各データ値の割合となります。

テーブル・チューニングの初回サンプルで返される値が 1 つだけであっても、追加のサンプリングで複数の個別値が返される場合は、これらのサンプリング結果によって通常の選択性が変更されます。例えば、990 個の値の初回ランダム・サンプルで検出された値が 1 つだけであっても、その後のサンプリングで他の個別値の単一インスタンスが 10 個検出されたとします。この状況では、初回の外れ値が選択性の値に影響し、選択性の値は 1/1000 (0.1%) に設定されます。10 個の非外れ値はそれぞれ 1000 個のレコード中に 1 つしか発生しないためです。

最も一般的な Outlier Selectivity の例として、NULL を許可するプロパティが挙げられます。あるプロパティに対する NULL のレコード数が、そのプロパティに対する特定のデータ値を持つレコード数を大幅に上回ると、NULL が異常値になります。以下に、FavoriteColors フィールドの Selectivity と Outlier Selectivity を示します。

```
SELECTIVITY of FIELD FavoriteColors
  CURRENT =     1.8966%
  CALCULATED =  1.4405%
  CURRENT OUTLIER = 45.0000%, VALUE = <Null>
  CALCULATED OUTLIER = 39.5000%, VALUE = <Null>
```

フィールドに 1 つの個別値のみが含まれている (すべての行の値が同じ) 場合、そのフィールドの Selectivity は 100% です。Selectivity が 100% の値は、異常値とは見なされません。テーブル・チューニングは、データをサンプリングして Selectivity と Outlier Selectivity の値を設定します。これを判断するため、テーブル・チューニングは最初に少数のレコードをテストします。すべてのレコードのフィールド値が同じである場合、インデックスが付けられていないフィールドの値がすべて同じであるという仮定を裏付けるために、ランダムに選択した最大 100,000 個のレコードをテストします。テーブル・チューニングは、フィールドにインデックスが作成されている場合、そのフィールドがインデックスの最初のフィールドである場合、およびフィールドとインデックスの照合タイプが同じ場合にのみ、フィールドのすべての値が同じであるかどうかを完全に判断できます。

- インデックスが付けられていないフィールドに、ランダムに選択された 100,000 個のレコードのテストで検出できなかった他の値が含まれることがわかっている場合、Selectivity と Outlier Selectivity を手動で設定する必要があります。
- インデックスが付けられていないフィールドに他の値が含まれることがわかっている場合は、手動で 100% の Selectivity を指定して Outlier Selectivity を削除し、CALCSELECTIVITY=0 を設定できます。これにより、テーブル・チューニングが選択性を計算したり、この値を外れ値として指定したりしないようにします。

これらの Selectivity、Outlier Selectivity、および Outlier Value の計算値を変更するには、テーブルのチューニングの表示から個々のフィールドを選択します。これにより、表示の右側の **[詳細]** 領域に、そのフィールドに対するこれらの値が表示されます。Selectivity、Outlier Selectivity、および Outlier Value を、予期される完全なデータ・セットにより適した値に変更することができます。

- Selectivity は、パーセント (%) 記号を含む行のパーセント、または行の整数 (パーセント記号なし) のどちらかで指定できます。行の整数として指定する場合、InterSystems IRIS はエクステント・サイズを使用して Selectivity のパーセントを計算します。
- 以前に異常値のなかったフィールドに Outlier Selectivity および Outlier Value を指定できます。パーセント (%) 記号を持つパーセントとして Outlier Selectivity を指定します。Outlier Selectivity のみを指定した場合、テーブルのチューニングでは Outlier Value が <Null> であると仮定されます。 Outlier Value のみを指定した場合、テーブルのチューニングでは Outlier Selectivity を指定しない限りこの値を保存しません。

1.3.3 CALCSELECTIVITY パラメータと、Selectivity 計算の抑止

特定のプロパティの Selectivity を、テーブルのチューニング機能で計算しないようにすることが必要な場合があります。Selectivity が計算されないようにするには、プロパティの CALCSELECTIVITY パラメータを 0 に設定します (既定値は 1)。CALCSELECTIVITY は、スタジオの **[新規プロパティウィザード]** の **[プロパティパラメータ]** ページで設定できます。また、インスペクタのプロパティ・パラメータのリストでも設定できます (このパラメータを表示するには、プロパティ・パラメータ・リストを一度縮小してから再展開することが必要な場合があります)。

例えば、すべての行に含まれる値が 1 つのみ (Selectivity=100%) だとわかっているフィールドには、CALCSELECTIVITY=0 を指定する必要があります (そのフィールドにインデックスが作成されていない場合)。

1.3.4 [メモ] 列

管理ポータルの **[テーブル・チューニング情報]** オプションでは、フィールドごとに [メモ] 列が表示されます。このフィールドの値はシステムによって定義され、変更することはできません。以下の値があります。

- `RowID field` : テーブルには、システムによって定義される RowID が 1 つあります。通常、その名前は ID ですが、システムによって割り当てられた別の名前を持つ場合もあります。その値はすべて (定義上は) 一意であるため、選択性は常に 1 です。クラス定義に SqlRowIdPrivate が含まれている場合、[メモ] 列の値は `RowID field, Hidden field` です。
- `Hidden field` : 非表示フィールドはプライベートとして定義され、SELECT * によって表示されることはありません。既定では、CREATE TABLE では RowID フィールドは非表示として定義されます。%PUBLICROWID キーワードを指定すると、RowID を非表示でなく、パブリックとして定義できます。既定では、永続クラス定義によって定義さ

れたテーブルでは、RowID は非表示でないとして定義されます。SqlRowIdPrivate を指定すると、RowID を非表示およびプライベートとして定義できます。コンテナ・フィールドは、非表示として定義されます。

- `Stream field`：ストリーム・データ型（文字ストリーム（CLOB）またはバイナリ・ストリーム（BLOB））で定義されたフィールドを示します。ストリーム・ファイルには平均フィールド・サイズはありません。
- `Parent reference field`：親テーブルを参照するフィールドです。

IDENTITY フィールド、ROWVERSION フィールド、SERIAL フィールド、または UNIQUEIDENTIFIER（GUID）フィールドは、[メモ] 列では識別されません。

1.3.5 平均フィールド・サイズ

テーブルのチューニングを実行すると、現在のテーブル・データ・セットに基づいて、すべての非ストリーム・フィールドの平均フィールド・サイズ（文字数）が計算されます。これは、（特に明記されていない限り）小数点以下 2 桁に丸められた `AVG($LENGTH(field))` と同じです。予期されるフィールド・データの平均サイズを反映するよう、個々のフィールドについてこの平均サイズを変更できます。

- NULL：$LENGTH 関数は NULL フィールドを 0 の長さを持つものとして処理するため、NULL フィールドは長さ 0 で平均化されます。これにより、平均フィールド・サイズが 1 文字未満になる場合があります。
- 空の列：列にデータが含まれていない（すべての行のフィールド値がない）場合、平均フィールド・サイズの値は 0 ではなく、1 です。`AVG($LENGTH(field))` は、データが含まれていない列については 0 です。
- ExtentSize=0：ExtentSize を 0 に設定すると、すべてのフィールドの平均フィールド・サイズは 0 にリセットされます。
- 論理フィールド値：平均フィールド・サイズは、常にフィールドの論理（内部）値に基づいて計算されます。
- リスト・フィールド：InterSystems IRIS リスト・フィールドは、論理（内部）エンコード値に基づいて計算されます。このエンコードされた長さは、リスト内の要素の合計の長さより長くなります。
- コンテナ・フィールド：コンテナ・フィールドは、コレクション・オブジェクトの合計の長さより大きくなります。例えば、Sample.Person で、Home コンテナ・フィールドの平均フィールド・サイズは、Home_Street、Home_City、Home_State、および Home_Zip の平均フィールド・サイズの合計より大きくなります。詳細は、″クラスの定義と使用″ の “コレクション・プロパティの SQL プロジェクションの制御” を参照してください。
- ストリーム・フィールド：ストリーム・フィールドには平均フィールド・サイズはありません。

プロパティ・パラメータ CALCSELECTIVITY がプロパティ/フィールドで 0 に設定されると、テーブルのチューニングでそのプロパティ/フィールドの平均フィールド・サイズが計算されません。

テーブルのチューニングの表示から個々のフィールドを選択することで、平均フィールド・サイズの計算値を変更できます。これにより、表示の右側の **[詳細]** 領域に、そのフィールドの値が表示されます。平均フィールド・サイズは、予期される完全なデータ・セットにより適した値に変更できます。この値の設定時にテーブルのチューニングで検証が実行されないため、フィールドがストリーム・フィールドではないこと、および指定した値が最大フィールド・サイズを超えない（MaxLen）ことを確認する必要があります。

平均フィールド・サイズは、管理ポータルの **[カタログの詳細]** タブの **[フィールド]** オプション・テーブルにも表示されます。平均フィールド・サイズの値を表示するには、**[フィールド]** オプション・テーブルでテーブルのチューニングが実行されている必要があります。詳細は、このドキュメントの “管理ポータルの SQL インタフェースの使用法” の章の ″[カタログの詳細] タブ″ のセクションを参照してください。

1.3.6 [BlockCount のマップ] タブ

テーブルのチューニングの **[BlockCount のマップ]** タブには、**[SQL マップ名]**、**[BlockCount]**（正の整数として）、および **[BlockCount のソース]** が表示されます。**[BlockCount のソース]** は、`Defined in class definition`、`Estimated by class compiler`、または `Measured by TuneTable` のいずれかになります。テーブルのチューニングを実行

すると、Estimated by class compiler が Measured by TuneTable に変わります。Defined in class definition の値には影響しません。

テーブルのチューニングの表示から個々の SQL マップ名フィールドを選択することで、BlockCount の計算値を変更できます。これにより、表示の右側の [詳細] 領域にあるフィールドに値が表示されます。BlockCount は、予期される完全なデータ・セットにより適した値に変更できます。この値の設定時にテーブルのチューニングで検証が実行されないため、ブロックカウントが有効な値であることを確認する必要があります。BlockCount を変更すると、[BlockCount のソース] が Defined in class definition に変更されます。詳細は、このドキュメントの "BlockCount" のセクションを参照してください。

1.4 テーブルのチューニングの統計のエクスポートと再インポート

テーブル・チューニング統計をテーブルまたはテーブルのグループからエクスポートし、それらのテーブル・チューニング統計をテーブルまたはテーブルのグループにインポートできます。以下に示す 3 つの状況で、このエクスポート/インポートの実行が必要になることがあります(わかりやすくするために、1 つのテーブルの統計のエクスポート/インポートについて説明します。実際には、多くの場合、相互に関連する複数のテーブルの統計のエクスポート/インポートが実行されます)。

- プロダクション・システムをモデル化する場合：実際のデータを使用してプロダクション・テーブルを完全に生成し、テーブル・チューニングを使用して最適化します。テスト環境で、同じテーブル定義でテーブルを作成しますが、はるかに少ないデータを使用します。テーブル・チューニング統計をプロダクション・テーブルからエクスポートし、テスト・テーブルにインポートすることにより、テスト・テーブルでプロダクション・テーブルの最適化をモデル化できます。
- プロダクション・システムを複製する場合：実際のデータを使用してプロダクション・テーブルを完全に生成し、テーブル・チューニングを使用して最適化します。同じテーブル定義の 2 つ目のプロダクション・テーブルを作成します(例えば、プロダクション環境とそのバックアップ環境や、各テーブルに異なる病院の患者レコードが格納されている複数の同じテーブル定義など)。テーブル・チューニング統計を最初のテーブルからエクスポートし、2 つ目のテーブルにインポートすることにより、テーブル・チューニングをもう一度実行したり、2 つ目のテーブルに代表的なデータが移入されるまで待つというオーバーヘッドを伴わずに、最初のテーブルと同じ最適化を 2 つ目のテーブルに適用できます。
- 以前の統計セットに戻す場合：テーブル・チューニングを実行したり、明示的に統計を設定したりすることにより、テーブルの最適化統計を作成します。これらの統計をエクスポートすることにより、それらを保持したまま、他の統計設定を試すことができます。最適な統計セットを特定したら、それらをテーブルにインポートして戻すことができます。

テーブル・チューニングの統計は、$SYSTEM.SQL.Stats.Table.Export() メソッドを使用して、XML ファイルにエクスポートできます。このメソッドでは、以下の例に示すように、1 つのネームスペース内の 1 つのテーブル、複数のテーブル、またはすべてのテーブルについて、テーブルのチューニングの統計をエクスポートできます。

ObjectScript

```
DO $SYSTEM.SQL.Stats.Table.Export("C:\AllStats.xml")
/*  Exports TuneTable Statistics for all schemas/tables in the current namespace */
```

ObjectScript

```
DO $SYSTEM.SQL.Stats.Table.Export("C:\SampleStats.xml","Sample")
/*  Exports TuneTable Statistics for all tables in the Sample schema */
```

ObjectScript

```
DO $SYSTEM.SQL.Stats.Table.Export("C:\SamplePStats.xml","Sample","P*")
/*  Exports TuneTable Statistics for all tables beginning with the letter "P" in the Sample schema
*/
```

ObjectScript

```
DO $SYSTEM.SQL.Stats.Table.Export("C:\SamplePersonStats.xml","Sample","Person")
/*  Exports TuneTable Statistics for the Sample.Person table */
```

$SYSTEM.SQL.Stats.Table.Export() を使用してエクスポートされたテーブル・チューニングの統計は、$SYSTEM.SQL.Stats.Table.Import() メソッドを使用して再インポートできます。

$SYSTEM.SQL.Stats.Table.Import() には、ClearCurrentStats ブーリアン・オプションがあります。TRUE の場合、$SYSTEM.SQL.Stats.Table.Import() は、統計をインポートする前に既存のテーブルから以前の EXTENTSIZE、SELECTIVITY、BLOCKCOUNT、および他のテーブル・チューニング統計を消去します。これは、インポート・ファイルで指定されていないテーブル統計を、テーブルの永続クラスで定義されたままにするのではなく完全に消去する場合に使用できます。既定値は FALSE (0) です。

$SYSTEM.SQL.Stats.Table.Import() は、対応するテーブルが見つからないときには、そのテーブルをスキップし、インポート・ファイル内でその次に指定されているテーブルに処理を進めます。テーブルが見つかったものの、一部のフィールドが見つからないときには、それらのフィールドが単にスキップされます。

クラス・ストレージ定義内のマップの BlockCount は継承できません。BlockCount は、マップが生成されたクラスのストレージ定義内にのみ現れます。$SYSTEM.SQL.Stats.Table.Import() では、投影されたテーブルの BlockCount メタデータのみが設定され、クラス・ストレージの BlockCount メタデータは設定されません (マップがスーパークラス内で生成された場合)。

2
インデックスの定義と作成

2.1 概要

インデックスとは、InterSystems IRIS® データ・プラットフォームがクエリや他の操作の最適化に使用できる、永続クラスにより管理される構造のことです。

インデックスは、テーブル内のフィールドの値に対して定義することも、クラス内の対応するプロパティに対して定義することもできます(また、インデックスは複数のフィールド/プロパティを組み合わせた値に対して定義することもできます)。SQL フィールドとテーブルの構文、またはクラス・プロパティ構文を使用して定義されているかどうかに関係なく、同じインデックスが作成されます。特定のタイプのフィールド (プロパティ) を定義すると、InterSystems IRIS によってインデックスが自動的に定義されます。データを格納したフィールドや、データを確実に派生できるフィールドに、追加のインデックスを定義できます。InterSystems IRIS には、さまざまなタイプのインデックスが用意されています。同じフィールド (プロパティ) に複数のインデックスを定義して、目的が異なるさまざまなタイプのインデックスを用意できます。

InterSystems IRIS では、データベースに対してデータの挿入、更新、または削除の操作を実行すると、SQL フィールドとテーブルの構文またはクラス・プロパティ構文を使用しているかどうかに関係なく、既定ではインデックスが生成され、維持されます。この既定はデータへの変更を迅速に実行するためにオーバーライドできます (%NOINDEX キーワードを使用)。その後で、個別の操作として対応するインデックスをビルドまたはリビルドします。テーブルにデータを移入する前にインデックスを定義できます。さらに、すでにデータが移入されているテーブルにインデックスを定義して、それから別の操作としてインデックスを構築することもできます。

InterSystems IRIS では、SQL クエリを作成および実行するときに、利用可能なインデックスが活用されます。既定では、クエリのパフォーマンスが最適化されるように、使用するインデックスが選択されます。必要に応じて、特定のクエリまたはすべてのクエリに 1 つ以上のインデックスが使用されないようにするために、この既定をオーバーライドできます。インデックスの最適な使用方法の詳細は、"“インデックスの作成と定義”" ページの "インデックスの使用" のセクションを参照してください。

2.1.1 インデックスの属性

すべてのインデックスに一意の名前があります。この名前は、レポート作成、インデックス構築、インデックス削除をはじめとするデータベース管理に使用します。他の SQL エンティティと同様に、インデックスにも SQL インデックス名および対応するインデックス・プロパティ名の両方があります。これらの名前は、許可されている文字、大文字と小文字の区別、および最大長が異なります。SQL CREATE INDEX コマンドを使用して定義する場合、システムによって対応するインデックス・プロパティ名が生成されます。永続クラス定義を使用して定義する場合、SqlName キーワードにより、ユーザは異なる SQL インデックス名 (SQL マップ名) を指定することができます。管理ポータルの SQL インタフェース [カタログの詳細] には、各インデックスの SQL インデックス名 ([**SQL マップ名**]) および対応するインデックス・プロパティ名 ([**インデックス名**]) が表示されます。

インデックス・タイプは 2 つのインデックス・クラス・キーワード、Type および Extent により定義されます。InterSystems IRIS で使用可能なインデックスのタイプは以下のとおりです。

- 標準インデックス (Type = index) ー インデックス値と、値を収めた行の RowID を関連付ける永続配列です。ビットマップ・インデックス、ビットスライス・インデックス、またはエクステント・インデックスとして明示的に定義されていないインデックスは、すべて標準インデックスです。
- ビットマップ・インデックス (Type = bitmap) ー 指定されたインデックス値に対応する一連の RowID 値を表すために一連のビット文字列を使用する特殊な種類のインデックスです。InterSystems IRIS には、ビットマップ・インデックス用のパフォーマンス最適化機能が数多く用意されています。
- ビットスライス・インデックス (Type = bitslice) ー 合計や値域条件など、特定の式についてきわめて高速な評価を可能にする特殊な種類のインデックスです。ビットスライス・インデックスは、特定の SQL クエリで自動的に使用されます。
- 列指向インデックス (Type = columnar) ー 基盤となるデータが複数の行にわたって格納されている列に対してきわめて高速なクエリを実行できる特殊な種類のインデックスです。特に、フィルタリングと集約の処理を伴うクエリで効果的です。列指向インデックスは、2022.2 の試験的機能です。
- エクステント・インデックス ー エクステント内のすべてのオブジェクトのインデックスです。詳細は、"クラス定義リファレンス" の "Extent" インデックス・キーワードのページを参照してください。

1 つのテーブル (クラス) で使用できるインデックスの最大数は 400 です。

2.1.2 ストレージ・タイプとインデックス

ここで説明するインデックス機能は、永続クラスに保存されたデータに適用されます。InterSystems SQL は、InterSystems IRIS で既定のストレージ構造である %Storage.Persistent (%Storage.Persistent にマップしたクラス) を使用して保存したデータと、%Storage.SQL (%Storage.SQL にマップしたクラス) を使用して保存したデータに対応するインデックス機能をサポートしています。%Storage.SQL にマップしたクラスのインデックスは、機能インデックス・タイプを使用することで定義できます。このインデックスは、既定のストレージを使用するクラスのインデックスと同じ方法で定義しますが、以下の特別な考慮事項が付随します。

- このクラスでは、IdKey 機能インデックスを定義する必要があります (システムによる自動的な割り当てがない場合)。後述の "マスタ・マップ" を参照してください。
- この機能インデックスは、INDEX として定義する必要があります。

詳細は %Library.FunctionalIndex を参照してください。

2.1.3 インデックスのグローバル名

インデックス・データの保存先となるグローバルの添え字付き名前は、該当のデータをテーブルに保存するグローバルの名前で決まります。これらの名前は、テーブルを定義する USEEXTENTSET クラス・パラメータと DEFAULTGLOBAL クラス・パラメータの値に依存して、永続クラスで決まるか、CREATE TABLE 文に %CLASSPARAMETER キーワードを使用することで決まります。USEEXTENTSET と DEFAULTGLOBAL の詳細は、それぞれ "ハッシュ定義のグローバル名" と "ユーザ定義のグローバル名" を参照してください。

2.1.4 マスタ・マップ

すべてのテーブルにマスタ・マップが自動的に定義されます。マスタ・マップは、インデックスではありません。マップの添え字フィールドを使用して、データ自体に直接アクセスするマップです。既定では、マスタ・マップの添え字フィールドは、システム定義の RowID フィールドです。既定では、RowID フィールドを使用したこの直接データ・アクセスは、SQL マップ名 (SQL インデックス名) IDKEY で表されます。

既定では、ユーザ定義の主キーは IDKEY ではありません。ほとんどの場合、RowID の整数を使用したマスタ・マップ検索の方が主キー値による検索よりも効率的であるためです。ただし、主キーを IDKEY として指定した場合は、主キー・インデックスがテーブルのマスタ・マップとして定義され、SQL マップ名は主キー SQL インデックス名になります。

単一フィールドの主キー/IDKEY の場合、主キー・インデックスがマスタ・マップになりますが、マスタ・マップのデータ・アクセス列は RowID のままです。これは、レコードの一意の主キー・フィールド値とその RowID 値の間には 1 対 1 の一致があり、また、RowID の方が効率的な検索であると考えられるためです。複数フィールドの主キー/IDKEYの場合、マスタ・マップに主キー・インデックス名が指定され、マスタ・マップのデータ・アクセス列は主キー・フィールドになります。

管理ポータルの SQL の [カタログの詳細] タブを使用して、マスタ・マップ定義を表示できます。このタブには、マスタ・マップ・データが格納されるグローバル名などの項目が表示されます。SQL および既定のストレージの場合、このマスタ・マップ・グローバルは既定で ^package.classnameD に設定され、あいまいさを回避するために、ネームスペースが記録されます。カスタム・ストレージの場合は、マスタ・マップのデータ・ストレージ・グローバルは定義されません。DATALOCATIONGLOBAL クラス・パラメータを使用して、データ・ストレージ・グローバル名を指定できます。

SQL および既定のストレージの場合、添え字付き名前を持つグローバルである `^package.classnameD` または `^hashpackage.hashclass.1` にマスタ・マップ・データが格納されます (詳細は、"インデックスのグローバル名" を参照してください)。グローバル名には、対応する SQL テーブル名ではなく、永続クラス名が指定されます。また、グローバル名では大文字と小文字が区別されます。グローバル名を ZWRITE に渡すことで、マスタ・マップ・データを表示できます。

マスタ・マップを使用したデータ・アクセスは、テーブルが大きい場合は特に、非効率的です。このことから、WHERE 条件や JOIN 操作などの操作で指定したデータ・フィールドへのアクセスで使用できるインデックスをユーザ側で定義しておくことをお勧めします。

2.2 自動定義されたインデックス

テーブルを定義するときに、特定のインデックスが自動的に定義されます。テーブルを定義するときに以下のインデックスが自動的に生成され、テーブルのデータを追加または変更するときに、これらのインデックスに値が入力されます。以下の定義について説明します。

- IDKEY でない主キーを定義すると、タイプ Unique の対応するインデックスが生成されます。主キー・インデックスの名前は、ユーザが指定した名前にすることも、テーブルの名前から導出することもできます。例えば、名前のない主キーを定義した場合、対応するインデックスの名前は tablenamePKEY# になります (# は、それぞれの一意の主キー制約の連続する整数です)。
- UNIQUE フィールドを定義すると、tablenameUNIQUE# という名前で UNIQUE フィールドごとにインデックスが生成されます (# は、それぞれの一意の主キー制約の連続する整数です)。
- UNIQUE 制約を定義すると、指定された名前で UNIQUE 制約ごとにインデックスが生成され、一意の値を共に定義するフィールドにインデックスが付けられます。
- シャード・キーを定義すると、ShardKey という名前のシャード・キー・フィールドにインデックスが生成されます。

管理ポータルの SQL の [カタログの詳細] タブを使用して、これらのインデックスを表示できます。CREATE INDEX コマンドを使用すると、UNIQUE フィールド制約を追加できます。DROP INDEX コマンドを使用すると、UNIQUE フィールド制約を削除できます。

既定では、RowID フィールドに IDKEY インデックスが生成されます。IDENTITY フィールドを定義しても、インデックスは生成されません。ただし、IDENTITY フィールドを定義して、そのフィールドを主キーにすると、InterSystems IRIS は IDENTITY フィールドに IDKEY インデックスを定義して主キー・インデックスにします。以下に例を示します。

SQL

```
CREATE TABLE Sample.MyStudents (
           FirstName VARCHAR(12),
           LastName VARCHAR(12),
           StudentID IDENTITY,
           CONSTRAINT StudentPK PRIMARY KEY (StudentID) )
```

同様に、IDENTITY フィールドを定義して、そのフィールドに UNIQUE 制約を適用すると、InterSystems IRIS は IDENTITY フィールドに明示的に IdKey/Unique インデックスを定義します。以下に例を示します。

SQL

```
CREATE TABLE Sample.MyStudents (
           FirstName VARCHAR(12),
           LastName VARCHAR(12),
           StudentID IDENTITY,
           CONSTRAINT StudentU UNIQUE (StudentID) )
```

こうした IDENTITY インデックスの作成処理は、明示的に定義された IdKey インデックスが存在しないことと、テーブルにデータが格納されていない場合にのみ実行されます。

2.2.1 ビットマップ・エクステント・インデックス

ビットマップ・エクステント・インデックスは、テーブルの特定のフィールドではなく、テーブルの行のビットマップ・インデックスです。ビットマップ・エクステント・インデックスでは、各ビットは連続した RowID の整数値を表し、各ビットの値は対応する行が存在しているかどうかを示します。InterSystems SQL はこのインデックスを使用して、テーブル内のレコード (行) 数を返す COUNT(*) のパフォーマンスを改善します。テーブルには最大 1 つのビットマップ・エクステント・インデックスを指定できます。複数のビットマップ・エクステント・インデックスを作成しようとすると、SQLCODE -400 エラーが発生し、メッセージ %msg `ERROR #5445:  : DDLBEIndex` が表示されます。

- CREATE TABLE を使用して定義されたテーブルでは、自動的にビットマップ・エクステント・インデックスが定義されます。この自動的に生成されたインデックスには、インデックス名 DDLBEIndex および SQL マップ名 %%DDLBEIndex が割り当てられます。
- 永続クラスとして定義されたテーブルでは、自動的にビットマップ・エクステント・インデックスが定義されません。ビットマップを持たない永続クラスにビットマップ・インデックスを追加すると、InterSystems IRIS は自動的にビットマップ・エクステント・インデックスを生成します。この生成されたビットマップ・エクステント・インデックスには、$ClassName のインデックス名と SQL マップ名が割り当てられています (ClassName は、テーブルの永続クラスの名前です)。

BITMAPEXTENT キーワードを指定した CREATE INDEX コマンドを使用して、テーブルにビットマップ・エクステント・インデックスを追加したり、自動的に生成されたビットマップ・エクステント・インデックスの名前を変更したりできます。詳細は、"CREATE INDEX" を参照してください。

管理ポータルの SQL の [カタログの詳細] タブを使用して、テーブルのビットマップ・エクステント・インデックスを表示できます。テーブルに指定できるビットマップ・エクステント・インデックスは 1 つのみですが、別のテーブルを継承したテーブルは、それ自身のビットマップ・エクステント・インデックスと継承元のテーブルのビットマップ・エクステント・インデックスの両方と共にリストされます。例えば、Sample.Employee テーブルが Sample.Person テーブルを継承した場合、**[カタログの詳細]** の [マップ/インデックス] では、Sample.Employee に `$Employee` と `$Person` の両方のビットマップ・エクステント・インデックスがリストされます。

多数の DELETE 操作が実行されるテーブルでは、ビットマップ・エクステント・インデックス用のストレージは徐々に効率が低下する可能性があります。BUILD INDEX コマンドを使用するか、管理ポータルでテーブルの **[カタログの詳細]** タブの [マップ/インデックス] オプションを選択し、**[インデックス再構築]** を選択して、ビットマップ・エクステント・インデックスを再構築することができます。

%SYS.Maint.Bitmap ユーティリティ・メソッドでは、ビットマップ・インデックスおよびビットスライス・インデックスと同様にビットマップ・エクステント・インデックスが圧縮されます。詳細は、"ビットマップ・インデックスの維持" を参照してください。

%BuildIndices() メソッドを呼び出したときに、%BuildIndices() の pIndexList 引数が指定されていない (すべての定義済みインデックスを構築する) 場合、pIndexList にビットマップ・エクステント・インデックスが名前で指定されている場合、または pIndexList に定義済みのビットマップ・インデックスが指定されている場合、このメソッドは既存のビットマップ・エクステント・インデックスを構築します。“プログラミングによるインデックスの構築”を参照してください。

2.3 インデックスの定義

インデックスを定義するには、以下の 2 つの方法があります。

- DDL を使用したインデックスの定義
- クラス定義を使用したインデックスの定義。内容は以下のとおりです。
  - インデックスを付けることができるプロパティ
  - プロパティの組み合わせに対するインデックス
  - インデックス照合
  - インデックスでの Unique、PrimaryKey、IdKey キーワードの使用
  - インデックスを使用したデータの保存
  - NULL のインデックス付け
  - コレクションのインデックス作成
  - 配列コレクションのインデックス作成
  - (ELEMENTS) および (KEYS) を使用したデータ型プロパティのインデックス作成
  - 埋め込みオブジェクト (%SerialObject) のプロパティのインデックス作成

2.3.1 DDL を使用したインデックスの定義

テーブルの定義に DDL 文を使用している場合、インデックスの生成および削除に以下の DDL コマンドを使用できます。

- CREATE INDEX
- DROP INDEX

DDL インデックス・コマンドは以下を実行します。

1. インデックスが追加または削除される、対応するクラス定義およびテーブル定義を更新します。変更されたクラス定義は、再コンパイルされます。
2. 必要に応じて、データベースのインデックス・データを追加または削除します。CREATE INDEX コマンドは、現在データベースに保存されているデータを使用してインデックスを生成します。同様に、DROP INDEX コマンドは、データベースからインデックス・データ (実際のインデックス) を削除します。

2.3.2 クラス定義を使用したインデックスの定義

クラス定義のテキストを編集することで、%Persistent クラス定義にインデックス定義を追加できます。インデックスは、1 つ以上のインデックス・プロパティ式で定義され、必要に応じて 1 つ以上のオプションのインデックス・キーワードが続きます。以下の形式をとります。

```
INDEX index_name ON index_property_expression_list [index_keyword_list];
```

以下はその説明です。

- index_name は、有効な識別子です。
- index_property_expression_list は、インデックスの基準となる 1 つ以上のコンマ区切りのプロパティ式のリストです。
- index_keyword_list は、角括弧で囲まれた、インデックス・キーワードのコンマ区切りリストです (オプション)。ビットマップ・インデックスまたはビットスライス・インデックスに、インデックスの Type を指定するために使用します。また、Unique インデックス、IdKey インデックス、または PrimaryKey インデックスを指定する場合にも使用します。(IdKey インデックスまたは PrimaryKey インデックスは、定義上、Unique インデックスでもあります)。インデックス・キーワードの完全なリストについては、"クラス定義リファレンス" を参照してください。

index_property_expression_list 引数は、1 つ以上のインデックス・プロパティ式で構成されています。インデックス・プロパティ式の構成要素は以下のとおりです。

- インデックスの作成対象のプロパティの名前。
- オプションの (ELEMENTS) 式または (KEYS) 式。これは、コレクションのサブ値に対するインデックス作成手段となります。インデックス・プロパティがコレクションでない場合、ユーザは BuildValueArray() メソッドを使用して、キーと要素を格納する配列を生成できます。キーと要素の詳細は、"コレクションのインデックス作成" のセクションを参照してください。
- オプションの照合式。この式は、照合名と、必要に応じてそれに続く 1 つ以上のコンマ区切りの照合パラメータのリストで構成されます。インデックス照合は、Unique インデックス、IdKey インデックス、または PrimaryKey インデックスには指定できません。Unique または PrimaryKey インデックスの照合は、インデックスを生成するプロパティ (フィールド) から取得します。IdKey インデックスの照合は、常に EXACT になります。有効な照合名のリストについては、"InterSystems SQL の使用法" の "照合" の章にある "照合タイプ" のセクションを参照してください。

注釈　CREATE INDEX コマンドを使用する場合とは異なり、インデックスをクラス定義に追加しても、コンパイル時にインデックスは自動的に構築されません。インデックスの構築の詳細は、"インデックスの構築" を参照してください。

例えば、以下のクラス定義は、2 つのプロパティとそれぞれのプロパティに基づくインデックスを定義しています。

Class Definition

```
Class MyApp.Student Extends %Persistent [DdlAllowed]
{
 Property Name As %String;
 Property GPA As %Decimal;

 Index NameIDX On Name;
 Index GPAIDX On GPA;
}
```

より複雑なインデックス定義は以下のようになります。

Class Member

```
 Index Index1 On (Property1 As SQLUPPER(77), Property2 AS EXACT);
```

2.3.2.1 インデックスを付けることができるプロパティ

以下の各プロパティにインデックスを作成できます。

- データベースに格納されているプロパティ。
- 格納されているプロパティから確実に派生できるプロパティ。

確実に派生できるものの、格納されていないプロパティでは、SQLComputed キーワードに True を定義する必要があります。プロパティ値の派生ではプロパティの計算コードを唯一の方法とする必要があります。プロパティを直接設定することはできません。

一般的に、インデックスを付けることができるのは、Calculated および SQLComputed として定義されている派生プロパティのみです。ただし、派生コレクションには例外があり、Calculated キーワードではなく、SQLComputed および Transient キーワードで定義されたコレクションにはインデックスを付けることができます。

テキストの長さに制限があるプロパティでは、データを取り込むことはできても、プロパティの内容が、インデックスの添え字にするには長すぎることがあります。複数のフィールドを結合することで添え字には長すぎるプロパティになる複合インデックスも、テキスト長さ制限の対象となることがあります。しかし、一般的に、長い文字列に対するインデックスは、それによって他の値が特定されることはまれなので、インデックスとしての有用性は限られています。したがって、この場合はエラー・メッセージが表示され、インデックスにデータは保存されません。それでも、長いテキスト・フィールドをインデックスにする場合は、該当のプロパティの照合文字列を 128 文字に制限した TRUNCATE 照合を定義できます。

注釈　IdKey インデックスによって使用されるどのプロパティの値においても、そのプロパティが永続クラスのインスタンスへの有効な参照でない限り、連続する 2 つの垂直バー (||) は使用しないでください。この制限は、InterSystems SQL の内部メカニズムで必要とされています。IdKey プロパティで || を使用すると、予測できない動作を起こす場合があります。

2.3.2.2 プロパティの組み合わせに対するインデックス

2 つ以上のプロパティ (フィールド) の結合にインデックスを作成できます。以下のように、クラス定義の中のインデックス定義の On 節を使用して、プロパティのリストを指定します。

Class Definition

```
Class MyApp.Employee Extends %Persistent [DdlAllowed]
{
 Property Name As %String;
 Property Salary As %Integer;
 Property State As %String(MAXLEN=2);

 Index MainIDX On(State,Salary);
}
```

複数のプロパティに対するインデックスは、以下のように、フィールド値の組み合わせを使用したクエリを実行する場合に便利です。

SQL

```
SELECT Name,State,Salary
  FROM Employee
  ORDER BY State,Salary
```

2.3.2.3 インデックス照合

Unique インデックス、PrimaryKey インデックス、または IdKey インデックスには、照合タイプを指定できません。その他のタイプのインデックスの場合、オプションで、インデックス定義で指定した各プロパティに照合タイプを指定できます。インデックスが適用される場合、インデックスの照合タイプは、プロパティ (フィールド) の照合タイプと一致している必要があります。

1. インデックス定義に、明示的に指定されたプロパティの照合が含まれている場合、インデックスはその照合を使用します。
2. インデックス定義に、明示的に指定されたプロパティの照合が含まれていない場合、インデックスはプロパティ定義で明示的に指定された照合を使用します。
3. プロパティ定義に、明示的に指定された照合が含まれていない場合、インデックスはプロパティのデータ型の既定の照合を使用します。

例えば、Name プロパティが文字列として定義されていると、そのプロパティの既定は SQLUPPER 照合になります。Name に対するインデックスを定義すると、そのプロパティの照合が既定で取得され、そのインデックスも SQLUPPER を使用して定義されます。プロパティの照合とインデックスの照合が一致します。

ただし、異なる照合を適用する比較の場合（例えば、`WHERE %EXACT(Name)=%EXACT(:invar)`）、この使用法でのプロパティの照合タイプは、インデックスの照合タイプと一致しなくなります。プロパティ比較の照合タイプとインデックスの照合タイプの間に不一致があると、インデックスが使用されなくなる可能性があります。そのため、この場合は、Name プロパティのインデックスを照合 EXACT で定義することをお勧めします。JOIN 文の ON 節で照合タイプを指定すると（例えば、`FROM Table1 LEFT JOIN Table2 ON %EXACT(Table1.Name) = %EXACT(Table2.Name)`）、ここで指定したプロパティの照合タイプとインデックスの照合タイプとの間の不一致が原因になり、InterSystems IRIS がインデックスを使用しなくなる可能性があります。

インデックスとプロパティとの間での照合の一致を制御するルールは以下のとおりです。

- 照合タイプが一致する場合は、常にインデックスが最大限使用されます。
- 照合タイプが不一致で、プロパティが EXACT 照合で指定されており（前述の説明を参照）、インデックスが別の照合を持っている場合は、インデックスが使用できるようにはなりますが、インデックス使用の効果は照合タイプが一致しているときよりも低下します。
- 照合タイプが不一致で、プロパティの照合が EXACT ではなく、プロパティの照合がインデックスの照合と一致しない場合は、インデックスが使用されなくなります。

インデックス定義でプロパティの照合を明示的に指定するには、以下の構文を使用します。

```
Index IndexName On PropertyName As CollationName;
```

以下はその説明です。

- IndexName は、インデックスの名前です。
- PropertyName はインデックスが付けられているプロパティです。
- CollationName はインデックスに使用されている照合タイプです。

以下はその例です。

Class Member

```
Index NameIDX On Name As Exact;
```

異なるプロパティは、異なる照合タイプを持つことができます。例えば、以下の例では、**F1** プロパティは SQLUPPER 照合を使用する一方で、**F2** は EXACT 照合を使用します。

Class Member

```
Index Index1 On (F1 As SQLUPPER, F2 As EXACT);
```

推奨される照合タイプのリストについては、″InterSystems SQL の使用法″ の “照合” の章にある “照合タイプ” のセクションを参照してください。

注釈　Unique、PrimaryKey、または IdKey として指定されたインデックスには、インデックス照合を指定できません。このインデックスの照合は、プロパティの照合から取得されます。

2.3.2.4 インデックスでの Unique、PrimaryKey、IdKey キーワードの使用

SQL では普通のことですが、InterSystems IRIS では一意キーと主キーの概念をサポートしています。また、InterSystems IRIS には IdKey を定義する機能もあります。IdKey とは、クラスのインスタンス (テーブルの行) に対して一意のレコード ID となるものです。これらの機能は、以下に示す Unique、PrimaryKey、IdKey の各キーワードを使用して実装されます。

- Unique ー インデックスのプロパティ・リストに含まれているプロパティに対する UNIQUE 制約を定義します。そのため、このプロパティ (フィールド) で一意のデータ値のみがインデックス化できるようになります。一意性は、プロパティの照合に基づいて判断されます。例えば、プロパティの照合が EXACT の場合、大文字/小文字の異なる値は一意になります。プロパティの照合が SQLUPPER の場合は、大文字/小文字が異なる値は一意にはなりません。ただし、未定義のプロパティに対してインデックスの一意性はチェックされないことに注意してください。SQL 標準に従い、未定義のプロパティは常に一意として扱われます。
- PrimaryKey ー インデックスのプロパティ・リストに含まれているプロパティに対する PRIMARY KEY 制約を定義します。
- IdKey ー 一意の制約を定義し、インスタンス (行) の一意の ID を定義するために使用するプロパティを指定します。IdKey は、常に EXACT 照合を持ちます (データ型が文字列でも同じ照合になります)。

このようなキーワードの構文を表示すると、以下の例のようになります。

Class Definition

```
Class MyApp.SampleTable Extends %Persistent [DdlAllowed]
{
  Property Prop1 As %String;
  Property Prop2 As %String;
  Property Prop3 As %String;

  Index Prop1IDX on Prop1 [ Unique ];
  Index Prop2IDX on Prop2 [ PrimaryKey ];
  Index Prop3IDX on Prop3 [ IdKey ];
}
```

注釈　IdKey、PrimaryKey、Unique の各キーワードは、標準インデックスで使用する場合にのみ有効です。ビットマップ・インデックスやビットスライス・インデックスで使用することはできません。

また、以下のように、IdKey と PrimaryKey の両方のキーワードを組み合わせて指定する構文も有効です。

Class Member

```
Index IDPKIDX on Prop4 [ IdKey, PrimaryKey ];
```

この構文では、IDPKIDX インデックスがクラス (テーブル) の IdKey であり、かつその主キーであることを指定します。これらのキーワードを上記以外の形で組み合わせた場合は、すべて冗長になります。

インデックスがこれらのキーワードのいずれかを使用して定義されていると、そのインデックスに関連付けられたプロパティに特定の値が指定されているクラスのインスタンスを開くことができるメソッドがあります。詳細は、“インデックス・キーによるインスタンスのオープン” のセクションを参照してください。

IdKey キーワードの詳細は、″クラス定義リファレンス″ の ″IdKey″ のページを参照してください。PrimaryKey キーワードの詳細は、″クラス定義リファレンス″ の ″PrimaryKey″ のページを参照してください。Unique キーワードの詳細は、″クラス定義リファレンス″ の ″Unique″ のページを参照してください。

2.3.2.5 インデックスを使用したデータの保存

インデックスの Data キーワードを使用して、1 つ以上のデータ値のコピーをインデックスに保存するように指定できます。

Class Definition

```
Class Sample.Person Extends %Persistent [DdlAllowed]
{
 Property Name As %String;
 Property SSN As %String(MAXLEN=20);

 Index NameIDX On Name [Data = Name];
}
```

この場合、インデックス NameIDX には、さまざまな Name 値の照合値 (大文字) によって添え字が付けられます。Name の実際の (照合されていない) 値のコピーは、インデックスに保存されています。これらのコピーは、SQL を使用して Sample.Person テーブルが変更された場合や、オブジェクトを使用して対応する Sample.Person クラスまたはそのインスタンスが変更された場合も維持されます。

選択 (多数の行から数行を選ぶ) や抽出 (多数の列から数列を返す) を頻繁に実行する場合、インデックスにデータのコピーを維持する機能は大変便利です。

例えば、Sample.Person テーブルに対する以下のクエリがあります。

SQL

```
SELECT Name FROM Sample.Person ORDER BY Name
```

SQL エンジンは、テーブルのマスタ・データを読まずに、NameIDX を読むだけですべての要求を満たすことができます。

注釈　　ビットマップ・インデックスでは、値を保存できません。

2.3.2.6 NULL のインデックス付け

インデックスが付けられたフィールドのデータに NULL (データが存在しない) が含まれている場合、対応するインデックスでは、このフィールドがインデックス NULL 標識を使用して表されます。既定では、インデックス NULL 標識の値は -1E14 です。インデックス NULL 標識を使用することで、NULL 以外のすべての値の前に NULL 値を照合するようになります。INDEXNULLMARKER プロパティ・パラメータを使用することで、特定のフィールドについてインデックス NULL 標識の値を変更できます。

2.3.2.7 コレクションのインデックス作成

プロパティにインデックスが付けられる場合、インデックスに配置される値は照合プロパティ値全体です。コレクションの場合は、プロパティ名に (ELEMENTS) または (KEYS) を追加することで、コレクションの要素値とキー値に対応するインデックス・プロパティを定義することができます。(ELEMENTS) と (KEYS) を使用すると、1 つのプロパティ値から複数の値を生成して、それらの各サブ値にインデックスを作成するように指定できます。プロパティがコレクションの場合、ELEMENTS トークンはコレクションの要素を値で参照し、KEYS トークンは同様の要素を位置で参照します。1 つのインデックス定義に ELEMENTS と KEYS の両方が存在する場合、インデックスのキー値には、キーとそれに関連付けられた要素の値が保存されます。

例えば、Sample.Person クラスの FavoriteColors プロパティに基づくインデックスがあるとします。このプロパティのコレクション内の項目のインデックスのうち、最も単純な形式は以下のいずれかになります。

Class Member

```
 INDEX fcIDX1 ON (FavoriteColors(ELEMENTS));
```

または

Class Member

```
 INDEX fcIDX2 ON (FavoriteColors(KEYS));
```

FavoriteColors(ELEMENTS) は FavoriteColors プロパティの要素、FavoriteColors(KEYS) は FavoriteColors プロパティのキーです。一般的な形式は、propertyName(ELEMENTS) または propertyName(KEYS) です。ここで、そのコレクションの内容は、いくつかのデータ型の List Of または Array Of として定義されたプロパティに含まれる一連の要素です。コレクションについては、"クラスの定義と使用" の "コレクションを使用した作業" の章を参照してください。

リテラル・プロパティ ("クラスの定義と使用" の "リテラル・プロパティの定義と使用" の章を参照) にインデックスを作成するには、propertyNameBuildValueArray() メソッドでインデックス値配列を作成します (以下のセクションを参照)。固有のコレクションと同様、(ELEMENTS) 構文および (KEYS) 構文は、インデックス値配列と併用可能です。

プロパティ・コレクションが配列として投影される場合、インデックスがコレクション・テーブルに投影されるためには以下の制限に従う必要があります。インデックスには (KEYS) を含める必要があります。インデックスは、コレクション自体とオブジェクトの ID 値以外のプロパティを参照することはできません。投影されるインデックスが、インデックスに格納される DATA も定義する場合、格納されるデータ・プロパティも、そのコレクションと ID に制限される必要があります。そうしないと、インデックスは投影されません。この制限は、配列として投影されるコレクション・プロパティ上のインデックスに適用されます。リストとして投影されるコレクション上のインデックスには適用されません。詳細は、"クラスの定義と使用" の "コレクション・プロパティの SQL プロジェクションの制御" を参照してください。

コレクションの要素値またはキー値に対応するインデックスは、インデックスを使用したデータの保存やインデックス固有の照合など、標準インデックスのすべての機能を持つこともできます。

InterSystems SQL は、FOR SOME %ELEMENT 述語を指定することによって、コレクション・インデックスを使用することができます。

2.3.2.8 (ELEMENTS) および (KEYS) を使用したデータ型プロパティのインデックス作成

データ型プロパティのインデックスを作成するために、BuildValueArray() メソッドを使用してインデックス値配列を作成することもできます。このメソッドは、プロパティ値をキーおよび要素の配列に構文解析します。このメソッドは、それを行うために、関連付けられているプロパティの値から派生した要素値のコレクションを作成します。このメソッドは、既存のコレクション・プロパティでは機能しません。

BuildValueArray() を使用してインデックス値配列を作成する場合、その構造はインデックス作成に適しています。

BuildValueArray() メソッドには propertyNameBuildValueArray() という名前があり、そのシグニチャは以下のようになります。

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

以下はその説明です。

- BuildValueArray() メソッドの名前は、複合メソッドに対する一般的な方法でプロパティ名から派生します。
- 最初の引数はプロパティ値です。
- 2 番目の引数は、参照によって渡される配列です。これは、キーによって添え字が付けられた配列が要素と等しい、キーと要素のペアを含む配列です。
- このメソッドは %Status 値を返します。

2.3.2.9 配列コレクションのインデックス作成

配列コレクションのプロパティにインデックスを指定するには、そのインデックス定義にキーを追加する必要があります。例えば以下のようにします。

Class Definition

```
Class MyApp.Branch Extends %Persistent [ DdlAllowed ]
{
  Property Name As %String;
  Property Employees As Array Of MyApp.Employee;

  Index EmpIndex On (Employees(KEYS), Employees(ELEMENTS));
}
```

これらのキーは、配列要素の子テーブルにある行の RowID を特定します。このキーがないと、親テーブルから子テーブルにインデックスが投影されません。この投影が発生しないことから、親テーブルへの INSERT 操作が失敗します。

2.3.2.10 埋め込みオブジェクト (%SerialObject) のプロパティのインデックス作成

埋め込みオブジェクト内のプロパティにインデックスを作成するには、その埋め込みオブジェクトを参照する永続クラスでインデックスを作成します。次の例で示しているように、プロパティ名に、テーブル (%Persistent クラス) 内の参照フィールドの名前および埋め込みオブジェクト (%SerialObject) 内のプロパティを指定する必要があります。

Class Definition

```
Class Sample.Person Extends (%Persistent) [ DdlAllowed ]
{  Property Name As %String(MAXLEN=50);
   Property Home As Sample.Address;
   Index StateInx On Home.State;
}
```

ここでは、Home は、State プロパティを含む埋め込みオブジェクト Sample.Address を参照する Sample.Person 内のプロパティです。以下に例を示します。

Class Definition

```
Class Sample.Address Extends (%SerialObject)
{  Property Street As %String;
   Property City As %String;
   Property State As %String;
   Property PostalCode As %String;
 }
```

永続クラスのプロパティ参照に関連付けられている埋め込みオブジェクトのインスタンス内のデータ値のみにインデックスが作成されます。%SerialObject プロパティにインデックスを直接作成することはできません。%Library.SerialObject の SqlCategory (および SqlCategory を明示的に定義していない %SerialObject のすべてのサブクラス) は STRING です。

埋め込みオブジェクトのプロパティで、SQL 文 CREATE INDEX を使用してインデックスを定義することもできます。以下に例を示します。

SQL

```
CREATE INDEX StateIdx ON TABLE Sample.Person (Home_State)
```

詳細は、"シリアル・オブジェクトの概要" と "埋め込みオブジェクト (%SerialObject)" を参照してください。

2.4 インデックス・タイプのまとめ

InterSystems SQL には、さまざまな状況で有用なさまざまなインデックス・タイプが用意されています。スキーマを最適化するには、ユース・ケースに最適に機能するインデックスを定義する必要があります。以下の表は、これらのインデックス・タイプをまとめたものです。各タイプの詳細は、リンクされたセクションを参照してください。

インデックスを追加することにより、テーブルが占めるストレージ領域の量と、データの更新または追加の際に実行される操作の数の両方が増えることに留意してください。

テーブル 2-1:

インデックス・タイプ	説明	フィールド・タイプ	ユース・ケースの例
標準	特定の列内の値を共有する行をグループ化します	任意	ほとんどの状況で効率的
ビットマップ	列内の一意の値ごとに、その値が含まれる行を示すビット文字列を格納します ビットマップ・インデックスを定義すると、ビットマップ・エクステント・インデックスが自動的に作成されます	数値	以下を使用するクエリ： 1 つのテーブルに対する複数の条件の AND または OR RANGE 条件
ビットマップ・エクステント	ビットマップ・インデックスが作成されたとき、または CREATE TABLE を使用してテーブルが定義されたときに自動的に作成される存在ビットマップ	数値（"ビットマップ" を参照）	以下を使用するクエリ： COUNT コマンド
ビットスライス	数値をバイナリに変換し、そのバイナリ値をビットマップとして格納します	数値	以下を使用するクエリ： SUM、COUNT または AVG コマンド
列指向	フィールドのデータのコピーを、圧縮され、ベクトル化された形式で格納します	数値、カーディナリティが低い短い文字列	以下を使用するクエリ： SUM または AVG コマンド RANGE 条件

2.5 ビットマップ・インデックス

ビットマップ・インデックスは特殊なタイプのインデックスで、指定したインデックス付きのデータ値に対応する一連の ID 値を表す一連のビット文字列を使用します。

ビットマップ・インデックスには以下のような重要な特長があります。

- ビットマップは高圧縮されています。つまり、ビットマップ・インデックスは標準インデックスに比べてサイズをはるかに小さくすることができるので、ディスクおよびキャッシュの使用量を大幅に削減できます。
- ビットマップ演算は、トランザクション処理に最適化されています。テーブルでビットマップ・インデックスを使用すると、標準インデックスを使用する場合に比べ、パフォーマンスが低下しません。
- ビットマップでの論理演算（カウント、AND、OR）を使用すると、パフォーマンスが向上します。
- SQL エンジンは、ビットマップ・インデックスを駆使する、特別な最適化機能を備えています。

ビットマップ・インデックスの作成は、テーブルに固有の ID フィールドの性質によって異なります。

- テーブルの ID フィールドが正の整数値の単一フィールドとして定義されていると、この ID フィールドを使用してフィールドにビットマップ・インデックスを定義できます。このタイプのテーブルでは、システムによって割り当てられた一意の正整数 ID が使用されるか、または IdKey を使用してカスタム ID 値が定義されます。このとき、IdKey

は、%Integer タイプで MINVAL が 0 より大きい単一のプロパティか、%Numeric タイプで SCALE が 0 で MINVAL が 0 より大きい単一のプロパティに基づきます。

- テーブルの ID フィールドが正の整数値の単一フィールドとして定義されていない場合 (子テーブルなど)、正の整数をとる %BID (ビットマップ ID) フィールドを定義できます。%BID フィールドはサロゲート ID フィールドとして機能します。これによって、このテーブルのフィールドにビットマップ・インデックスを作成できます。

ビットマップ・インデックスは、以下の制限事項に従って標準インデックスと同様に機能します。インデックス値は照合されており、複数のフィールドの結合に対しインデックスを作成できます。

この章では、ビットマップ・インデックスに関連する以下の項目について説明します。

- ビットマップ・インデックス演算
- DDL を使用したビットマップ・インデックスの定義
- クラス定義を使用したビットマップ・インデックスの定義
- ビットマップ・エクステント・インデックスの生成
- インデックス・タイプの選択
- ビットマップ・インデックスに関する制限事項
- ビットマップ・インデックスの維持
- ビットマップ・チャンクの SQL 操作

2.5.1 ビットマップ・インデックス演算

ビットマップ・インデックスは以下のように動作します。多数の列を持つ Person テーブルがあるとします。

図 2-1: Person テーブル

Person					
RowID	Name	Age	State	Job	...
1	Smith	24	NY	Lawyer	...
2	Jones	35	NY	Doctor	...
3	Presley	48	CA	Farmer	...
4	Nixon	72	NY	Singer	...
...	...	...	...	...	...

このテーブルの各行には、システムにより RowID 番号 (増加する整数値) が割り当てられます。ビットマップ・インデックスは、一連のビット文字列 (1 または 0 を含む文字列) を使用します。ビット文字列では、ビットの順序の位置がインデックス付きテーブルの RowID に対応します。上記のテーブルで State が“NY”である場合を例にとると、ビット文字列の“NY”を含む行に対応する各位置には 1 が、その他の位置には 0 が入ります。

例えば、State のビットマップ・インデックスは以下のとおりです。

図 2-2: State ビットマップ・インデックス

StateIndex					
	Row 1	Row 2	Row 3	Row 4	...
CA	0	0	1	0	...
NY	1	1	0	1	...
WY	0	0	0	0	...
...	...	...	...	...	...

また、Age は以下のようになります。

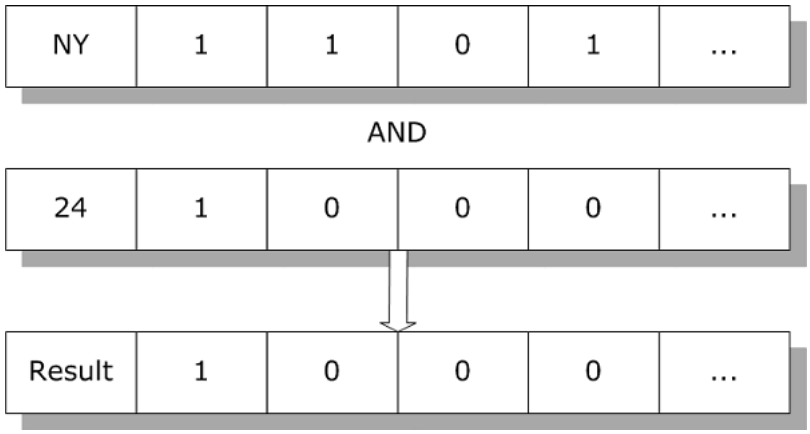
図 2-3: Age ビットマップ・インデックス

AgeIndex					
	Row 1	Row 2	Row 3	Row 4	...
24	1	0	0	0	...
35	0	1	0	0	...
48	0	0	1	0	...
...	...	...	...	...	...

注釈　ここで示す Age フィールドは、通常のデータ・フィールドにすることも、値を確実に派生できるフィールド (Calculated および SQLComputed) にすることもできます。

通常の演算にビットマップ・インデックスを使用する方法とは別に、SQL エンジンは、複数インデックスの結合を使用した特別なセットベースの演算を、ビットマップ・インデックスを使用して効率的に実行できます。例えば、ニューヨークに住む 24 歳以上に当てはまる Person のすべてのインスタンスを得るために、SQL エンジンは Age と State の論理 AND を実行します。

図 2-4: 複数のインデックスの使用



結果として得られたビットマップには、検索基準に当てはまるすべての行のセットが含まれます。SQL エンジンは、これを使用して行からデータを返します。

SQL エンジンは、ビットマップ・インデックスを以下の演算に使用できます。

- 任意のテーブルの複数の条件について AND を実行
- 任意のテーブルの複数の条件について OR を実行
- 任意のテーブルの RANGE 条件
- 任意のテーブルの COUNT 演算

2.5.2 DDL を使用したビットマップ・インデックスの定義

テーブルの定義に DDL 文を使用している場合、以下の DDL コマンドを使用して、正整数 ID を持つテーブルのビットマップ・インデックスを作成および削除することもできます。

- CREATE INDEX
- DROP INDEX

これは、CREATE INDEX 文に BITMAP キーワードを追加する必要がある点以外は、標準インデックスの生成とまったく同じです。

SQL

```
CREATE BITMAP INDEX RegionIDX ON TABLE MyApp.SalesPerson (Region)
```

2.5.3 クラス定義を使用した IdKey ビットマップ・インデックスの定義

テーブルの ID が値（一意の正の整数に制限されている値）のフィールドの場合、標準インデックスの作成と同様に、新規インデックス・ウィザードを使用するか、クラス定義のテキストを編集することで、ビットマップ・インデックス定義をクラス定義に追加できます。ただし、ビットマップ・インデックスの場合は、インデックスの Type を“bitmap”に指定する必要があります。

Class Definition

```
Class MyApp.SalesPerson Extends %Persistent [DdlAllowed]
{
 Property Name As %String;
 Property Region As %Integer;

 Index RegionIDX On Region [Type = bitmap];
}
```

2.5.4 クラス定義を使用した %BID ビットマップ・インデックスの定義

テーブルの ID が正整数に制限されていない場合、%BID フィールドを作成して、これをビットマップ・インデックス定義の作成に使用できます。このオプションは、特定のテーブルに、その ID フィールドのデータ型にかかわらず使用でき、複数のフィールド（子テーブルを含む）で構成される IDKEY の場合でも使用可能です。%BID ビットマップは、既定の構造テーブルと%Storage.SQL テーブルのいずれかのデータ・ストレージ・タイプに作成できます。この機能は、"あらゆるテーブルのビットマップ" (BAT) と呼ばれます。

こういったテーブルでビットマップ・インデックスを使用できるようにするには以下の手順に従います。

1. %BID フィールドを定義するか、既存のフィールドを %BID フィールドとして特定します。このフィールドのデータ型は、一意の正整数値に制限する必要があります。例えば、このテーブルの IDKey は、値が正整数に制限されていない 2 つのフィールドの複合です。これにより、データ型 (%Counter) が正整数である MyBID フィールドが %BID フィールドの候補となります。

   Class Definition

   ```
   Class MyTable Extends %Persistent [ DdlAllowed ]
   {
     Property IdField1 As %Integer;
     Property IdField2 As %Integer;
     Property MyBID As %Counter; /* %BID Field */

     Index IDIdx On (IDfield1, IDfield2) [ IdKey, Unique ];
   }
   ```

2. SQL コンパイラに対して %BID フィールドを特定する BIDField クラス・パラメータを定義します。その値を、%BID フィールドの SQLFieldName に設定します。例えば以下のようにします。

   Class Definition

   ```
   Class MyTable Extends %Persistent [ DdlAllowed ]
   {
     Parameter BIDField = "MyBID"; /* BIDField Class Parameter */

     Property IdField1 As %Integer;
     Property IdField2 As %Integer;
     Property MyBID As %Counter;

     Index IDIdx On (IDfield1, IDfield2) [ IdKey, Unique ];
   }
   ```

3. BATKey インデックスを定義します。このインデックスは、SQL クエリ・プロセッサのマスタ・マップとデータ・マップとして機能します。多くの場合、%BID フィールドは、BATKey インデックスの 1 番目の添え字です。これにより、このマップ・データには、IDKEY フィールドと、最高速のアクセスを必要とするあらゆる補足プロパティを追加できます。%BID フィールドにインデックスを設定する必要があります。例えば以下のようにします。

Class Definition

```
Class MyTable Extends %Persistent [ DdlAllowed ]
{
  Parameter BIDField = "MyBID";

  Property IdField1 As %Integer;
  Property IdField2 As %Integer;
  Property MyBID As %Counter;

  Index MyBATKey On MyBID [ Type = key, Unique ]; /* BATKey Index */
  Index IDIdx On (IDfield1, IDfield2) [ IdKey, Unique ];
}
```

4. SQL コンパイラに対して BATKey インデックスを特定する BATKey クラス・パラメータを定義します。その値を、BATKey インデックスの SQLFieldName に設定します。例えば以下のようにします。

Class Definition

```
Class MyTable Extends %Persistent [ DdlAllowed ]
{
  Parameter BIDField = "MyBID";
  Parameter BATKey = "MyBATKey"; /* BATKey Class Parameter */

  Property IdField1 As %Integer;
  Property IdField2 As %Integer;
  Property MyBID As %Counter;

  Index MyBATKey On MyBID [ Type = key, Unique ];
  Index IDIdx On (IDfield1, IDfield2) [ IdKey, Unique ];
}
```

5. %BID ロケータ・インデックスを定義するか、既存のインデックスを %BID ロケータ・インデックスとして特定します。このインデックスによって、%BID インデックスがテーブルの IDKey フィールドに関連付けられます。例えば以下のようにします。

Class Definition

```
Class MyTable Extends %Persistent [ DdlAllowed ]
{
  Parameter BIDField = "MyBID";
  Parameter BATKey = "MyBATKey";

  Property IdField1 As %Integer;
  Property IdField2 As %Integer;
  Property MyBID As %Counter;

  Index MyBATKey On MyBID [ Type = key, Unique ];
  Index IDIdx On (IDfield1, IDfield2) [ IdKey, Unique ];
  Index BIDLocIdx On (IDfield1, IDfield2, MyBID) [ Unique ]; /* %BID Locator Index */
}
```

これで、このテーブルではビットマップ・インデックスがサポートされるようになります。必要に応じて、標準構文を使用してビットマップ・インデックスを定義できます。例えば、`Index RegionIDX On Region [Type = bitmap];` と定義します。

このように作成したテーブルではビットスライス・インデックスもサポートされます。標準構文を使用してビットスライス・インデックスを定義することもできます。

2.5.5 ビットマップ・エクステント・インデックスの生成

ビットマップ・インデックスはビットマップ・エクステント・インデックスが必要です。1 つ以上のビットマップ・インデックスが定義されている場合、永続クラスを定義するとビットマップ・エクステント・インデックスのみが生成されます。したがって、ビットマップ・インデックスを含む永続クラスをコンパイルする場合、そのクラスに対するビットマップ・エクステント・インデックスが定義されていないと、クラス・コンパイラはビットマップ・エクステント・インデックスを生成します。CREATE TABLE DDL 文を使用して定義したテーブルでは、自動的にビットマップ・エクステント・インデックスが生成されます。

永続クラス定義からすべてのビットマップ・インデックスを削除すると、ビットマップ・エクステント・インデックスが自動的に削除されます。ただし、ビットマップ・エクステント・インデックスの名前を変更（CREATE BITMAPEXTENT INDEX コマンドを使用するなど）してビットマップ・インデックスを削除しても、ビットマップ・エクステント・インデックスは削除されません。

あるクラスのインデックスを作成する際、明示的にビットマップ・エクステント・インデックスを構築した場合、またはビットマップ・インデックスを作成していて、ビットマップ・エクステント・インデックスが空である場合に、ビットマップ・エクステント・インデックスが作成されます。

クラスは、プライマリ・スーパークラスから、定義済みまたは生成されたビットマップ・エクステント・インデックスを継承します（存在する場合）。プライマリ・スーパークラスから継承されたビットマップ・エクステント・インデックスは、ビットマップ・インデックスと見なされ、サブクラスでのビットマップ・エクステント・インデックスの生成をトリガします（そのサブクラスで、ビットマップ・エクステント・インデックスが明示的に定義されていない場合）。ビットマップ・エクステント・インデックスは以下のように定義されます。

Class Definition

```
Class Test.Index Extends %Registered Object
{
  Property Data As %Integer [ InitialExpression = {$RANDOM(100000)}];

  Index DataIndex On Data [ Type = bitmap ];
  Index ExtentIndex [ Extent, Type = bitmap ];
}
```

InterSystems IRIS では、将来の可能性を考慮してスーパークラスでビットマップ・エクステント・インデックスは生成されません。つまり、InterSystems IRIS では、`type = bitmap` であるインデックスが存在しない限り、永続クラスでビットマップ・エクステント・インデックスが生成されることはありません。将来一部のサブクラスで `type = bitmap` のインデックスが導入されるかもしれないという推定だけでは不十分です。

注釈　運用システム（ユーザが特定のクラスを頻繁に使用し、そのクラスをコンパイルし、続いてビットマップ・インデックス構造を作成しているような環境）でクラスにビットマップ・インデックスを追加するプロセスでは、特別な注意が必要です。そのようなシステムでは、ビットマップ・エクステント・インデックスは、コンパイルの完了とインデックスの作成手順の中間で生成される可能性があります。これにより、インデックス作成手順では、ビットマップ・エクステント・インデックスが暗黙的に作成されず、ビットマップ・エクステント・インデックスの処理が部分的に完了することがあります。

2.5.6 インデックス・タイプの選択

以下は、ビットマップ・インデックスと標準インデックスのどちらを選択するべきかを判断するための指針です。一般的には、次のどの種類のキーおよび参照にインデックスを作成する場合でも標準インデックスを使用します。

- 主キー
- 外部キー
- 一意のキー
- リレーションシップ
- 単純オブジェクト参照

上記以外のキーまたは参照の場合、一般的にはビットマップ・インデックスの使用をお勧めします（システムによって割り当てられた数値 ID 番号をテーブルで使用していることが前提となります）。

その他の要素

- 通常、複数のプロパティに対するビットマップ・インデックスよりも、プロパティごとに個別のビットマップ・インデックスの方がパフォーマンスが高くなります。これは、SQL エンジンは AND 演算と OR 演算を使用して個別のビットマップ・インデックスを効率的に結合できるためです。

- プロパティ（または、まとめてインデックスを作成する必要がある複数のプロパティ）の個別値（または値の組み合わせ）が 10,000 ～ 20,000 個を超える場合は、標準インデックスの使用を検討してください。しかし、これらの値が不均一に分散されているために多数の行に対する値の数が非常に少ない場合は、ビットマップ・インデックスを使用した方が有利です。常に、インデックスによる要求の全体量が少しでも少なくなる方法を使用します。

2.5.7 ビットマップ・インデックスに関する制限事項

すべてのビットマップ・インデックスに、以下のような制限事項があります。

- UNIQUE 列にビットマップ・インデックスを定義することはできません。
- ビットマップ・インデックスには、値を保存できません。
- ID フィールドの SqlCategory が INTEGER、DATE、POSIXTIME、または NUMERIC（scale=0）である場合を除き、フィールドにビットマップ・インデックスを定義することはできません。
- 格納しているレコード数が 1,000,000 個を超えるテーブルでは、一意の値の数が 10,000 個を超えたときに、標準インデックスよりもビットマップ・インデックスの方が効率が低下します。そのため、巨大なテーブルについては、10,000 個を超える一意の値が含まれている（またはその可能性のある）フィールドに対してビットマップ・インデックスを使用しないようにしてください。また、テーブルのサイズにかかわらず、20,000 個を超える一意の値が含まれている可能性のあるフィールドにはビットマップ・インデックスを使用しないでください。これらは一般的な概数であり、正確な数ではありません。

以下のようなテーブルでビットマップ・インデックスを使用するには、%BID プロパティを作成する必要があります。

- 整数以外のフィールドを一意の ID キーとして使用している。
- 複数フィールドの ID キーを使用している。
- 親子リレーションシップ内の子テーブルである。

$SYSTEM.SQL.Util.SetOption() メソッド `SET status=$SYSTEM.SQL.Util.SetOption("BitmapFriendlyCheck",1,.oldval)` を使用すると、この制限をコンパイル時にチェックし、定義されたビットマップ・インデックスが %Storage.SQL クラス内で許可されるかどうかを判断するようにシステム全体用の構成パラメータを設定できます。このチェックは、%Storage.SQL を使用するクラスにのみ適用されます。既定値は 0 です。$SYSTEM.SQL.Util.GetOption("BitmapFriendlyCheck") を使用すると、このオプションの現在の構成を判断できます。

2.5.7.1 アプリケーション・ロジックの制限事項

ビットマップ構造は、ビット文字列の配列で表現できます。この配列の各要素が固定のビット数による 1 つの「チャンク」を表しています。未定義はすべてのビットが 0 のチャンクと同等であるため、配列はスパースになることがあります。すべてのビットが 0 のチャンクを表す配列要素は、存在している必要は一切ありません。このため、アプリケーション・ロジックでは、0 値のビットの $BITCOUNT(str,0) カウントに依存しないようにする必要があります。

ビット文字列は内部フォーマットを含んでいるため、アプリケーション・ロジックでは、ビット文字列の物理長に依存することや、同じビット値を持つ 2 のビット文字列の等価性に依存することは絶対に避けてください。ロールバック操作後、ビット文字列はトランザクション前のビット値にリストアされます。ただし、内部フォーマットのために、ロールバックされたビット文字列はトランザクション前のビット文字列と等しくならないことや、同じ物理長にならないことがあります。

2.5.8 ビットマップ・インデックスの維持

揮発性のテーブル（多数の INSERT 操作および DELETE 操作が実行されるもの）では、ビットマップ・インデックス用のストレージは徐々に効率が低下する可能性があります。ビットマップ・インデックスを維持するには、%SYS.Maint.Bitmap ユーティリティ・メソッドを実行してビットマップ・インデックスを圧縮し、それらを最適な効率が得られるように復元します。1 つのクラスを対象にビットマップ・インデックスを圧縮するには OneClass() メソッド、ネームスペース全体でビットマップ・

インデックスを圧縮するには Namespace() メソッドをそれぞれ使用できます。これらの維持メソッドは、実働システム上で実行できます。

%SYS.Maint.Bitmap ユーティリティ・メソッドを実行した結果は、このメソッドを呼び出したプロセスに書き込まれます。これらの結果は、クラス %SYS.Maint.BitmapResults にも書き込まれます。

2.5.9 ビットマップ・チャンクの SQL 操作

InterSystems SQL には、ビットマップ・インデックスを直接操作するための以下の拡張機能が用意されています。

- %CHUNK 関数
- %BITPOS 関数
- %BITMAP 集約関数
- %BITMAPCHUNK 集約関数
- %SETINCHUNK 述語条件

これらの拡張機能はすべて、ビットマップ表現に関する InterSystems SQL の規約に従い、正の整数のセットを一連のビットマップ・チャンク (それぞれ最大 64,000 個の整数) として表します。

これらの拡張機能により、クエリ内でも埋め込み SQL でも、特定の条件やフィルタをより簡単かつ効率的に操作することができます。埋め込み SQL では、これらの拡張機能によって、単一チャンク・レベルでは特に、ビットマップのシンプルな入力と出力が実現します。これらの拡張機能では、%BITMAP() および %SQL.Bitmap クラスによって処理される、完全なビットマップの処理がサポートされています。また、外部キー値、親による子テーブルの参照、関連付けのどちらか一方の列など、非 RowID 値のビットマップ処理も可能です。

例えば、指定したチャンクのビットマップを出力するには、以下のように入力します。

SQL

```
SELECT %BITMAPCHUNK(Home_Zip) FROM Sample.Person
WHERE %CHUNK(Home_Zip)=2
```

テーブル全体のチャンクをすべて出力するには、以下のように入力します。

SQL

```
SELECT %CHUNK(Home_Zip),%BITMAPCHUNK(Home_Zip) FROM Sample.Person
GROUP BY %CHUNK(Home_Zip) ORDER BY 1
```

2.5.9.1 %CHUNK 関数

%CHUNK(f) は、ビットマップ・インデックスが付いたフィールドの f 値に対するチャンク割り当てを返します。これは、f¥64000+1 として計算されます。ビットマップ・インデックスが付いたフィールドでないフィールドまたは値 f の %CHUNK(f) は、常に 1 を返します。

2.5.9.2 %BITPOS 関数

%BITPOS(f) は、ビットマップ・インデックスが付いたフィールドの f 値に割り当てられた、そのチャンク内のビット位置を返します。これは、f#64000+1 として計算されます。ビットマップ・インデックスが付いたフィールドでないフィールドまたは値 f の %BITPOS(f) は、その整数値より 1 大きい値を返します。文字列の整数値は 0 です。

2.5.9.3 %BITMAP 集約関数

集約関数 %BITMAP(f) は、多数の f 値を 1 つの %SQL.Bitmap オブジェクトに結合します。ここでは、結果セット内のそれぞれの値 f について、適切なチャンク内の f に対応するビットが 1 に設定されます。前述のすべての f は通常、正の整数のフィールド (または式) です。一般的には RowID ですが、必ず RowID というわけではありません。

2.5.9.4 %BITMAPCHUNK 集約関数

集約関数 %BITMAPCHUNK(f) は、フィールド f の多くの値を 64,000 ビットの InterSystems SQL 標準ビットマップ文字列に結合します。ここでは、セット内のそれぞれの値 f について、ビット f#64000+1=%BITPOS(f) が 1 に設定されます。%CHUNK(f) の値に関係なく、結果内でビットが設定されます。%BITMAPCHUNK() は、空のセットについては NULL を生成し、他の集約関数と同様に、入力に含まれる NULL 値は無視します。

2.5.9.5 %SETINCHUNK 述語条件

条件 (f %SETINCHUNK bm) が true になるのは、($BIT(bm,%BITPOS(f))=1) の場合のみです。bm は、ビットマップ式文字列 (例 : 入力ホスト変数 :bm)、%BITMAPCHUNK() 集約関数の結果などです。%CHUNK(f) の値に関係なく、<bm> ビットがチェックされます。<bm> がビットマップでない場合、または NULL である場合、この条件は FALSE を返します。(f %SETINCHUNK NULL) は FALSE になります (UNKNOWN ではありません)。

2.6 ビットスライス・インデックス

ビットスライス・インデックスは、数値データ・フィールドを特定の数値演算に使用する場合に、そのフィールドに使用します。ビットスライス・インデックスは、各数値データをバイナリのビット文字列として表します。ビットスライス・インデックスは、ビットマップ・インデックスのようにブーリアン・フラグを使用して数値データのインデックスを作成するのではなく、バイナリの各数値を表し、バイナリ値の桁ごとにビットマップを作成してそのバイナリの桁が 1 である行を記録します。これは、以下の演算のパフォーマンスを大幅に向上させることができる非常に特殊なタイプのインデックスです。

- SUM、COUNT、または AVG 集約計算(ビットスライス・インデックスは COUNT(*) 計算には使用できません)。ビットスライス・インデックスは、その他の集約関数には使用されません。
- `TOP n ...` で指定されたフィールド。`ORDER BY field` 演算。
- `WHERE field > n` や `WHERE field BETWEEN lownum AND highnum` など、範囲条件演算で指定されたフィールド。

SQL オプティマイザは、定義されたビットスライス・インデックスを使用するべきかどうかを決定します。一般に、オプティマイザは、かなりの数 (数千) の行が処理される場合にのみ、ビットスライス・インデックスを使用します。

文字列データ・フィールドについてビットスライス・インデックスを作成できますが、ビットスライス・インデックスは、これらのデータ値をキャノニック形式の数値として表します。つまり、“abc” のような数値以外の文字列には 0 としてインデックスが作成されます。このタイプのビットスライス・インデックスは、文字列フィールドの値があるレコードを COUNT ですばやくカウントし、NULL のレコードをカウントしない場合に使用できます。

以下の例では、Salary はビットスライス・インデックスの候補となります。

SQL

```
SELECT AVG(Salary) FROM SalesPerson
```

ビットスライス・インデックスは、WHERE 節を使用するクエリで集約計算に使用できます。これは、WHERE 節に多数のレコードが含まれる場合に最も効果的です。以下の例では、SQL オプティマイザは多くの場合、Salary にビットスライス・インデックス (定義されている場合) を使用します。その場合、Region でも、定義されたビットマップを使用するか、Region にビットマップ一時ファイルを生成して、ビットマップ・インデックスが使用されます。

SQL

```
SELECT AVG(Salary) FROM SalesPerson WHERE Region=2
```

ただし、WHERE 条件がインデックスにより満たされず、集約されるフィールドを含むテーブルを読み取ることによって実行されなければならない場合には、ビットスライス・インデックスは使用されません。以下の例では、Salary でビットスライス・インデックスを使用しません。

SQL

```
SELECT AVG(Salary) FROM SalesPerson WHERE Name LIKE '%Mc%'
```

ビットスライス・インデックスは、数値を含むすべてのフィールドに対して定義できます。InterSystems SQL では、ObjectScript $FACTOR 関数で説明されているように、scale パラメータを使用して小数をビット文字列に変換します。ビットスライス・インデックスは、データ型が文字列のフィールドに定義できます。この場合、数値以外の文字列データ値は、ビットスライス・インデックスを定義するために 0 として扱われます。

システムで割り当てた行 ID に正整数値が指定されているテーブルのフィールドや、%BID プロパティが定義されているテーブル (ビットマップ (およびビットスライス) インデックスのサポート用) のフィールドにビットスライス・インデックスを定義できます。

ビットスライス・インデックスは、複数のフィールドの連結ではなく、単一のフィールド名にのみ定義できます。WITH DATA 節は指定できません。

次に示す例では、ビットスライス・インデックスとビットマップ・インデックスを比較しています。行 1、2、および 3 の値 1、5、および 22 にビットマップ・インデックスを作成すると、ビットマップ・インデックスは以下の値に対してインデックスを作成します。

```
^gloI("bitmap",1,1)= "100"
^gloI("bitmap",5,1)= "010"
^gloI("bitmap",22,1)="001"
```

行 1、2、および 3 の値 1、5、および 22 にビットスライス・インデックスを作成すると、まず、その値が以下のビット値に変換されます。

```
1 =  00001
5 =  00101
22 = 10110
```

その後で、以下に示すビットに対してインデックスを作成します。

```
^gloI("bitslice",1,1)="110"
^gloI("bitslice",2,1)="001"
^gloI("bitslice",3,1)="011"
^gloI("bitslice",4,1)="000"
^gloI("bitslice",5,1)="001"
```

この例では、ビットマップ・インデックスの値 22 には 1 つのグローバル・ノードの設定が必要になります。ビットスライス・インデックスの値 22 には 3 つのグローバル・ノードの設定が必要になります。

INSERT または UPDATE では、単一のビット文字列ではなく、すべての n ビットスライスでビットを設定する必要があることに注意してください。このような追加のグローバル設定操作は、ビットスライス・インデックスの入力に関連する INSERT 操作と UPDATE 操作のパフォーマンスに影響を及ぼすことがあります。INSERT、UPDATE、または DELETE 操作を使用してビットスライス・インデックスを生成および維持すると、ビットマップ・インデックスや通常のインデックスを使用する場合よりも速度が遅くなります。複数のビットスライス・インデックスを維持したり、頻繁に更新されるフィールドでビットスライス・インデックスを維持すると、パフォーマンス・コストが増える可能性があります。

揮発性のテーブル (多数の INSERT 操作、UPDATE 操作および DELETE 操作が実行されるもの) では、ビットスライス・インデックス用のストレージは徐々に効率が低下する可能性があります。%SYS.Maint.Bitmap ユーティリティ・メソッドは、ビットマップ・インデックスとビットスライス・インデックスの両方を圧縮して、元の高い効率を復元します。詳細は、"ビットマップ・インデックスの維持" を参照してください。

2.7 列指向インデックス

列指向インデックスは、頻繁にクエリされるものの、テーブルに基盤となる行ストレージ構造があるフィールドに使用します。既定では、テーブルのそれぞれの行が $LIST として別々のグローバル添え字で格納されます。列指向インデックスには、特定のフィールドのデータが圧縮ベクトル化形式で保存されます。

InterSystems SQL DDL を使用して列指向インデックスを定義するには、次のように CREATE INDEX の `CREATE COLUMNAR INDEX` 構文を使用します。

```
CREATE COLUMNAR INDEX indexName ON table(column)
```

永続クラスに列指向インデックスを定義するには、次のように、定義するインデックスに `type = columnar` キーワードを指定します。

```
Index indexName ON propertyName [ type = columnar ]
```

以下の DDL の例は、テーブルの特定の列に列指向インデックスを定義する方法を示しています。

```
CREATE TABLE Sample.BankTransaction (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2),
  Type VARCHAR(10))

CREATE COLUMNAR INDEX AmountIndex
ON Sample.BankTransaction(Amount)
```

このテーブルの Amount 列に対して AVG による集約計算を実行し、その結果の表示を預金額のみに絞り込むとします。

SQL

```
SELECT AVG(Amount) FROM Sample.BankTransaction WHERE Type = 'Deposit'
```

この計算では、1 つの列 (`Amount`) の一部の行 (`WHERE Type = 'Deposit'`) のみを必要とするとき、メモリに各 $LIST グローバルをロードする必要があります。列指向インデックスを定義したフィールドに対する AVG 計算の実行でクエリ・プランからアクセスする情報は、$LIST グローバルから直接取得されるのではなく、この列指向インデックスから取得されます。

シリアル・プロパティやサブクラスには列指向インデックスを定義できません。ただし、サブクラスの非シリアル・プロパティ (%Integer など) にはインデックスを定義できます。

列指向インデックスはビットマップ・インデックスに類似していますが、等値条件ではビットマップ・インデックスよりもわずかに効率が低下します。ビットマップ・インデックスでは値ごとにすでにビット文字列が存在しますが、列指向インデックスの場合は、ベクトル化演算を実行して列指向インデックスから値を取得します。範囲条件では、列指向インデックスの方が効率的であることが普通です。ビットマップ・インデックスでは複数のビット文字列を結合する必要がありますが、列指向インデックスでは、1 回のベクトル化演算で同じ演算をこなすことができます。

列指向インデックスとテーブルのストレージ・レイアウトの定義の詳細は、"SQL テーブルのストレージ・レイアウトの選択"を参照してください。

2.8 インデックスの構築

インデックスは以下のように構築または再構築できます。

- BUILD INDEX SQL コマンドを使用して、指定したインデックスを構築するか、テーブル、スキーマ、または現在のネームスペースに対して定義したすべてのインデックスを構築します。このオプションは、本稼働システムで使用しても安全です。
- 管理ポータルを使用して、指定したクラス (テーブル) のインデックスをすべて再構築します。
- %BuildIndices() メソッド (または %BuildIndicesAsync() メソッド) を使用します。

インデックス・システムの構築でお勧めできる方法は、BUILD INDEX SQL コマンドを使用することです。インデックスの構築は、以下を実行します。

1. 現在のインデックスの内容を削除します。
2. メインのテーブルをスキャンし (各行を読み取り)、テーブルの各行にインデックスのエントリを追加します。必要に応じ、$SortBegin と $SortEnd を使用して、並列実行と効率的なバッチ並べ替えのための低レベル最適化を適用します。

本稼働システムで BUILD INDEX を使用する場合、目的のインデックスは一時的に選択不可とされます。したがって、構築中のインデックスをクエリで使用することはできません。このインデックスを使用するクエリはパフォーマンスが低下します。

2.8.1 BUILD INDEX によるインデックスの構築

クラス・レベルまたは DDL レベルでインデックスを作成した後は、BUILD INDEX コマンドを使用して、そのインデックスを構築する必要があります。この文を使用すると、ネームスペースのすべてのインデックス、スキーマのすべてのインデックス、またはコマンドで指定したインデックスのみを構築できます。既定では、各テーブルのインデックスを作成する前に、そのテーブルに対するエクステント・ロックが取得されます。インデックスの作成が終了するとロックが解除され、そのインデックスをアクティブ・システムで安全に使用できるようになります。構築中のインデックスをクエリで使用することはできません。BUILD INDEX コマンドの実行中に INSERT コマンドでテーブルに挿入されたデータまたは UPDATE コマンドで更新されたテーブルのデータは、構築プロセスに取り込まれます。

BUILD INDEX では、発生したエラーがあれば、現在のプロセスに対するジャーナル設定を使用してログに記録されます。%NOLOCK オプションを指定してロック動作を無効化でき、%NOJOURN オプションを指定してジャーナル動作を無効化できます。

以下の例では、`MyApp.Salesperson` クラスにインデックスを構築します。

Class Member

```
BUILD INDEX FOR TABLE MyApp.SalesPerson
BUILD INDEX FOR TABLE MyApp.SalesPerson INDEX NameIdx, SSNKey
```

1 番目の文で、指定した名前のテーブル (クラス) のインデックスをすべて構築しています。2 番目の文で、NameIdx インデックスと SSNKey インデックスのみを構築しています。

2.8.2 管理ポータルによるインデックスの構築

テーブルに既存のインデックスを構築 (インデックスを再構築) するには、以下の操作を実行します。

1. 管理ポータルで、**[システムエクスプローラ]**、[SQL] の順に選択します。ページ上部の現在のネームスペース名をクリックして、ネームスペースを選択します。利用可能なネームスペースのリストが表示されます。ネームスペースの選択後に、画面の左側にある **[スキーマ]** ドロップダウン・リストを選択します。これには、現在のネームスペースのスキーマのリストと、各スキーマに関連付けられているテーブルまたはビューがあるかどうかを示すブーリアン・フラグが表示されます。
2. このリストからスキーマを選択すると、**[スキーマ]** ボックスに表示されます。上記と同様に、ドロップダウン・リストでは、テーブル、システム・テーブル、ビュー、プロシージャ、またはスキーマに属するすべてを選択できます。[テーブル]

または [すべて] を選択し、[テーブル] フォルダを開いて、このスキーマのテーブルのリストを開きます。テーブルがない場合は、フォルダを開くと空白ページが表示されます([テーブル] または [すべて] を選択しない場合は、[テーブル] フォルダを開くとネームスペース全体のテーブルがリストされます)。

3. 一覧表示されているテーブルのいずれかを選択します。これにより、テーブルの **[カタログの詳細]** が表示されます。
   - すべてのインデックスを再構築する場合 : **[アクション]** ドロップダウン・リストをクリックし、**[テーブルのインデックスを再構築]** を選択します。
   - 1 つのインデックスを再構築する場合 : **[インデックス]** ボタンをクリックして、既存のインデックスを表示します。リストの各インデックスに、**[インデックス再構築]** のオプションがあります。

注意　他のユーザがテーブルのデータにアクセスしている間は、この方法でインデックスを再構築しないでください。アクティブ・システム上でインデックスを再構築するには、BUILD INDEX を使用するか、プログラムによる方法を使用します。

2.8.3 プログラミングによるインデックスの構築

インデックスを作成するテーブルに **%Persistent** クラスによって用意されている %BuildIndices() メソッドと %BuildIndicesAsync() メソッドを使用することもできます。これらのメソッドは、InterSystems IRIS の既定のストレージ構造を使用しているクラス向けにのみ提供されています。これらのメソッドを呼び出すには、指定のクラスにインデックス定義を 1 つ以上追加してコンパイルしておく必要があります。メソッドの詳細は、以下のリンクからクラスリファレンスのページを参照してください。

- %Library.Persistent.%BuildIndices() : %BuildIndices() はバックグラウンド・プロセスとして実行されますが、呼び出し元に制御が戻るには、%BuildIndices() が完了するまで待機する必要があります。
- %Library.Persistent.%BuildIndicesAsync() : %BuildIndicesAsync() は %BuildIndices() をバックグラウンド・プロセスとして開始し、すぐに呼び出し元に制御が戻ります。%BuildIndicesAsync() の最初の引数は、queueToken 出力引数です。その他の引数は、%BuildIndices() と同じです。

2.9 インデックスの検証

以下のメソッドのいずれかを使用して、インデックスを検証できます。

- $SYSTEM.OBJ.ValidateIndices() は、テーブルのインデックスを検証するほか、そのテーブルのコレクション子テーブル内のあらゆるインデックスも検証します。
- %Library.Storage.%ValidateIndices() は、テーブルのインデックスを検証します。コレクション子テーブルのインデックスは、別個の %ValidateIndices() 呼び出しで検証する必要があります。

両方のメソッドは、指定されたテーブルの 1 つまたは複数のインデックスのデータ整合性を確認するほか、必要に応じて、検出されたインデックス整合性の問題を修正します。インデックス検証は、以下の 2 つの手順で実行されます。

1. テーブル (クラス) 内のすべての行 (オブジェクト) のインデックス・エンティティが適切に定義されていることを確認します。
2. 各インデックスを詳しく調べて、インデックスが付いたすべてのエントリに対して、テーブル (クラス) 内に値および一致するエントリがあることを確認します。

いずれかのメソッドで不一致が見つかった場合は、オプションでそのメソッドでインデックスの構造またはコンテンツを修正できます。標準インデックス、ビットマップ・インデックス、ビットマップ・エクステント・インデックス、およびビットスライス・

インデックスを検証し、必要に応じて修正できます。既定では、どちらのメソッドもインデックスを検証しますが、修正はしません。

SetMapSelectability() を使用して、%ValidateIndices() 引数に autoCorrect=1 と lockOption>0 の両方を指定している場合にのみ、%ValidateIndices() を使用して READ アクティブ・システム上と WRITE アクティブ・システム上のインデックスを修正できます。%ValidateIndices() は処理が著しく遅いので、アクティブ・システム上でインデックスを構築するには %BuildIndices() の使用をお勧めします。

一般に、%ValidateIndices() はターミナルから実行します。出力は現在のデバイスに表示されます。このメソッドは、指定したインデックス名の %List、または指定したテーブル (クラス) に定義したすべてのインデックスに適用できます。これは、指定されたクラスに由来するインデックスにのみ機能します。インデックスがスーパークラスに由来している場合は、そのスーパークラスで %ValidateIndices() を呼び出すことで、そのインデックスを検証できます。これは READONLY クラスではサポートされていません。

2.9.1 シャード・クラスのインデックスの検証

%ValidateIndices() は、シャード・クラスおよびシャード・マスタ・クラス・テーブル (Sharded=1) でサポートされます。%ValidateIndices は、クラス・メソッドとして直接呼び出すことも、シャード・マスタ・クラスで $SYSTEM.OBJ.ValidateIndices から呼び出すこともできます。これにより、各シャードのシャード・ローカル・クラスでインデックス検証が実行され、シャード・マスタの呼び出し元に結果が返されます。シャード・クラスで %ValidateIndices() を使用すると、詳細フラグは強制的に 0 に設定されます。現在のデバイスへの出力はありません。検出/修正された問題がある場合、参照渡しの errors() 配列で返されます。

2.10 クエリ処理でのインデックスの使用

インデックスは、頻繁に要求されるデータのサブセットをソートして管理することで、クエリを最適化するメカニズムを提供します。どのフィールドにインデックスを定義するかを決定する際にも注意が必要です。インデックスが少なすぎる場合やインデックスに誤りがある場合は主要なクエリの速度が大きく低下します。また、インデックスが多すぎる場合は、INSERT や UPDATE のパフォーマンスが低下します (インデックス値の設定や更新が必要になるため)。

2.10.1 インデックスの対象

インデックスの追加によってクエリのパフォーマンスが向上するかどうかを判断するには、管理ポータルの SQL インタフェースからクエリを実行して、[パフォーマンス] のグローバル参照の数をメモします。インデックスを追加してからクエリを再度実行して、グローバル参照の数をメモします。インデックスが有用な場合は、グローバル参照の数が少なくなります。インデックスの使用を抑止するには、WHERE 節または ON 節の条件の前に %NOINDEX キーワードを使用します。

JOIN で指定したフィールド (プロパティ) のインデックスを作成します。LEFT OUTER JOIN では、左のテーブルから開始し、その後、右のテーブルを調査します。そのため、右のテーブルのフィールドからインデックスを作成する必要があります。以下の例では、T2.f2 のインデックスを作成します。

```
FROM Table1 AS T1 LEFT OUTER JOIN Table2 AS T2 ON T1.f1 = T2.f2
```

INNER JOIN では、両方の ON 節のフィールドに対するインデックスが必要です。

プラン表示を実行して、最初のマップをたどります。クエリ・プランの最初の箇条項目が "Read master map" の場合や、最初の箇条項目が "Read master map" のモジュールをクエリ・プランが呼び出す場合は、クエリの最初のマップはインデックス・マップではなくマスタ・マップになります。マスタ・マップは、データに対するインデックスではなくデータ自体を読み取るため、ほとんどの場合、これは非効率的なクエリ・プランになります。テーブルが比較的小さなものでない限り、このクエリを再実行したときにクエリ・プランの最初のマップに [インデックス・マップの読み取り] と表示されるようにインデックスを作成する必要があります。

WHERE 節の等値条件で指定されるフィールドのインデックスを作成します。

WHERE 節の範囲条件で指定されるフィールドや GROUP BY 節および ORDER BY 節で指定されるフィールドのインデックスを作成することが必要になる場合もあります。

範囲条件に基づくインデックスでは、クエリの速度が遅くなる場合があります。大部分の行が指定された範囲条件を満たす場合にこのようになる可能性があります。例えば、レコードの大部分が前の日付を持つデータベースでクエリ節 `WHERE Date < CURRENT_DATE` を使用する場合に、Date にインデックスを作成すると、クエリの速度が大幅に低下します。この原因は、クエリ・オプティマイザは、範囲条件で返される行は比較的少ないと仮定し、この状況に合わせて最適化するからです。これが発生しているかどうかは、範囲条件の前に %NOINDEX を配置してクエリを再度実行することで判断できます。

インデックスが作成されているフィールドを使用して比較を実行する場合、その比較に指定するフィールドの照合タイプは、対応するインデックスの照合タイプと同じにする必要があります。例えば、SELECT の WHERE 節または JOIN の ON 節の Name フィールドは、その Name フィールドに定義されたインデックスと同じ照合にする必要があります。フィールドの照合とインデックスの照合に不一致があると、インデックスの効果が低下するか、インデックスがまったく使用されなくなります。詳細は、このドキュメントの“インデックスの定義と構築”の章の ″インデックス照合″ を参照してください。

インデックスの作成方法および使用可能なインデックスのタイプとオプションの詳細は、″InterSystems SQL リファレンス″ の ″CREATE INDEX″ コマンドと、このドキュメントの“インデックスの定義と構築”の章を参照してください。

2.10.2 インデックス構成オプション

以下のシステム全体の構成メソッドは、クエリでインデックスの使用を最適化するために使用できます。

- 主キーを IDKey インデックスとして使用するには、$SYSTEM.SQL.Util.SetOption() メソッドを `SET status=$SYSTEM.SQL.Util.SetOption("DDLPKeyNotIDKey",0,.oldval)` のように設定します。既定値は 1 です。
- SELECT DISTINCT クエリでインデックスを使用するには、$SYSTEM.SQL.Util.SetOption() メソッドを `SET status=$SYSTEM.SQL.Util.SetOption("FastDistinct",1,.oldval)` のように設定します。既定値は 1 です。

詳細は、″システム管理ガイド″ にリストされている ″SQL およびオブジェクトの設定ページ″ を参照してください。

2.10.3 %ALLINDEX、%IGNOREINDEX、%NOINDEX の使用法

FROM 節では、%ALLINDEX と %IGNOREINDEX の optimize-option キーワードをヒントとしてサポートしています。これらの optimize-option キーワードは、クエリ内のすべてのインデックスの使用を制御します。

条件レベル・ヒント %NOINDEX を使用すると、特定の条件のインデックスの使用に対する例外を指定できます。%NOINDEX ヒントは、インデックスを使用しない各条件の前に配置します。例えば、`WHERE %NOINDEX hiredate < ?` のようにします。通常、この方法は条件によって圧倒的多数のデータが選択される（または選択されない）場合に使用します。「より小さい（<）」または「より大きい（>）」条件では、通常、条件レベル・ヒント %NOINDEX を使用すると効果的です。等価条件では、条件レベル・ヒント %NOINDEX を使用しても効果はありません。結合条件では、%NOINDEX は、ON 節の結合でサポートされます。

%NOINDEX キーワードを使用すると、FROM 節で確立されたインデックス最適化をオーバーライドできます。以下の例では、%ALLINDEX 最適化のキーワードが E.Age 条件を除くすべての条件テストに適用されます。

SQL

```
SELECT P.Name,P.Age,E.Name,E.Age
FROM %ALLINDEX Sample.Person AS P LEFT OUTER JOIN Sample.Employee AS E
     ON P.Name=E.Name
WHERE P.Age > 21 AND %NOINDEX E.Age < 65
```

2.11 インデックス使用状況の分析

SQL クエリ・キャッシュで定義したインデックスの使用状況分析に使用できるツールが 2 つあります。

- 管理ポータルの [インデックス分析] SQL パフォーマンス・ツール。
- %SYS.PTools.UtilSQLAnalysis のメソッド indexUsage()、tableScans()、tempIndices()、joinIndices()、および outlierIndices()。

2.11.1 インデックス分析

以下のいずれかを使用して、管理ポータルから SQL クエリのインデックス使用を分析できます。

- **[システムエクスプローラ]**、**[ツール]**、**[SQL パフォーマンス・ツール]**、**[インデックス分析]** の順に選択します。
- **[システムエクスプローラ]**、**[SQL]** の順に選択し、**[ツール]** ドロップダウン・メニューから **[インデックス分析]** を選択します。

[インデックス分析] には、現在のネームスペースの SQL 文カウントが表示され、5 つのインデックス分析レポート・オプションが用意されています。

2.11.1.1 SQL 文カウント

[SQL インデックス・アナライザ] の上部には、ネームスペースに含まれるすべての SQL 文をカウントするオプションがあります。**[SQL 文の収集]** ボタンをクリックします。**[SQL インデックス・アナライザ]** では、カウントの処理中には “SQL 文の収集中 ....” と表示され、カウントが完了すると “完了!” と表示されます。SQL 文は、クエリ・キャッシュのカウント、クラス・メソッドのカウント、およびクラス・クエリのカウントという 3 つのカテゴリでカウントされます。これらのカウントは、現在のネームスペース全体に対するものであり、**[スキーマ選択]** オプションの影響を受けません。対応するメソッドは、%SYS.PTools.UtilSQLAnalysis クラスの getSQLStmts() です。

[ステートメントを削除] ボタンを使用して、現在のネームスペース内の収集されたすべての文を削除できます。このボタンにより、clearSQLStatements() メソッドが呼び出されます。

2.11.1.2 レポート・オプション

現在のネームスペース内の選択済みスキーマのクエリ・キャッシュのレポートを確認するか、(スキーマが選択されていない場合) 現在のネームスペース内のすべてのクエリ・キャッシュのレポートを確認できます。システム・クラス・クエリ、INSERT 文、IDKEY インデックスを、この分析から除外することも、この分析の対象にすることもできます。スキーマ選択およびスキップ・オプションのチェック・ボックスは、ユーザによってカスタマイズされます。

インデックス分析レポート・オプションを以下に示します。

- **インデックス使用** : このオプションは、現在のネームスペース内のすべてのクエリ・キャッシュを取得し、それぞれにプラン表示を生成します。さらに、各インデックスが各クエリで使用された回数とネームスペース内のすべてのクエリによる各インデックスの合計使用量のカウントを保持します。これを使用して、使用されていないインデックスを明らかにし、削除したり役立つように変更することができます。結果セットは、最少使用のインデックスから最多使用のインデックスに向けて順序付けされます。
- **テーブル・スキャンを行うクエリ** : このオプションは、テーブル・スキャンを実行する、現在のネームスペース内のすべてのクエリを識別します。可能な限り、テーブル・スキャンは回避する必要があります。テーブル・スキャンを必ず防ぐことはできませんが、テーブルで多数のテーブル・スキャンが発生する場合は、テーブルに定義されているインデックスを確認する必要があります。多くの場合、テーブル・スキャンのリストと一時インデックスのリストは重複しています。一方を固定して、もう一方を削除します。結果セットでは、最大ブロック・カウントから最小ブロック・カウントの順にテーブルがリストされます。[プラン表示] リンクにより、[文テキストとクエリ・プラン] が表示されます。

- **一時インデックスを使用するクエリ**：このオプションは、SQL を解決するための一時インデックスを構築する、現在のネームスペース内のすべてのクエリを識別します。場合によっては、一時インデックスの使用が役立ち、パフォーマンスが向上します。例えば、InterSystems IRIS でマスタ・マップを順序どおりに読み込むために使用できる、範囲条件に基づく小さなインデックスを構築します。一時インデックスは別のインデックスの単純なサブセットとして非常に効率が良い場合があります。また、一時インデックスがパフォーマンスの低下につながることもあります。例えば、条件を含むプロパティに対する一時インデックスを構築するために、マスタ・マップをスキャンする場合です。この状況は、必要なインデックスが見つからないことを意味します。インデックスは、一時インデックスと一致するクラスに追加する必要があります。結果セットでは、最大ブロック・カウントから最小ブロック・カウントの順にテーブルがリストされます。[プラン表示] リンクにより、[文テキストとクエリ・プラン] が表示されます。
- **JOIN のインデックスがないクエリ**：このオプションは、現在のネームスペース内で結合が含まれるすべてのクエリを検証し、その結合をサポートするために定義されたインデックスがあるかどうかを確認します。結合のサポートに利用できるインデックスはランク付けされます。このランクは、0 (インデックスが存在しない) から 4 (結合を完全にサポートするインデックス) までの範囲になります。外部結合では、1 方向のインデックスが必要です。内部結合では、双方向のインデックスが必要です。既定では、結果セットには、JoinIndexFlag < 4 がある行のみが含まれます。JoinIndexFlag=4 は、結合を完全にサポートするインデックスがあることを意味します。
- **外れ値インデックスを使用したクエリ**：このオプションは、現在のネームスペース内で異常値が含まれるすべてのクエリを識別し、その異常値をサポートするために定義されたインデックスがあるかどうかを確認します。異常値のサポートに利用できるインデックスはランク付けされます。このランクは、0 (インデックスが存在しない) から 4 (異常値を完全にサポートするインデックス) までの範囲になります。既定では、結果セットには、OutlierIndexFlag < 4 がある行のみが含まれます。OutlierIndexFlag=4 は、異常値を完全にサポートするインデックスがあることを意味します。

これらのオプションのいずれかを選択すると、自動的に操作が実行されて、結果が表示されます。オプションを最初に選択したとき、または対応するメソッドを最初に呼び出したときに、結果データが生成されます。そのオプションを再度選択しても、そのメソッドを再度呼び出しても、InterSystems IRIS は同じ結果を再表示します。新しい結果データを生成するには、**[SQL 文の収集]** ボタンを使用して、インデックス・アナライザの結果テーブルを再初期化する必要があります。**[次で始まるすべてのシステムクラスとルーチンをスキップ]** または **[ループロジックがないため、INSERT ステートメントをスキップ]** チェック・ボックス・オプションを変更しても、インデックス・アナライザの結果テーブルを再初期化できます。%SYS.PTools.UtilSQLAnalysis のメソッドの新しい結果データを生成するには、getSQLStmts() を呼び出して、インデックス・アナライザの結果テーブルを再初期化する必要があります。

2.12 インデックスのリスト表示

INFORMATION.SCHEMA.INDEXES 永続クラスは、現在のネームスペース内のすべての列インデックスに関する情報を表示します。これは、インデックス付けされた列ごとに 1 つのレコードを返します。これは、インデックスの名前や、インデックスのマップ先のテーブル名、列名など、さまざまなインデックス・プロパティを提供します。各列レコードには、インデックス・マップにおける列の順序位置も含まれます。この値は、インデックスが複数の列にマップされない限り 1 です。また、ブーリアン・プロパティ PRIMARYKEY および NONUNIQUE も提供します (0= インデックス値は一意である必要があります)。

以下の例では、現在のネームスペース内のすべての非システム・インデックスについて、インデックスに属する列ごとに 1 行が返されます。

SQL

```
SELECT Index_Name,Table_Schema,Table_Name,Column_Name,Ordinal_Position,
Primary_Key,Non_Unique
FROM INFORMATION_SCHEMA.INDEXES WHERE NOT Table_Schema %STARTSWITH '%'
```

管理ポータルの SQL インタフェースの [カタログの詳細] の **[マップ/インデックス]** オプションを使用して、選択したテーブルのインデックスをリスト表示できます。このオプションでは、インデックスごとに 1 行と、INFORMATION.SCHEMA.INDEXES で提供されないインデックス情報が表示されます。

2.13 メソッドのオープン、存在確認、および削除

InterSystems IRIS インデックス作成機能は、以下の操作をサポートします。

- インデックス・キーによるインスタンスのオープン
- インスタンスが存在するかどうかの確認
- インスタンスの削除

2.13.1 インデックス・キーによるインスタンスのオープン

ID キー、主キー、または一意のインデックスの場合、indexnameOpen() メソッド (indexname はインデックスの名前) を使用することによって、インデックスのプロパティ値が指定値と一致するオブジェクトを開くことができます。このメソッドには、インデックス内の各プロパティに対応する 1 つの引数があるため、メソッドには 3 つ以上の引数があります。

- 最初の引数は、それぞれインデックス内のプロパティに対応します。
- 最後から 2 番目の引数は、オブジェクトを開くときに使用される並行処理値を指定します (使用可能な並行処理の設定は、"クラスの定義と使用" の "オブジェクト同時処理" に示されています)。
- 最後の引数は、メソッドがインスタンスを開くことに失敗した場合に、%Status コードを受け取ることができます。

このメソッドは、一致するインスタンスを見つけると OREF を返します。

例えば、クラスに以下のインデックス定義が含まれているとします。

Class Member

```
Index SSNKey On SSN [ Unique ];
```

そして、参照されたオブジェクトがディスクに保存され、一意の ID 値を持つ場合、以下のようにメソッドを呼び出すことができます。

ObjectScript

```
SET person = ##class(Sample.Person).SSNKeyOpen("111-22-3333",2,.sc)
```

正常に完了すると、このメソッドによって、SSN プロパティの値が 111-22-3333 である Sample.Person のインスタンスの OREF に、person の値が設定されます。

メソッドへの 2 番目の引数は並行処理値を指定し、上の例では 2 (共有) です。3 番目の引数は、オプションの %Status コードを保持します。指定された値と一致するオブジェクトをメソッドが見つけられなかった場合、エラー・メッセージがステータス・パラメータ sc に書き込まれます。

このメソッドは、%Compiler.Type.Index.Open() メソッドとして実装されます。このメソッドは、%Persistent.Open() メソッドおよび %Persistent.OpenId() メソッドに似ています。異なるのは、このメソッドが、OID 引数や ID 引数ではなくインデックス定義内のプロパティを使用する点です。

2.13.2 インスタンスが存在するかどうかの確認

indexnameExists() メソッド (indexname はインデックスの名前) は、メソッドの引数で指定されているインデックス・プロパティ値を持つインスタンスが存在するかどうかを確認します。このメソッドには、インデックス内の各プロパティに対応する 1 つの引数があり、その最後のオプションの引数は、オブジェクトの ID が指定値と一致する場合に、その ID を受け取ることができます。このメソッドは、ブーリアン値を返し、成功 (1) または失敗 (0) を示します。このメソッドは、%Compiler.Type.Index.Exists() メソッドとして実装されます。

例えば、クラスに以下のインデックス定義が含まれているとします。

Class Member

```
Index SSNKey On SSN [ Unique ];
```

そして、参照されたオブジェクトがディスクに保存され、一意の ID 値を持つ場合、以下のようにメソッドを呼び出すことができます。

ObjectScript

```
SET success = ##class(Sample.Person).SSNKeyExists("111-22-3333",.id)
```

正常に完了すると、success が 1 になり、id には、検出されたオブジェクトと一致する ID が含まれます。

このメソッドは、以下を除くすべてのインデックスの値を返します。

- ビットマップ・インデックス、またはビットマップ・エクステント・インデックス。
- インデックスに (ELEMENTS) 式または (KEYS) 式が含まれている場合。そのようなインデックスの詳細は、“コレクションのインデックス作成”のセクションを参照してください。

2.13.3 インスタンスの削除

indexnameDelete() メソッド (indexname はインデックスの名前) は、Unique、PrimaryKey、IdKey の各インデックスに使用します。このメソッドは、指定されたキーのプロパティ/列の値に一致するインスタンスを削除します。オプションの引数が 1 つあり、これを指定して操作の並行処理を指定できます。このメソッドは %Status コードを返します。%Compiler.Type.Index.Delete() メソッドとして実装されます。

3

SQL テーブルのストレージ・レイアウトの選択

InterSystems IRIS® では、ここに示すようなリレーショナル・テーブルは論理的な抽象概念です。データの基礎的な物理ストレージ・レイアウトを反映したものではありません。

OrderID	OrderDate	Customer	Priority	Status	TotalAmount

InterSystems IRIS の低レベル・データ構造であるグローバルに固有の柔軟性を使用して、行、列、またはその両方のどれにデータを保存するかを指定できます。データのサイズおよびクエリとトランザクションの性質によっては、適切なストレージ・レイアウトを選択することにより、クエリのパフォーマンスやトランザクションのスループットが大幅に向上することがあります。

ストレージ形式の選択によって、クエリの作成方法と INSERT などの他の SQL 文の作成方法が変わることはありません。ストレージ形式は、シャーディングなどの他のストレージ・オプションと補完的な関係にあります。大規模なテーブルでは、シャーディングによってクエリのパフォーマンスをさらに改善できる可能性があります。どのストレージ・レイアウトでも、テーブルにインデックスを定義してパフォーマンス上の利点をさらに得ることができます。詳細は、"ストレージ・レイアウト上のインデックス" を参照してください。

行ベースのストレージ形式と列ベース (列指向) のストレージ形式を以下の表に示します。

行ストレージ (既定)	列指向ストレージ
プライマリ・データは 1 つのグローバルに格納されます。データの各行は、さまざまなデータ型の要素に対応したリスト・エンコーディングを使用して、別々のグローバル添え字で格納されます。一般的に、トランザクションは個別の行を変更し、行単位で格納されているデータを効率的に処理しますが、分析クエリの処理は遅くなることがあります。	プライマリ・データは、列ごとに 1 つのグローバルに格納されます。64,000 個のデータ要素から成るシーケンスがそれぞれ別々のグローバル添え字で格納されます。データ型が同じ要素の格納に最適化されたベクトル・エンコーディングを使用してデータがエンコードされます。一般的に分析クエリの実行は高速ですが、トランザクションの処理は遅くなることがあります。
	
以下の場合に行ストレージを使用します。 · トランザクションのデータをリアルタイムで処理するオンライン・トランザクション処理 (OLTP)。 · データの頻繁な挿入、更新、削除。 · 行全体を一括して選択し、迅速に実体化するクエリ。	以下の場合に列指向ストレージを使用します。 · 特定の列のデータをフィルタ処理して集計し、分析クエリを実行するオンライン分析処理 (OLAP)。 · 更新、挿入、削除の頻度が低いデータ、または LOAD DATA の使用などによってこれらの各操作が一括で実行されるデータ。

ストレージ・レイアウトの選択は厳正に科学的な作業ではありません。さまざまなレイアウトを試し、クエリ・テストを何回も実行して最適なレイアウトを探し出す必要がある作業です。ストレージ・レイアウトを行とするか列指向とするかを判断する作業の概要と、各レイアウトの選択事例は、動画 "What Is Columnar Storage?" を参照してください。

注釈　　現在のところ、データに定義したストレージ・レイアウトは、データを再読み込みしない限り、変更できません。

3.1 行ストレージ・レイアウト

InterSystems IRIS では、InterSystems SQL DDL 言語または永続クラスでテーブルを定義するときに、ストレージ・レイアウトは既定で行ストレージになります。

3.1.1 DDL を使用した行ストレージ・テーブルの定義

InterSystems SQL DDL で CREATE TABLE コマンドを使用すると、既定でテーブルが行ストレージ・レイアウトで定義されます。CREATE TABLE には、オプションの WITH STORAGETYPE = ROW 節が用意されていて、列定義の後で指定できますが、省略してもかまいません。以下の 2 つの構文は同等です。

```
CREATE TABLE table (   column type,   column2 type2,   column3 type3)


CREATE TABLE table (   column type,   column2 type2,   column3 type3) WITH STORAGETYPE = ROW
```

以下の CREATE TABLE コマンドでは銀行トランザクションのテーブルが作成されます。このテーブルには、口座番号、トランザクションの日付、トランザクションの説明、トランザクション金額、トランザクション・タイプ (″預金″、″引き出し″、″振り込み″) の各列があります。WITH STORAGETYPE 節を省略しているので、このテーブルは既定の行ストレージになります。

SQL

```
CREATE TABLE Sample.BankTransaction (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2),
  Type VARCHAR(10))
```

3.1.2 永続クラスを使用した行ストレージ・テーブルの定義

DDL で作成するテーブル同様に、永続クラスを使用して作成するテーブルでも、既定で行ストレージ・レイアウトが使用されます。必要に応じて、値 ″row″ または空文字列 (″″) を指定した STORAGEDEFAULT パラメータを定義できますが、どちらも省略してかまいません。以下の各構文は同等です。

```
Parameter STORAGEDEFAULT = "row";


Parameter STORAGEDEFAULT = ""; /* Or can be omitted entirely */
```

次の永続クラスは、前のセクションで作成した DDL 定義のテーブルと同様の行ストレージ・テーブルの定義を示しています。USEEXTENTSET パラメータを指定することで、テーブル・ストレージが、より効率的な一群のグローバルに編成されます。ビットマップ・エクステント・インデックスによって、エクステント・セットにあるすべての ID のインデックスが作成されます。これにより、カウントなどの操作の効率が向上します。DDL コマンドを使用してテーブルを定義すると、InterSystems SQL によってこれらの設定が自動的に適用され、投影された永続クラスに追加されます。詳細は、″永続クラスの作成によるテーブルの定義″ を参照してください。

```
  Class Sample.BankTransaction Extends %Persistent [ DdlAllowed ]
  {
  Parameter USEEXTENTSET = 1;

    Property AccountNumber As %Integer;
    Property TransactionDate As %Date;
    Property Description As %String(MAXLEN = 10);
    Property Amount As %Numeric(SCALE = 2);
    Property Type As %String(VALUELIST = ",Deposit,Withdrawal,Transfer");

    Index BitmapExtent [ Extent, Type = bitmap ];
  }
```

注釈　パラメータ名 STORAGEDEFAULT にある DEFAULT は、このクラスにストレージ定義エントリを生成するときに、このパラメータ値が既定で使用されることを意味します。USEEXTENTSET や DEFAULTGLOBAL など、ストレージに影響する大半のクラス・パラメータと同様に、ストレージを生成するとき、クラスを初めてコンパイルするとき、または新しいプロパティを追加するときにのみ、この値が考慮されます。ストレージの XData ブロックに保存されたストレージ定義がすでに存在するクラスでは、これらのパラメータを変更しても、クラス・エクステントについて格納されているデータも含め、そのストレージには何の影響もありません。

3.1.3 行ストレージの詳細

行ストレージによるテーブルでは、すべてのデータが 1 つのグローバルに格納されます。添え字付きの各グローバルには、テーブルの各列の値から成る 1 行のデータを収めた $LIST 値が格納されます。$LIST データ型は、さまざまなデータ型の要素を格納し、空文字列 (' ') と NULL 値を効率的にエンコードします。このような特徴によって $LIST はデータの行の格納に適した型になっています。各列にはさまざまな型の値があり、NULL 値が存在することもあるからです。

前のセクションで取り上げた `BankTransaction` テーブルに、以下のトランザクション・レコードがあるとします。

SQL

```
SELECT AccountNumber,TransactionDate,Description,Amount,Type FROM Sample.BankTransaction
```

AccountNumber	TransactionDate	Description	Amount	Type
10001234	2022年2月22日	預金口座への預金	40.00	預金
10001234	2022年3月14日	ベンダへの支払い	-20.00	引き出し
10002345	2022年7月30日	当座預金への振り替え	-25.00	振り込み
10002345	2022年8月13日	預金口座への預金	30.00	預金

必要に応じて、管理ポータルで [**システムエクスプローラ**]、[**グローバル**] の順にクリックすることで、グローバル・ストレージ構造を検査できます。続いて、テーブルが置かれたネームスペースで [**SQLテーブル名を表示**] を選択して、テーブルに対応するデータ/マスタ・グローバルを探し出すことができます。詳細は、"グローバルの管理" を参照してください。以下のコードは、InterSystems IRIS の大半の標準テーブルで見られるデータ/マスタ・グローバルの例です。

```
^BankT    = 4
^BankT(1) = $lb(10001234,66162,"Deposit to Savings",40,"Deposit")
^BankT(2) = $lb(10001234,66182,"Payment to Vendor ABC",-20,"Withdrawal")
^BankT(3) = $lb(10002345,66320,"Transfer to Checking",-25,"Transfer")
^BankT(4) = $lb(10002345,66334,"Deposit to Savings",30,"Deposit")
```

- `^BankT` はグローバルの名前です。ここに示した名前は説明を目的としたものにすぎません。DDL を使用して作成したテーブル、または USEEXTENTSET=1 パラメータを指定して永続クラスで作成したテーブルでは、`^EW3K.Cku2.1` のような名前で、より効率的なハッシュ化したグローバルが生成されます。永続クラス・テーブルで USEEXTENTSET=1 を指定していない場合、グローバルの名前は `^TableNameD` の形式になります。SQL テーブルの投影された永続クラスでは、`Storage` クラス・メンバの `<DataLocation>` 要素にグローバルが格納されます。例えば以下のようにします。

  ```
  Storage Default
  {
  ...
  <DataLocation>^BankT</DataLocation>
  ...
  }
  ```

- 最上位グローバル・ノードの値は、テーブルで最大の添え字です。この例では 4 になります。この整数の添え字は行 ID として使用されます。
- 各グローバルの添え字は、1 行にある各列の値を収めた $LIST です。各要素値の順序はストレージ定義で定義され、普通は列の順序と一致しています。この例では、各行の 2 番目の要素は TransactionDate 列に相当し、そのデータは $HOROLOG 形式 (1840 年 12 月 31 日からの経過日数) で格納されています。

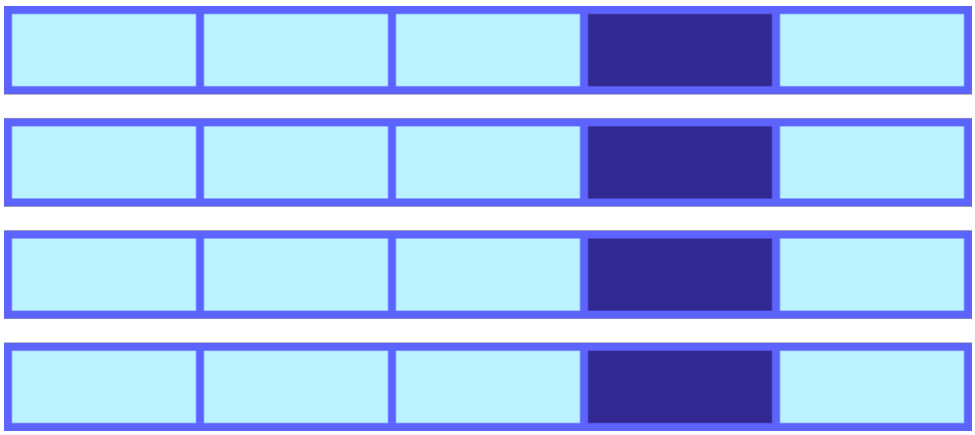
3.1.4 行ストレージによる分析クエリ処理

分析処理では行ストレージが効率面で不利であることを確認するために、BankTransaction テーブルにあるすべてのトランザクション金額の平均サイズを求めるクエリを実行してみます。

SQL

```
SELECT AVG(ABS(Amount)) FROM Sample.BankTransaction
```

このクエリに関連する値は Amount 列のみです。しかし、このクエリでこれらの値にアクセスするには、各行の全体をメモリに読み込む必要があります。InterSystems IRIS で読み込むことができるストレージの最小単位はグローバル・ノードであるからです。



クエリが各行に個別にアクセスしているかどうかを確認するには、クエリ実行プランを分析します。管理ポータルで、**[システムエクスプローラ]**、**[SQL]**、**[プラン表示]** の順に選択します。または、SQL コマンドの EXPLAIN query を使用します。このクエリに "Read master map ... looping on IDKEY" のような文があると、テーブルの行にある各列の値がクエリに関連するかどうかに関係なく、各行が読み込まれます。

行ストレージでテーブルに対する分析クエリのパフォーマンスを改善するには、頻繁に使用するフィールドにインデックスを作成し、WHERE 節でフィルタ処理します。詳細は、"ストレージ・レイアウトのインデックス" を参照してください。

3.1.5 行ストレージによるトランザクション処理

トランザクション処理では行ストレージが効率的であることを確認するために、BankTransaction テーブルに新しい行を挿入してみます。

SQL

```
INSERT INTO Sample.BankTransaction VALUES (10002345,TO_DATE('01 SEP 2022'),'Deposit to
Savings',10.00,'Deposit')
```

既存のどの $LIST グローバルもメモリに読み込むことを必要とせずに、INSERT 操作で新しい $LIST グローバルを作成します。

3.2 列指向ストレージ・レイアウト

InterSystems SQL DDL 言語または永続クラスを使用して、テーブル全体を列指向ストレージを持つものとして定義できます。

3.2.1 DDL を使用した列指向ストレージ・テーブルの定義

InterSystems SQL DDL で列指向ストレージによるテーブルを定義するには、CREATE TABLE で列定義の後に `WITH STORAGETYPE = COLUMNAR` を使用します。

```
CREATE TABLE table (   column type,   column2 type2   column3 type3) WITH STORAGETYPE = COLUMNAR
```

列指向ストレージを使用するよう定義されているテーブルが DDL を使用して作成されている場合、システムでは VARCHAR 列の長さが考慮され、12 文字 (列指向ストレージの現在の文字列長上限) より長い場合はそれらの VARCHAR 列は行指向ストレージに戻されます。長さが 12 文字以下のその他の VARCHAR 列を含め、その他すべての列は引き続き列指向ストレージに格納されるため、テーブルは混合ストレージ・レイアウトになります。すべての列が列指向ストレージに適合する場合、各フィールドは列指向レイアウトで格納され、取り込みおよびクエリ実行のパフォーマンスが最高になり、最も効率的なストレージが提供されます。

このコマンドによって、それまでに実行した口座トランザクションの履歴データを収めた `TransactionHistory` テーブルが作成されます。このテーブルには、"行ストレージ・レイアウト" のセクションで作成した `BankTransaction` テーブルと同じ列があります。この例では、`Description` 列は行指向ストレージを使用して格納され、その他の列は列指向ストレージを使用して格納されます。

```
CREATE TABLE Sample.TransactionHistory (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2),
  Type VARCHAR(10))
WITH STORAGETYPE = COLUMNAR
```

3.2.2 永続クラスを使用した列指向ストレージ・テーブルの定義

永続クラスを使用して列指向ストレージ・テーブルを作成するには、STORAGEDEFAULT パラメータの値を "columnar" に設定します。

```
Parameter STORAGEDEFAULT = "columnar"
```

DDL を介して列指向ストレージを使用するようテーブルを定義する場合とは異なり、このクラスに格納されている文字列が列指向ストレージの内部の文字列長制限を満たしていることの検証は行われません。このため、STORAGEDEFAULT パラメータで設定された列指向ストレージを使用するよう定義されているテーブルに格納される文字列長には特に注意が必要です。上限 (12 文字) より長い一意の文字列を大量に取り込んで格納する際にエラーが発生する可能性があるためです。代わりに、クラス内の文字列型のフィールドの MAXLEN を 12 以下に設定できます。

次の永続クラスは、前のセクションで作成した DDL 定義のテーブルと同様の列指向ストレージ・テーブルを定義しています。"永続クラスを使用した行ストレージ・テーブルの定義" で作成した BankTransaction テーブルと同様に、このテーブルでは USEEXTENTSET パラメータとビットマップ・エクステント・インデックスが定義されます。この設定の詳細は、"永続クラスの作成によるテーブルの定義" を参照してください。

```
Class Sample.TransactionHistory Extends %Persistent [ DdlAllowed, Final ]
{
  Parameter STORAGEDEFAULT = "columnar";
  Parameter USEEXTENTSET = 1;

  Property AccountNumber As %Integer;
  Property TransactionDate As %Date;
  Property Description As %String(MAXLEN = 10);
  Property Amount As %Numeric(SCALE = 2);
  Property Type As %String(VALUELIST = "-Deposit-Withdrawal-Transfer");

  Index BitmapExtent [ Extent, Type = bitmap ];
}
```

任意のテーブルを列指向として宣言できます。ただし、既定のストレージ・レイアウトとして列指向を使用するテーブルでは、Final クラス・キーワードまたは NoExtent クラス・キーワードを指定し、その直下のすべてのサブクラスを明示的に Final として定義する必要があります。

すでに説明したように、新しいテーブルまたは列のストレージ定義を生成するときに InterSystems IRIS でどのストレージ・タイプを使用するかを STORAGEDEFAULT パラメータで指定します。ただし、列のタイプによっては、列指向ストレージでの使用に最適化されたベクトル・データ型へ適切にエンコードされない列があります。このようにエンコードされない列として、シリアル、配列、リストなどがあります。このようなデータ型を列指向ストレージで使おうとするとコンパイル・エラーが発生します。以下の場合は、InterSystems IRIS によって自動的に行ストレージに戻されます。

- 列タイプに列指向ストレージとの互換性がない場合。この互換性がないタイプの例として、ストリーム、配列、リストがあります。
- 列指向ストレージに適切に収まらない列タイプの場合。この例として、13 文字以上の文字列があります。この場合は、列に STORAGETYPE = COLUMNAR 節を設定して、ストレージ・タイプをオーバーライドできます。詳細は、"混合ストレージ・レイアウト" のセクションを参照してください。13 文字以上の文字列に列指向ストレージを使用する必要がある例として、文字列のカーディナリティが低い場合が挙げられます。これは、対応できる文字列値がきわめて限られていることを意味します。すべての文字列が異なる場合は、行ストレージの使用が適切です。

3.2.3 列指向ストレージの詳細

3.2.3.1 グローバル構造

列指向ストレージによるテーブルでは、各列のデータセットが別々のグローバルに格納されます。列グローバルのそれぞれでは、行の値要素のデータ型がすべて同じで、64,000 行ごとに別々の添え字に "チャンク化" されます。これは、$BIT 値の格納方法と同様です。例えば、100,000 行で構成するテーブルでは、列グローバルごとに 2 つの添え字があ

ります。1 番目の添え字には先頭の 64,000 行の値が格納され、2 番目の添え字には、残りの 36,000 行の値が格納されます。InterSystems IRIS では、特殊なベクトル・エンコーディングを使用して、同じデータ型のデータが効率的に格納されます。

前のセクションで定義した TransactionHistory テーブルに以下のレコードがあるとします。

SQL

```
SELECT AccountNumber,TransactionDate,Description,Amount,Type FROM Sample.TransactionHistory
```

AccountNumber	TransactionDate	Description	Amount	Type
10001234	2022年2月22日	預金口座への預金	40.00	預金
10001234	2022年3月14日	ベンダへの支払い	-20.00	引き出し
10002345	2022年7月30日	当座預金への振り替え	-25.00	振り込み
10002345	2022年8月13日	預金口座への預金	30.00	預金

必要に応じて、管理ポータルで [**システムエクスプローラ**]、[**グローバル**] の順にクリックすることで、グローバル・ストレージ構造を検査できます。このテーブルが置かれたネームスペースで [**SQLテーブル名を表示**] を選択して、テーブルに対応するグローバルを探し出すことができます。詳細は、"グローバルの管理" を参照してください。

データ/マスタ・グローバルは、各行の添え字を格納し、行の操作に関与するデータの参照に使用されます。データは列ごとに別々のグローバルに格納されるので、各添え字の行は空です。以下のコードはデータ/マスタ・グローバルの例です。

```
^THist = 4
```

^THist はグローバルの名前です。ここに示した名前は説明を目的としたものにすぎません。DDL を使用して作成したテーブル、または USEEXTENTSET=1 パラメータを指定して永続クラスで作成したテーブルでは、^EW3K.B3vA.1 のような名前で、より効率的なハッシュ化したグローバルが生成されます。永続クラス・テーブルで USEEXTENTSET=1 を指定していない場合、グローバルの名前は ^*TableName*D の形式になります。SQL テーブルの投影された永続クラスでは、Storage クラス・メンバの <DataLocation> 要素にグローバルが格納されます。例えば以下のようにします。

```
Storage Default
{
...
<DataLocation>^THist</DataLocation>
...
}
```

このテーブルには、各列に 1 つずつ、合計で 5 つの追加のグローバルがあり、その名前は ^THist.V1、^THist.V2 のような形式になっています。このグローバルのそれぞれには、1 つの列の各行の値がベクトル・エンコーディングで格納されています。これは、同じ型の値で機能して、スパース・データを効率的にエンコードするように設計された内部データ型です実際のエンコーディングは内部処理ですが、[**グローバル**] ページと ZWRITE などの情報コマンドによって、判読できる形式で以下の情報が提供されます。

- データの型
- 列にある、NULL ではない要素の数
- ベクトルの長さ

このテーブルの行数は 64,000 未満なので、各列グローバルにある添え字は 1 つのみです。ここに示した各グローバルのデータは、読みやすくするために切り詰めています。

```
^THist.V1(1) = {"type":"integer", "count":4, "length":5, "vector":[,10001234,...]}
^THist.V2(1) = {"type":"integer", "count":4, "length":5, "vector":[,66162,...]}
^THist.V3(1) = {"type":"string", "count":4, "length":5, "vector":[,"Deposit to Savings",...]}
^THist.V4(1) = {"type":"decimal", "count":4, "length":5, "vector":[,40,...]}
^THist.V5(1) = {"type":"string", "count":4, "length":5, "vector":[,"Deposit",...]}
```

この列グローバルを 200,000 行のテーブルで作成すると、要素が 64,000 個の 3 つの添え字と要素が 8,000 個の 1 つの添え字にデータが分散します。列には NULL 値が存在するので、要素の数はグローバルの長さよりも小さくなります。

```
^MyCol.V1(1) = {"type":"integer", "count":63867, "length":64000, "vector":[,1,1,1,,...]}
^MyCol.V1(2) = {"type":"integer", "count":63880, "length":64000, "vector":[1,1,1,,1,...]}
^MyCol.V1(3) = {"type":"integer", "count":63937, "length":64000, "vector":[1,1,1,2,2,...]}
^MyCol.V1(4) = {"type":"integer", "count":7906, "length":8000, "vector":[1,1,1,,2,...]}
```

3.2.3.2 列指向ストレージによる文字列照合

既定で列指向ストレージを使用するテーブルに文字列型のフィールドを定義すると、別途指定しない限り、照合は EXACT として定義されます。フィールドの MAXLEN が 12 文字を超えても文字列型のフィールドには EXACT 照合が使用されるので、そのフィールドは行ストレージに戻されます。

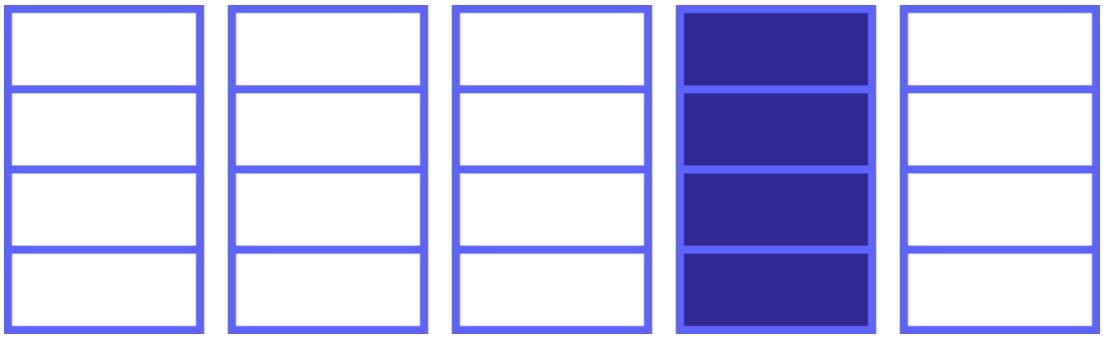
3.2.4 列指向ストレージによる分析クエリ処理

分析処理では列指向ストレージが効率的であることを確認するために、`TransactionHistory` テーブルにあるすべてのトランザクション金額の平均サイズを求めるクエリを実行してみます。

SQL

```
SELECT AVG(ABS(Amount)) FROM Sample.TransactionHistory
```

列指向ストレージでは、このクエリを実行すると `Amount` 列のグローバルのみがメモリにロードされ、その列のデータを使用して平均値が計算されます。他の列のデータはメモリにロードされないので、データが行単位で格納されている場合よりも効率的なクエリになります。また、最適化されたベクトル・エンコーディングは、個々の値に対してではなく、ベクトル全体に対して一度で効率的に実行される専用のベクトル化演算を備えています。例えば、すべての要素の和をベクトル内部で計算する方法は、要素を 1 つずつ加算する方法よりもはるかに高速です。特に、行データを保持している $list から値をその都度抽出する必要がある場合に顕著です。このようなベクトル化演算の多くは、低レベルの SIMD (単一命令、多重データ) チップセット最適化を活用しています。



クエリ実行プランを分析することで、列指向ストレージの効率が活用されるクエリであるかどうかを確認できます。管理ポータルで、**[システムエクスプローラ]**、**[SQL]**、**[プラン表示]** の順に選択します。または、SQL コマンドの EXPLAIN query を使用します。"列指向インデックスの読み取り"、"ベクトル演算の適用"、"列指向データ/インデックス・マップ" などの文がクエリ・プランにあれば、そのクエリは列指向グローバルのデータにアクセスします。

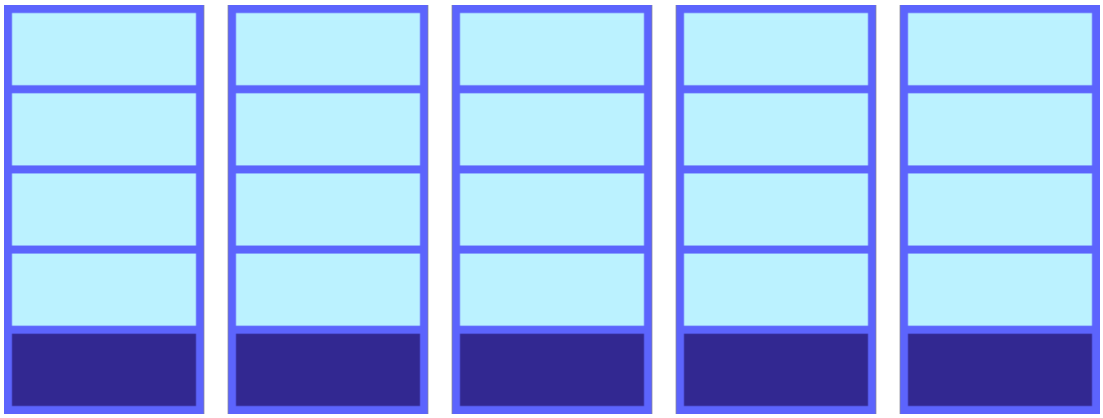
3.2.5 列指向ストレージによるトランザクション処理

トランザクション処理では列指向ストレージが効率面で不利であることを確認するために、TransactionHistory テーブルに新しい行を挿入してみます。

SQL

```
INSERT INTO Sample.TransactionHistory VALUES (10002345,TO_DATE('01 SEP 2022'),'Deposit to
Savings',10.00,'Deposit')
```

すべての列グローバルにわたって行データが分散しているので、挿入を実行するには、INSERT 操作でこれらのグローバルごとに最後のチャンクをメモリにロードする必要があります。



列指向ストレージ・レイアウトへのデータ挿入はメモリ効率の点で劣ることから、挿入の実行頻度を低くするか、LOAD DATA コマンドの使用などによって一括で挿入します。InterSystems IRIS には、列指向テーブルに対する INSERT をメモリにバッファリングしたうえでディスクにチャンクを書き込む最適化が用意されています。

3.3 混合ストレージ・レイアウト

さらに高い柔軟性を目指し、行ストレージと列指向ストレージが混在するテーブルを定義できます。このようなテーブルでは、テーブルの全体的なストレージ・タイプを指定したうえで、特定のフィールドを、それとは異なるストレージ・タイプを持つものとして設定します。

頻繁に集計するフィールドのように分析クエリの対象とする列が少ないトランザクションベースのテーブルで混合ストレージが効果的です。大部分のテーブル・データを行単位で格納したうえで、頻繁に集計する列を列指向形式で格納できます。フィルタ処理やグループ化などが適用されずに、通常は行レベルのクエリ結果にそのまま返される列も行ストレージに適していることがあります。これにより、クエリ結果に追加する前の列を実体化するコストを削減できます。この方法で 1 つのテーブルに対してトランザクション・クエリと分析クエリを実行できます。

3.3.1 DDL を使用した混合ストレージの定義

InterSystems SQL DDL で混合ストレージを持つテーブルを定義するには、CREATE TABLE コマンドで個々の列に WITH STORAGETYPE = ROW 節または WITH STORAGETYPE = COLUMNAR 節を指定します。

以下の構文で作成されるテーブルは、既定である行ベースのストレージ・レイアウトになりますが、3 番目の列では列指向ストレージを使用しています。

```
CREATE TABLE table (   column type,   column2 type2   column3 type3 WITH STORAGETYPE = COLUMNAR)
```

以下の構文で作成されるテーブルは列ベースのストレージ・レイアウトになりますが、3 番目の列は行レイアウトで格納されます。

```
CREATE TABLE table (   column type,   column2 type2   column3 type3 WITH STORAGETYPE = ROW) WITH
STORAGETYPE = COLUMNAR
```

以下の CREATE TABLE コマンドで作成する BankTransaction テーブルでは、Amount 列のデータのみに列指向ストレージが使用され、他のすべてのデータは行レイアウトで格納されます。

```
CREATE TABLE Sample.BankTransaction (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2) WITH STORAGETYPE = COLUMNAR,
  Type VARCHAR(10))
```

3.3.2 永続クラスを使用した混合ストレージ・テーブルの定義

永続クラスを使用して混合ストレージによるテーブルを作成するには、個々のプロパティに STORAGEDEFAULT パラメータを指定します。このパラメータに有効な値は "columnar" と "row" (既定値) です。

```
Property propertyName AS dataType(STORAGEDEFAULT = ["row" | "columnar"])
```

以下の永続クラスは、列指向ストレージ・テーブルの定義を示しています。このテーブルは、前のセクションで作成した DDL 定義のテーブルと同等です。

```
  Class Sample.BankTransaction Extends %Persistent [ DdlAllowed, Final ]
  {
    Parameter STORAGEDEFAULT = "columnar";
    Parameter USEEXTENTSET = 1;

    Property AccountNumber As %Integer;
    Property TransactionDate As %Date;
    Property Description As %String(MAXLEN = 100);
    Property Amount As %Numeric(SCALE = 2, STORAGEDEFAULT = "columnar");
    Property Type As %String(VALUELIST = "-Deposit-Withdrawal-Transfer");

    Index BitmapExtent [ Extent, Type = bitmap ];
  }
```

3.3.3 混合ストレージの詳細

3.3.3.1 グローバル・ストレージ

混合ストレージによるテーブルでは、以下のような複数のグローバル・ストレージ構造の組み合わせを使用します。

- 行ストレージ・レイアウトによるデータは、データ/マスタ・グローバルに $list 形式で格納されます。
- 列指向ストレージ・レイアウトによるデータは、それぞれ別々のグローバルにベクトル・エンコーディングで格納されます。

以下の BankTransaction テーブルを考えます。

```
  Class Sample.BankTransaction Extends %Persistent [ DdlAllowed ]
  {
    Parameter STORAGEDEFAULT = "row";
    Parameter USEEXTENTSET = 1;

    Property AccountNumber As %Integer;
    Property TransactionDate As %Date;
    Property Description As %String(MAXLEN = 100);
    Property Amount As %Numeric(SCALE = 2, STORAGEDEFAULT = "columnar");
    Property Type As %String(VALUELIST = "-Deposit-Withdrawal-Transfer");

    Index BitmapExtent [ Extent, Type = bitmap ];
  }
```

このテーブルは、"永続クラスを使用した混合ストレージ・テーブルの定義" で取り上げた例とほとんど同じですが、Amount 列には列指向ストレージを使用し、他のすべての列には行ストレージを使用していることがわかります。テーブル・データの論理抽象化のこの例は、"行ストレージの詳細" と "列指向ストレージの詳細" で取り上げたテーブルと同じです。

SQL

```
SELECT AccountNumber,TransactionDate,Description,Amount,Type FROM Sample.BankTransaction
```

AccountNumber	TransactionDate	Description	Amount	Type
10001234	2022年2月22日	預金口座への預金	40.00	預金
10001234	2022年3月14日	ベンダへの支払い	-20.00	引き出し
10002345	2022年7月30日	当座預金への振り替え	-25.00	振り込み
10002345	2022年8月13日	預金口座への預金	30.00	預金

管理ポータルの [**グローバル**] ページに、このデータがどのように格納されているかが表示されます。データ/マスタ・グローバルには行のデータが格納されます。この形式は、"行ストレージの詳細" で取り上げた形式に類似していますが、Amount 列のデータが存在しません。

```
^BankT    = 4
^BankT(1) = $lb(10001234,66162,"Deposit to Savings","Deposit")
^BankT(2) = $lb(10001234,66182,"Payment to Vendor ABC","Withdrawal")
^BankT(3) = $lb(10002345,66320,"Transfer to Checking","Transfer")
^BankT(4) = $lb(10002345,66334,"Deposit to Savings","Deposit")
```

このテーブルには、Amount 列のデータを格納する別のグローバルがあります。この形式は、"列指向ストレージの詳細" で取り上げた形式と同等です。

```
^BankT.V1(1) = {"type":"decimal", "count":4, "length":5, "vector":[,40,-20,-25,30]}
```

このテーブルのメタデータには、列の順序に関する情報が記述されています。InterSystems IRIS では、この情報を使用して、行グローバルと列グローバルに格納したデータを使用するリレーショナル・テーブルを作成します。

3.3.3.2 文字列の照合

列指向ストレージを使用する (テーブルの既定ストレージの使用または特定フィールドに対する明示的な設定によって指定) すべての文字列型フィールドは、ストレージ・レイアウトが混在した状態にするときに、EXACT 照合を備えるように定義されます。

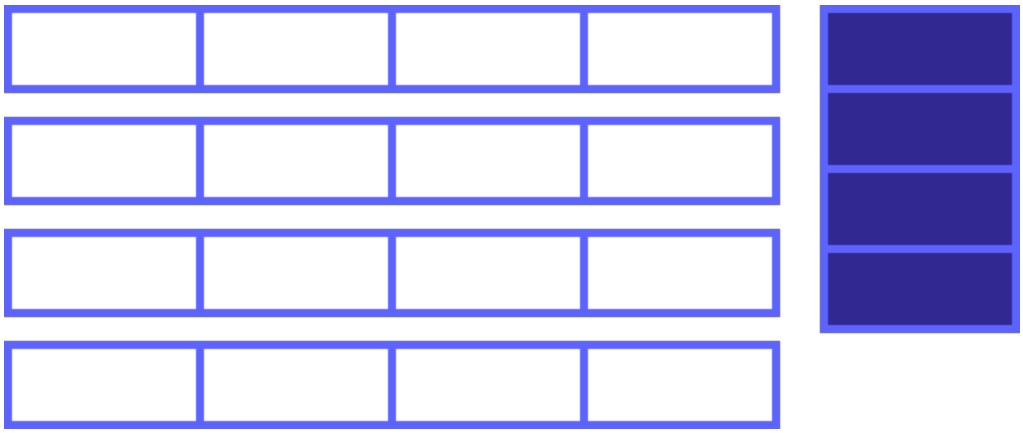
3.3.4 混合ストレージによる分析クエリ処理

分析クエリの効率は、アクセスするデータに左右されます。前のセクションで作成した BankTransaction テーブルを考えます。このテーブルでは、Amount 列にのみ列指向ストレージを使用しています。トランザクション金額すべての平均サイズを求めるクエリは、Amount 列にのみアクセスするので効率的です。

SQL

```
SELECT AVG(ABS(Amount)) FROM Sample.BankTransaction
```

以下の図では、Amount 列が、行単位で格納されている他の列から分離されています。



しかし、行として格納されている他の列の集計も実行するクエリでは、目に見えるパフォーマンス上の利点はなくなります。

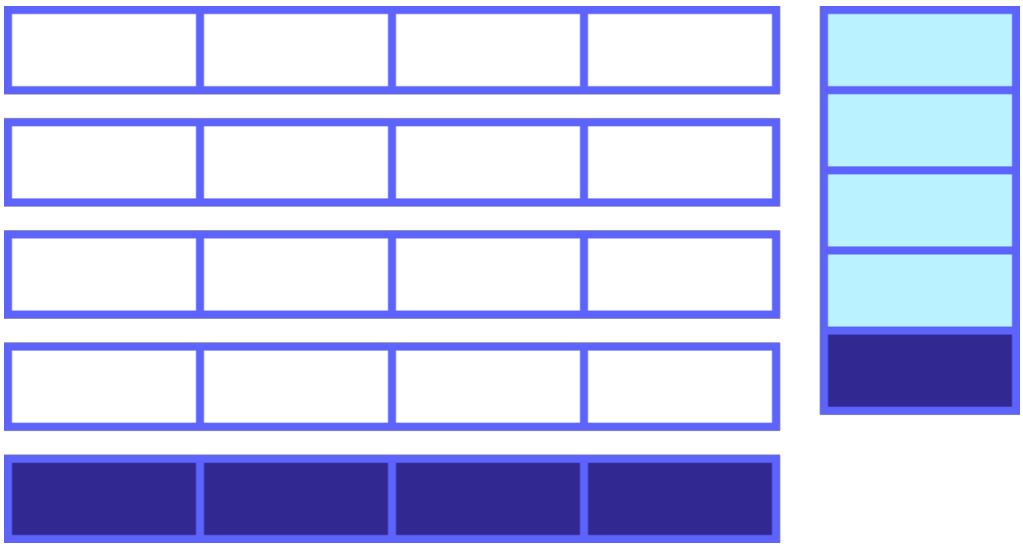
3.3.5 混合ストレージによるトランザクション処理

データの頻繁な更新と挿入を必要とする混合ストレージ・テーブルでは、行ストレージのみのテーブルよりもパフォーマンスが低下することが考えられますが、列指向ストレージのみのテーブルほどの低下ではありません。例えば、Amount 列にのみ列指向ストレージを使用した BankTransaction テーブルに新しい行を挿入するとします。

SQL

```
INSERT INTO Sample.BankTransaction
  VALUES (10002345,TO_DATE('01 SEP 2022'),'Deposit to Savings',10.00,'Deposit')
```

INSERT 操作では既存のどの行グローバルもロードされませんが、新しい Amount 値を挿入するには、Amount 列グローバルの最後のチャンク全体をメモリにロードする必要があります。データによっては、トランザクション向けとアナリティクス向けに別々のテーブルを維持するよりも、トランザクション処理によるこのオーバーヘッドを受け入れる方が望ましい場合もあります。



3.4 ストレージ・レイアウトのインデックス

選択したストレージ形式のタイプによって、テーブルに対するインデックスの定義が阻害されることはありません。インデックスから得られる利点は、ストレージ形式によって異なります。

3.4.1 行ストレージ・レイアウトのインデックス

"行ストレージによる分析クエリ処理" のセクションで説明したように、行ストレージ・レイアウトを使用したテーブルでは、その列に対するフィルタと集計の操作が低速になることがあります。このようなテーブルにビットマップ・インデックスまたは列指向インデックスを定義すると、このような分析操作のパフォーマンスの向上が期待できます。

ビットマップ・インデックスは、指定したインデックス付きのデータ値に対応する一連の ID 値を表す一連のビット文字列を使用します。この形式は高い比率で圧縮されているので、検索する行の数を削減できます。また、ブーリアン論理を使用してさまざまなビットマップ・インデックスを結合できるので、複数のフィールドやフィールド値を扱うフィルタの効率化を図ることができます。ビットマップの使用の詳細は、"ビットマック・インデックス" を参照してください。

以前のセクションで取り上げた `BankTransaction` テーブルを使用して、`Type` 列に以下のビットマップ・インデックスを作成するとします。

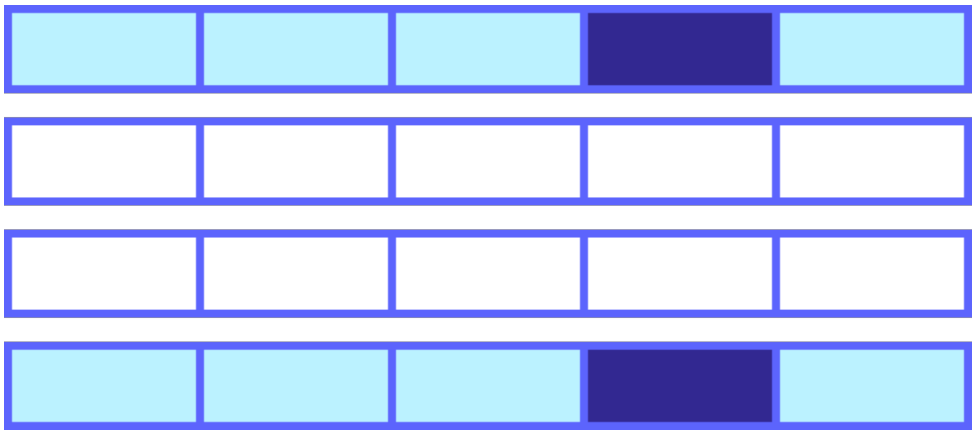
```
CREATE TABLE Sample.BankTransaction (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2),
  Type VARCHAR(10))

CREATE BITMAP INDEX TypeIndex
ON Sample.BankTransaction(Type)
```

それに続き、いずれかのトランザクション・タイプに基づいて行を制限する集計クエリを実行するとします。

SQL

```
SELECT AVG(ABS(Amount)) FROM Sample.BankTransaction WHERE Type = 'Deposit'
```

ビットマップ・インデックスを使用することで、以下の図のように、選択したトランザクション・タイプの行のみでクエリが繰り返されます。



ただし、この図でわかるように、ビットマップ・インデックスを使用して適格な行を探し出した後、必要な要素は行ごとに 1 つの要素のみであるにもかかわらず、行全体を取得し続ける必要があります。数百万行ものテーブルになると、フィルタで抽出された行であっても、パフォーマンス上の深刻なコストになることがあります。

代替策として、頻繁にクエリの対象となる列に列指向インデックスを定義します。列指向インデックスには、"列指向ストレージの詳細" で説明したものと同じベクトル化列データが格納されます。このインデックスを使用すると、追加したインデックスのストレージが必要にはなるものの、行ストレージ・テーブルに対する分析クエリのパフォーマンスが向上します。

InterSystems SQL DDL を使用して列指向インデックスを定義するには、次のように CREATE INDEX の CREATE COLUMNAR INDEX 構文を使用します。

```
CREATE COLUMNAR INDEX indexName ON table(column)
```

永続クラスに列指向インデックスを定義するには、次のように、定義するインデックスに type = columnar キーワードを指定します。

```
Index indexName ON propertyName [ type = columnar ]
```

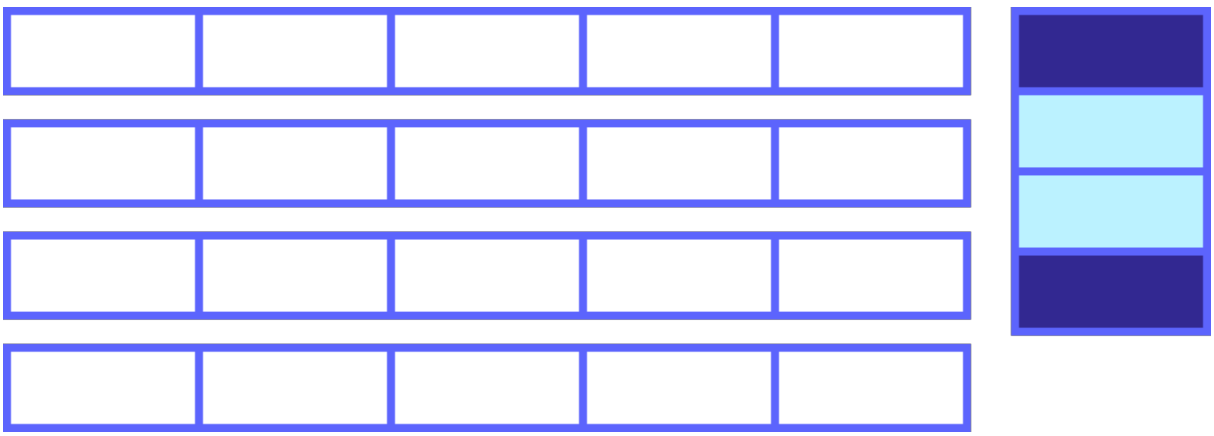
もう一度 BankTransaction テーブルを使用して、以下のように Type 列にビットマップ・インデックス、Amount 列に列指向インデックスをそれぞれ作成します。

```
CREATE TABLE Sample.BankTransaction (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2),
  Type VARCHAR(10))

CREATE BITMAP INDEX TypeIndex
ON Sample.BankTransaction(Type)

CREATE COLUMNAR INDEX AmountIndex
ON Sample.BankTransaction(Amount)
```

ここでは、以前に取り上げた集計クエリで両方のインデックスを組み合わせて使用し、クエリの対象になるデータのみにアクセスします。まず、TypeIndex ビットマップ・インデックスに基づいて、アクセスすべき行を以下のクエリで探し出します。続いて、AmountIndex 列指向インデックスに基づいて、これらの行の Amount 値にアクセスします。

SQL

```
SELECT AVG(ABS(Amount)) FROM Sample.BankTransaction WHERE Type = 'Deposit'
```



3.4.2 列指向ストレージ・レイアウトのインデックス

列指向ストレージ・テーブルの主要な目的が集計操作とフィルタ操作であるならば、インデックスの追加によって得られるパフォーマンス面の利点は多くありません。例えば、ビットマップ・インデックスでフィルタ処理のパフォーマンスは向上しますが、そのインデックスで必要になる追加のストレージとインデックスに起因する取り込みのオーバーヘッドに見合うほどの利点ではありません。クエリの負荷に高度に選択的なフィールドや一意のキーが関係している場合は、通常のインデックスを定義する方に価値があることも考えられます。このようなインデックスを定義する価値があるかどうかを判断

するには、クエリの試験とトレードオフの分析が必要です。インデックスの定義の詳細は、"インデックスの定義と構築" を参照してください。

3.5 行ストレージと列指向ストレージの推奨できるアプリケーション

テーブルで使用するストレージを列指向または行単位のどちらとするかを明確に判断する公式を InterSystems では用意していませんが、InterSystems SQL スキーマの構造を定義する際に効果的と考えられる一般的な指針はあります。一般的には以下の指針に従います。

- InterSystems IRIS SQL テーブルの行数が 100 万を下回っていれば、列指向ストレージを検討する必要はありません。ベクトル化ストレージの利点によって小規模なテーブルに大きな変化が見られる可能性は低いといえます。
- InterSystems SQL を活用するアプリケーションやトランザクション処理アプリケーションなどのオブジェクトには、既定の行単位ストレージ・レイアウトを使用します。アプリケーションやプログラム・トランザクションに発行される大半のクエリでは、限られた数の行を取得または更新します。また、集計機能を使用することはほとんどありません。このようなケースは、列指向ストレージとベクトル化クエリ処理が提供する利点の対象外です。
- このようなアプリケーションで運用分析を採用している場合、分析クエリのパフォーマンスが不十分であれば列指向インデックスを追加します。このような場合は、数量や通貨のように集計に使用される数値フィールドを探すか、タイムスタンプのように、カーディナリティが高く、範囲条件に使用されるフィールドを探します。列指向インデックスをビットマップ・インデックスと組み合わせて使用すると、マスタ・マップや通常のインデックス・マップからの過剰な読み取り操作の発生を防止できます。
- 分析の事例で InterSystems IRIS SQL スキーマをデプロイしている場合は列指向ストレージ・レイアウトを使用します。このような事例として、スター・スキーマやスノーフレーク・スキーマなどの正規化されていないテーブル構造、ビットマップ・インデックスと一括取り込みの多用などがあります。列指向ストレージから多大な利点が得られるのは、複数の行にわたって値を集計する場合の分析クエリです。列指向テーブルを定義している場合、列指向ストレージに適切に収まらない列は、自動的に行単位ストレージに戻されます。このような列として、ストリーム、長い文字列、シリアル・フィールドなどがあります。InterSystems IRIS SQL では混合テーブル・レイアウトが全面的にサポートされていて、クエリ・プランの適格な部分にはベクトル化クエリ処理が使用されます。列指向テーブルではビットマップ・インデックスを省略できます。このようなテーブルでは値が制限されているからです。

上記の推奨事項は、データ関連の要因とアプリケーションを実行する環境による影響を受けます。したがって、代表的な設定でさまざまなレイアウトを試験して、最良のパフォーマンスが得られるレイアウトを判断することをお勧めします。

4

永続クラスによる SQL 最適化テーブルの定義

InterSystems IRIS の SQL では、データ定義言語 (DDL) 文を記述する代わりに、自身を SQL テーブルとして提示する永続クラスを Object Script に定義できます。このページでは、クラス定義に使用することで、テーブルにアクセスする SQL 文に優れたパフォーマンスを実現できる一群のクラス機能について説明します。これらの機能は、新しいクラス定義にも既存のクラス定義にも適用できますが、目的のテーブルにデータが存在する場合は使用に際して注意を要するものがあります。

4.1 グローバルの命名方法

データの保存先となるグローバルの名前は、テーブルを定義する USEEXTENTSET クラス・パラメータと DEFAULTGLOBAL クラス・パラメータの値で決まります。インデックスの命名方法も、グローバルの名前によって決まります。この 2 つのパラメータの関係は以下のとおりです。

- USEEXTENTSET=0 であれば、ユーザ指定の名前と付加された文字コードでグローバル名が構成されます。例えば、名前が `Sample.MyTest` であるクラスは、マスタ・マップでは `^Sample.MyTestD` を名前とするグローバルに対応し、すべてのインデックス・マップでは `^Sample.MyTestI` を名前とするグローバルに対応します。DEFAULTGLOBAL を指定している場合は、永続クラス名に代わって、指定したグローバル名が使用されます。
- USEEXTENTSET=1 の場合は、マスタ・マップおよび独立した各インデックス・マップに対し、ハッシュ化したグローバル名が作成されます。この処理では、パッケージ名とクラス名の両方がハッシュ化され、マスタ・マップとインデックス・マップごとに増分する整数が名前に付加されます。この名前は、そこから意味を読み取れるものではありませんが、グローバルの保存と検索では低レベルで優れた効率を提供します。インデックス・マップごとに別々のグローバルを使用することも、パフォーマンス面と運用面で有利です。DEFAULTGLOBAL を指定した場合は、ハッシュ化したパッケージ名とクラス名に代わって、指定した名前が使用されます。

USEEXTENTSET クラス・パラメータと DEFAULTGLOBAL クラス・パラメータの両方が、ストレージ定義がどのように生成されるかに影響します。すでにストレージ定義があるクラスでこれらのパラメータを変更しても、そのストレージ定義を再設定しない限り、何の効果もありません。"ストレージ定義の再設定" を参照してください。この処理を実行すると既存のどのデータにもアクセスできなくなるので、インポート・データをロードする前に、これらのパラメータを開発環境でのみ更新するようにします。

USEEXTENTSET は 1 に設定することをお勧めします。これは、CREATE TABLE を使用してテーブルを作成するときの既定の設定です。%Persistent から継承するクラスで下位互換性を確保する必要がある場合は、USEEXTENTSET の既定の設定は 0 です。したがって、新しいクラスでは、初めてコンパイルする前に、このパラメータを 1 に設定することをお勧めします。

USEEXTENTSET パラメータと DEFAULTGLOBAL パラメータの詳細は、それぞれ "ハッシュ定義のグローバル名" と "ユーザ定義のグローバル名" を参照してください。

4.2 ストレージ・レイアウトの決定

永続クラスを定義して、クラスのすべてのパラメータまたは一部のパラメータで列指向ストレージを活用できます。このような手法の利点と欠点を ″SQL テーブルのストレージ・レイアウトの選択″ で説明しています。

既定では、クラスのすべてのパラメータで、永続クラスによって行ストレージ・レイアウトが使用されます。ただし、STORAGEDEFAULT パラメータを使用すると、この既定の設定を列指向に設定できます。また、一部のパラメータに行ストレージを使用し、他のパラメータには列指向ストレージを使用する混合ストレージ・レイアウトを定義できます。

注釈　InterSystems IRIS では、クラスにストレージ定義エントリを生成するときに、STORAGEDEFAULT パラメータの値が既定で使用されます。クラスを初めてコンパイルするとき、または新しいプロパティを追加するときにのみ、この値が考慮されます。ストレージの XData ブロックに保存されたストレージ定義がすでに存在するクラスでは、このパラメータを変更しても、クラス・エクステントについて格納されているデータも含め、そのストレージには何の影響もありません。したがって、クラスを初めてコンパイルする前に、使用するストレージ・レイアウトを決定する必要があります。

4.3 インデックス

テーブルのフィールドまたはフィールドのグループにインデックスを定義できます。複数の異なるタイプのインデックス（標準、ビットマップ、ビットスライス、および列指向）を定義できます。SQL の最適化では、定義済みのインデックスを使用し、これらのインデックスの対象になっているフィールドが関係する述語に基づいて、クエリ、更新、削除の各操作で特定のレコードにアクセスします。

クラス定義で定義するインデックスと DDL 文で定義するインデックスとの大きな相違点の一つとして、クラスで定義するインデックスは作成と構築を別々の操作にする必要があることが挙げられます。その理由は、クラスのコンパイルはクラス定義にのみ影響し、インデックスのエントリも含め、そのデータは変化しないことにあります。定義したインデックスを今後の SQL クエリで使用するには、そのインデックスを手動で構築する必要があります。

クラス定義でインデックスを定義する例は、″クラス定義を使用したインデックスの定義″ を参照してください。

インデックスのフィールドの詳細は、″インデックスの対象″ を参照してください。

4.4 エクステント・インデックス

エクステント・インデックスは、特定のフィールドのインデックスは作成せず、マスタ・マップにある各行の ID エントリを収めただけの特殊なタイプのインデックスです。ビットマップ適合の IDKEY があるクラスでは、エクステント・インデックスをビットマップ・エクステント・インデックスとして実装できます。このインデックスは、きわめて効率的な存在確認とカウントの機能を提供します。例えば、クエリ `SELECT COUNT(*) FROM t` は、ビットマップ・エクステント・インデックスがあるテーブルでは、それがないテーブルよりも桁違いに優れた効率を発揮します。

クラスにビットマップ・エクステント・インデックスを定義するには、以下のように記述します。

```
Index BME [ Extent, Type = bitmap ];
```

まれな状況として、ビットマップ適合の IDKEY がないクラスでは、Type キーワードを定義しなくてもかまいません。

`CREATE TABLE` DDL 文を使用してテーブルを作成するときに、ビットマップ・エクステント・インデックスが自動的に作成されます。どの永続クラスにも、ビットマップ・エクステント・インデックスを追加することをお勧めします。標準のインデックス同様に、エクステント・インデックスも作成とは別に構築する必要があります。