Defining a High-Performance Schema

Version 2023.1
2024-07-11

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

Defining a High-Performance Schema
InterSystems IRIS Data Platform Version 2023.1 2024-07-11

For Support questions about any InterSystems products, contact:

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

Table of Contents

List of Figures

List of Tables

1

Table Statistics for Query Optimizer

To ensure maximum performance of InterSystems SQL tables, you can create and employ representative or anticipated data metrics. The optimizations can have a significant effect on any queries run against this table. The following performance optimizing considerations are discussed in this topic:

- ExtentSize, Selectivity, and BlockCount to specify table data estimates before populating the table with data; this metadata is used to optimize future queries.
- Running Tune Table to analyze representative table data in a populated table; this generated metadata is used to optimize future queries.
- Tune Table Calculated Values include ExtentSize, Selectivity, Outlier Selectivity, Average Field Size, and BlockCount.
- Exporting and Re-importing Tune Table Statistics

1.1 ExtentSize, Selectivity, and BlockCount

When the Query Optimizer decides the most efficient way to execute a specific SQL query, three of the things it considers are:

- *ExtentSize* row count for each table used within the query.
- *Selectivity* the percentage of distinct values calculated for each column used by the query.
- *BlockCount* count for each SQL map used by the query.

In order to ensure that the Query Optimizer can make the correct decisions, it is important that these values are set correctly.

- You can explicitly set any of these statistics during class (table) definition, prior to populating the table with data.
- After populating the table with representative data, you can run Tune Table to calculate these statistics.
- After running Tune Table, you can override a calculated statistic by specifying an explicit value.

You can compare your explicitly set statistics to the Tune Table generated results. If the assumptions made by Tune Table result in less-than-optimal results from the Query Optimizer, you can use an explicitly set statistic rather than a Tune Table generated statistic.

In Studio the Class Editor window displays the class source code. At the bottom of the source code it displays the Storage definition, which includes the class *ExtentSize*, and the *Selectivity* (and, where appropriate, the *OutlierSelectivity*) for each property.

1.1.1 ExtentSize

The *ExtentSize* value for a table is simply the number of rows (roughly) stored within the table.

At development time, you can provide an initial *ExtentSize* value. If you do not specify an *ExtentSize*, the default is 100,000.

Typically, you provide a rough estimate of what you expect the size of this table will be when populated with data. It is not important to have an exact number. This value is used to compare the relative costs of scanning over different tables; the most important thing is to make sure that the relative values of *ExtentSize* between associated tables represent an accurate ratio (that is, small tables should have a small value and large tables a large one).

- **CREATE TABLE** provides an %EXTENTSIZE parameter keyword to specify the expected number of rows in the table, as shown in the following example:

  SQL

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

- A persistent class definition for a table can specify an ExtentSize parameter within the storage definition:

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

  In this example, the fragment is the storage definition for the *MyClass* class, which specifies a value of 200 for *ExtentSize*.

If your table has real (or realistic) data, you can automatically calculate and set its *ExtentSize* value using the Tune Table facility within the Management Portal; for details, see the following section on Tune Table.

1.1.2 Selectivity

Within an InterSystems SQL table (class), every column (property) has a *Selectivity* value associated with it. The *Selectivity* value for a column is the percentage of rows within a table that would be returned as a result of query searching for a typical value of the column. The presence of outlier values may change how InterSystems SQL interprets *Selectivity* values.

Selectivity is based on roughly equal quantities of the distinct values. For example, suppose a table contains a CoinFlip column whose values are roughly evenly distributed between “heads” and “tails”. The *Selectivity* value for the CoinFlip column would be 50%. The *Selectivity* value for a more distinguishing property, such as State, is typically a smaller percentage.

A field in which all the values are the same has a selectivity of 100%. To determine this, the optimizer first tests a small number or records and if these all have the same field value, it will test up to 100,000 randomly selected records to support the assumption that all of the values of a non-indexed field are the same. If other values for a field might not be detected in a test of 100,000 randomly-selected records, you should set the selectivity manually.

A field that is defined as Unique (all values different) has a selectivity of 1 (which should not be confused with a selectivity of 1.0000%). For example, a RowID has a selectivity of 1.

At development time, you can provide this value by defining a *Selectivity* parameter within the storage definition that is part of the class definition for the table:

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

Typically you provide an estimate of what you expect the *Selectivity* will be when used within an application. As with *ExtentSize*, it is not important to have an exact number. Many of the data type classes provided by InterSystems IRIS will provide reasonable default values for *Selectivity*.

You can also use the **SetFieldSelectivity()** method to set the *Selectivity* value for a specific field (property).

If your table has real (or realistic) data, you can automatically calculate and set its *Selectivity* values using the Tune Table facility within the Management Portal. Tune Table determines if a field has an outlier value, a value that is far more common than any other value. If so, Tune Table calculates a separate *Outlier Selectivity* percentage, and calculates *Selectivity* based on the presence of this outlier value. The presence of an outlier value may dramatically change the *Selectivity* value.

Selectivity is used for query optimization. The same *Selectivity* value is used for a field specified in a **SELECT** query and the same field specified in the **SELECT** clause of a view. Note that a view may have a different distribution of rows than the source table. This can affect the accuracy of view field selectivity.

1.1.3 BlockCount

When you compile a persistent class, the class compiler computes approximate numbers of map blocks used by each SQL map based on the *ExtentSize* and the property definitions. You can view these *BlockCount* values in the **Map BlockCount** tab of the Tune Table facility. The *BlockCount* is identified in Tune Table as `Estimated by class compiler`. Note that if you change the *ExtentSize*, you must close and re-open the SQL Tune Table window to see this change reflected in the *BlockCount* values.

When you run Tune Table, it measures the actual block count for each SQL map. Unless specified otherwise, the Tune Table measured values replace the class compiler approximate values. These Tune Table measured values are represented in the class definition as negative integers, to distinguish them from specified BlockCount values. This is shown in the following example:

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

Tune Table measured values are represented in Tune Table as positive integers, identified as `Measured by TuneTable`.

You can define explicit BlockCount values in the class definition. You can explicitly specify a block count as a positive integer, as shown in the following example:

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

When you define a class you can omit defining the BlockCount for a map, explicitly specify a BlockCount as a positive integer, or explicitly define the BlockCount as NULL.

- If you do not specify a BlockCount, or specify a BlockCount of 0, the class compiler estimates the block count. Running Tune Table replaces the class compiler estimated value.
- If you specify an explicit positive integer BlockCount, running Tune Table does not replace this explicit BlockCount value. Explicit class definition block count values are represented in Tune Table as positive integers, identified as `Defined in class definition`. These block count values are not changed by subsequently running Tune Table.

- If you specify an explicit BlockCount of NULL, the SQL Map uses the BlockCount value estimated by the class compiler. Because BlockCount is “defined” in the class definition, running Tune Table does not replace this estimated BlockCount value.

The size of all InterSystems SQL map blocks is 2048 bytes (2K bytes).

Tune Table does not measure BlockCount in the following circumstances:

- If the table is a child table projected by an array or a list collection. The BlockCount values for these types of child tables are the same as BlockCount for the data map of the parent table.
- If a global map is a remote global (a global in a different namespace). The estimated BlockCount used during class compilation is used instead.

1.2 Tune Table

Tune Table is a utility that examines the data in a table and returns statistics about the ExtentSize (the number of rows in the table), the relative distribution of distinct values in each field, and the Average Field Size (average length of values in each field). It also generates the BlockCount for each SQL map. You can specify that Tune Table use this information to update the metadata associated with a table and each of its fields. The query optimizer can subsequently use these statistics to determine the most efficient execution plan for a query.

Using Tune Table on an external table will only calculate the ExtentSize. Tune Table cannot calculate field Selectivity values, Average Field Size, or map BlockCount values for an external table.

1.2.1 When to Run Tune Table

You should run Tune Table on each table after that table has been populated with a representative quantity of real data. Commonly, you only need to run Tune Table once, as a final step in application development, before the data goes “live.” Tune Table is not a maintenance utility; it *should not* be run at regular intervals on live data. However, you should re-run Tune Table when the table’s structure changes significantly via inserts, updates, or deletes. As a general rule, when 20% of a table changes, regardless of the type of change, you should re-run Tune Table because the following characteristics have likely changed:

- Relative Table Sizes: Tune Table assumes that it is analyzing a representative subset of the data. This subset can be only a small percentage of the full data set, if it is a representative subset. Tune Table results remain relevant as the number of rows in a table changes, provided that the ExtentSizes of tables involved in joins or other relationships maintain roughly the same relative sizes. ExtentSize needs to be updated if the ratio between joined tables changes by an order of magnitude. This is important for JOIN statements, because the SQL optimizer uses ExtentSize when optimizing the table join order. As a general rule, a smaller table is joined before a larger table, regardless of the join order specified in the query.
- Even Value Distribution: Tune Table assumes that every data value is equally likely. If it detects an outlier value, it assumes that every data value other than the outlier value is equally likely. Tune Table establishes Selectivity by analyzing the current data values for each field. Equal likelihood in real data is always a rough approximation; normal variation in the number of distinct data values and their relative distribution should not warrant re-running Tune Table. However, an order-of-magnitude change in the number of possible values for a field (the ratio of distinct values to records), or the overall likelihood of a single field value can result in inaccurate Selectivity. Dramatically changing the percentage of records with a single field value can cause Tune Table to designate an outlier value or to remove outlier value designation, significantly changing the calculated Selectivity. If the Selectivity of a field no longer reflects the actual distribution of data values, you should re-run Tune Table.
- A significant InterSystems IRIS upgrade, or a new site installation may warrant re-running Tune Table.

Since efficient query processing depends on Tune Table statistics, InterSystems IRIS runs Tune Table:

- Automatically when the first **SELECT** query is executed against a table that has never been tuned. Automatic Tune Table execution is only performed under very specific circumstances.
- Manually when you execute Tune Table using any of the supported user interfaces.

1.2.2 Automatic Tune Table

InterSystems IRIS automatically runs Tune Table the first time a **SELECT** query is executed against a table, subject to the following circumstances:

- The table has never been tuned, either by a prior execution of Tune Table or by setting ExtentSize or Selectivity values. All non-unique fields must have selectivity = "".
- The query has never been run. The query has not been cached. Executing the query invokes the optimizer to generate a query plan.
- The table is a regular table. It is not a view or a some other non-table data structure.
- The table is not sharded.
- The table is not a linked table.
- The query is not being run as part of the build routine.

1.2.3 Manual Tune Table

There are three manual interfaces for running Tune Table:

- Using the Management Portal SQL interface **Actions** drop-down list, which allows you to run Tune Table on a single table or on multiple tables.
- Invoking the **$SYSTEM.SQL.Stats.Table.GatherTableStats()** method for a single table, or all tables in the current namespace.
- Issuing the SQL command TUNE TABLE for a single table.

Tune Table purges cached queries that reference the table being tuned. The **TUNE TABLE** command provides a recompile cached queries option to regenerate the cached queries using the new Tune Table calculated values.

If the table is mapped to a readonly database, Tune Table cannot be performed and an error message is generated.

After running the Tune Table facility, the resulting *ExtentSize* and *Selectivity* values are saved in the class's storage definition. To view the storage definition, in Studio, from the **View** menu, select **View Storage**; Studio includes the storage at the bottom of the source code for the class.

Note: There are rare cases where running Tune Table can decrease SQL performance. While Tune Table can be run on live data, it is recommended that you run Tune Table on a test system with real data, rather than on a production system. You can use the optional System Mode configuration parameter to indicate whether the current system is a test system or a live system. When set, the System Mode is displayed at the top of the Management Portal page, and can be returned by the **$SYSTEM.Version.SystemMode()** method.

1.2.3.1 Tune Table from the Management Portal

To run Tune Table from the Management Portal:

1. Select **System Explorer**, then **SQL**. Select a namespace by clicking the **Switch** option at the top of the page, then selecting a namespace from the displayed list. (You can set the Management Portal default namespace for each user.)

2. Select a **Schema** from the drop-down list on the left side of the screen, or use a **Filter**. For further details on how to use **Schema** and **Filter**, refer to Browsing SQL Schemas.
3. Do one of the following:
   - Tune a Single Table: Expand the **Tables** category and select a table from the list. Once you have selected a table, click the **Actions** drop-down list and select **Tune Table Information**. This displays the table's current ExtentSize and Selectivity information. If Tune Table has never been run, ExtentSize=100000, no Selectivity, Outlier Selectivity, Outlier Value, or Average Field Size information is shown (other than the RowID having a selectivity of 1), and the Map BlockCount information is listed as `Estimated by class compiler`.

     From the **Selectivity** tab, select the **Tune Table** button. This runs Tune Table on the table, calculating the ExtentSize, Selectivity, Outlier Selectivity, Outlier Value, and Average Field Size values based on the data in the table. The Map BlockCount information is listed as `Measured by Tune Table`.

     Tune Table on a single table always runs as a background process and refreshes the table when done. This prevents timeout issues. While this background process is running, an `in process` message is displayed. The **Close** button is available to close the Tune Table window while the background process executes.
   - Tune All Tables in the Schema: click the **Actions** drop-down list and select **Tune All Tables in Schema**. This displays the Tune Table box. Select the **Finish** button to run Tune Table on all tables in the schema. When Tune Table completes this box displays a **Done** button. Select **Done** to exit the Tune Table box.

The SQL Tune Table window has two tabs: **Selectivity** and **Map BlockCount**. These tabs display the current values generated by Tune Table. They also allow you to manually set different values than the values generated by Tune Table.

The **Selectivity** tab contains the following fields:

- Current Table Extentsize. This field has an **edit** button that allows you to enter a different Table Extentsize.
- **Keep class up to date** check box. Any changes to statistics generated by Tune Table, or by user input value from the Tune Table interface, or from Tune Table methods are immediately represented in the class definition:
  - If this box is not checked (No), the up-to-date flag on the modified class definition is not set. This indicates that the class definition is out of date and should be recompiled. This is the default.
  - If this box is checked (Yes), the class definition remains flagged as up-to-date. This is the preferred option when making changes to statistics on a live system, because it makes it less likely that a table class definition will be recompiled.
- Fields table with columns for Field Name, Selectivity, Notes, Outlier Selectivity, Outlier Value, and Average Field Size. By clicking on a Fields table heading, you can sort by that column's values. By clicking on a Fields table row, you can manually set values for Selectivity, Outlier Selectivity, Outlier Value, and Average Field Size for that field.

The **Map BlockCount** tab contains the following fields:

- Map Name table with columns for SQL Map Name, BlockCount, and Source of BlockCount. The SQL Map Name for an index is the SQL index name; this may differ from the persistent class index property name.

  By clicking on an individual map name, you can manually set a BlockCount value for that map name.

From the **Selectivity** tab, you can click the **Tune Table** button to run Tune Table on this table.

1.2.3.2 Tune Table using a Method

You can use the **$SYSTEM.SQL.Stats.Table.GatherTableStats()** method to run the Tune Table facility in the current namespace.

- **GatherTableStats("Sample.MyTable")** to run Tune Table on a single table.

- **GatherSchemaStats(“Sample”)** to run Tune Table on all tables in the specified schema.
- **GatherTableStats(“*”)** to run Tune Table on all tables in the current namespace.

When using the **GatherTableStats()** method, the following error messages may be generated:

- Non-existent table: `DO $SYSTEM.SQL.Stats.Table.GatherTableStats("NoSuchTable")`

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

  `'Sample.MyView' is a view, not a table. No tuning will be performed.`

When running **GatherTableStats(”*”)** or **GatherSchemaStats(”SchemaName”)**, the system will use multiple processes to tune multiple tables in parallel, if the system supports parallel processing.

1.2.4 Running Tune Table on a Sharded Table

If Tune Table is run on a sharded table, the Tune Table operation is forwarded to each of the shards, where it runs against that shard of the table. Tune Table does not execute in the master namespace from which it was invoked. If Tune Table is run on a non-sharded table which has had its class definition exported to the shards, because that table is joined to a sharded table, the Tune Table operation is forwarded to each of the shards, and it is also executed in the master namespace.

The following guidelines should be followed when running Tune Table on a sharded table:

- Tune the shard-master table, not the shard-local table.
- EXTENTSIZE and BLOCKCOUNT values are per-shard values, not a total value for all shards.
- If using **$SYSTEM.SQL.Stats.Table.Export()** and **$SYSTEM.SQL.Stats.Table.Import()**, export/import the table statics for the shard-master table, not the shard-local table.
- When gather statistics for a sharded table, the RecompileCachedQueries argument is ignored, and cached queries for the table are always purged.
- Gathering statistics for a sharded table will define table statistics in both the shard-master and shard-local class/table definition. If manually editing tune table metadata in the class definitions, the suggested procedure is to modify the definition of the shard-master class, then recompile the shard-master class. When the shard-master class is compiled, the shard-master table statistics will be copied to the shard-local version of the class.

If **GatherTableStats()** or **GatherSchemaStats()** specifies a *logFile* parameter, the log file in the shard master instance has an entry for the specified table, such as the following:

- Sharded table: `TABLE: <tablename> Invoking TuneTable on shards for sharded table <tablename>`
- Non-sharded table: `TABLE: <tablename> Invoking TuneTable on shards for mapped non-sharded table <tablename>`

On each of the shard instances, a log file of the same name is created in the mgr/<shard-namespace> directory, logging the Tune Table information for this table on this shard. If a directory path was specified for the log file, it is ignored on the shards, and the file is always stored in mgr/<shard- namespace>.

1.3 Tune Table Calculated Values

The Tune Table operation calculates and sets table statistics based on the representative data in the table:

- ExtentSize, which may or may not be the actual number of rows in the table (Row Count).
- Selectivity for each property (field) in the table. You can optionally prevent selectivity calculation for individual properties.
- Outlier Selectivity for a property in which one value appears much more commonly than other values. An efficient query can make use of outlier optimization or use RTPC to choose a plan based on the query parameters as runtime.
- Notes for each property that identify certain property characteristics.
- Average Field Size for each property.
- SQL Map Name, BlockCount, and Source of BlockCount for the table.

1.3.1 Extent Size and the Row Count

When running the Tune Table facility from the Management Portal, the *ExtentSize* is the actual count of the rows currently in the table. By default, the **GatherTableStats()** method also uses the actual row count as the ExtentSize. When a table contains a large number of rows, it may be preferable to perform analysis on a smaller number of rows. You can use the SQL TUNE TABLE command and specify %SAMPLE_PERCENT to perform analysis on only a percentage of the total rows. You can use this option to improve performance when running against a table with a large number of rows. This %SAMPLE_PERCENT value should be large enough to sample representative data. If ExtentSize < 1000, **TUNE TABLE** analyzes all rows, regardless of the %SAMPLE_PERCENT value.

A specified ExtentSize can be smaller or larger than the actual number of rows. However, ExtentSize should not significantly exceed the actual number of rows in the current table data. When you specify an ExtentSize, Tune Table extrapolates row Ids for that number of rows, then performs sampling. If the ExtentSize greatly exceeds the actual number of rows, most of the sampled row Ids will not correspond to actual row data. If this is the case, field selectivities cannot be calculated; instead, Tune Table lists the ExtentSize you specified as the CALCULATED ExtentSize and a smaller number as the SAMPLESIZE; Tune Table returns <Not Specified> for these non-existent Calculated values.

You can set an ExtentSize of 0. This may be desirable when you have a table that is never intended to be populated with data, but used for other purposes such as query joins. When you set ExtentSize to 0, InterSystems IRIS sets the Selectivity of each field as 100%, and the Average Field Size of each field as 0.

1.3.2 Selectivity and Outlier Selectivity

Tune Table calculates a *Selectivity* for each property (field) value as a percentage. It does this by sampling the data, so selectivity is always an estimate, not an exact value. Selectivity is based on the assumption that all property values are, or could be, equally likely. This is a reasonable assumption for most data. For example, in a general population table most data values are typical: any given date of birth will appear in around .27% of the data (1 in 365); roughly half will be female and half male (50%). A field that is defined as Unique has a selectivity of 1 (which should not be confused with a selectivity of 1.0000 (1%). A selectivity percentage is sufficient for most properties.

For a few properties, Tune Table also calculates an *Outlier Selectivity*. This is a percentage for a single property value that appears much more frequently in the sample than the other data values. Tune Table only returns an outlier selectivity when there is a substantial difference between the frequency of one data value and the frequency of the other data values. Tune Table returns, at most, one outlier for a table, regardless of the distribution of data values. If an outlier is selected, Tune Table displays this value as the *Outlier Value*. NULL is represented as `<Null>`. If Tune Table returns an outlier selectivity, the normal selectivity is still the percentage of each non-outlier data value within the whole set of rows.

If the Tune Table initial sample returns only a single value, but additional sampling returns more than a single distinct value, the normal selectivity is modified by these sampling results. For example, an initial random sample of 990 values detects only one value, but subsequent sampling detects 10 single instances of other distinct values. In this situation, the initial outlier value influences the selectivity value, which is now set to 1/1000 (0.1%) since each of the 10 non-outlier values occurs only once in the 1000 records.

The most common example of outlier selectivity is a property that permits NULLs. If the number of records with NULL for a property greatly exceeds the number of records that have any specific data value for that property, NULL is the outlier. The following is the Selectivity and Outlier Selectivity for the FavoriteColors field:

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

If a field only contains one distinct value (all rows have the same value), that field has a Selectivity of 100%. A value that has a selectivity of 100% is not considered to be an outlier. Tune Table establishes Selectivity and Outlier Selectivity values by sampling the data. To determine this, the Tune Table first tests a small number or records and if these all have the same field value, it will test up to 100,000 randomly selected records to support the assumption that all of the values of a non-indexed field are the same. Tune Table can only fully determine if all the values of a field are the same if the field is indexed, the field is the first field of the index, and the field and the index have the same collation type.

- If a non-indexed field is known to have other values that might not be detected in a test of 100,000 randomly-selected records, you should set the selectivity and outlier selectivity manually.
- If a non-indexed field is known to have no other values, you can manually specify a Selectivity of 100%, delete any outlier selectivity, and set CALCSELECTIVITY=0 to prevent Tune Table attempting to calculate selectivity or specify this value as an outlier.

To modify these *Selectivity*, *Outlier Selectivity*, and *Outlier Value* calculated values, select an individual field from the Tune Table display. This displays these values for that field in the **Details** area to the right of the display. You can modify *Selectivity*, *Outlier Selectivity*, and/or *Outlier Value* to values that better fit the anticipated full data set.

- You can specify *Selectivity* either as a percentage of rows with a percent (%) sign, or as an integer number of rows (no percent sign). If specified as an integer number of rows, InterSystems IRIS uses the extent size to calculate the *Selectivity* percentage.
- You can specify an *Outlier Selectivity* and *Outlier Value* for a field that previously had no outlier. Specify *Outlier Selectivity* as a percentage with a percent (%) sign. If you specify just the *Outlier Selectivity*, Tune Table assumes the *Outlier Value* is <Null>. If you specify just the *Outlier Value*, Tune Table will not save this value unless you also specify an *Outlier Selectivity*.

1.3.3 CALCSELECTIVITY Parameter and Not Calculating Selectivity

Under certain circumstances, you may not want the Tune Table facility to calculate the *Selectivity* for a property. To prevent *Selectivity* from being calculated, specify the value of the property's *CALCSELECTIVITY* parameter to 0 (the default is 1). In Studio, you can set *CALCSELECTIVITY* on the **Property Parameters** page of the **New Property Wizard** or in the list of a property's parameters in the Inspector (you may need to contract and re-expand the property parameter list to display it).

One circumstance where you should specify *CALCSELECTIVITY*=0 is a field that is known to contain only one value in all rows (Selectivity=100%), if that field is not indexed.

1.3.4 The Notes Column

The Management Portal **Tune Table Information** option displays a Notes column for each field. The values in this field are system-defined and non-modifiable. They include the following:

- `RowID field`: A table has one RowID, which is defined by the system. Its name is commonly ID, but it can have a different system-assigned name. Because all of its values are (by definition) unique, its Selectivity is always 1. If the class definition includes SqlRowIdPrivate, the Notes column value is `RowID field, Hidden field`.

- `Hidden field`: A hidden field is defined as private, and is not displayed by SELECT *. By default, **CREATE TABLE** defines the RowID field as hidden; you can specify the %PUBLICROWID keyword to make the RowID not hidden and public. By default, tables defined by a persistent class definition define the RowID as not hidden; you can specify SqlRowIdPrivate to define the RowID as hidden and private. Container fields are defined as hidden.
- `Stream field`: Indicates a field defined with a stream data type, either character stream (CLOB) or binary stream (BLOB). A stream file has no Average Field Size.
- `Parent reference field`: A field that references a parent table.

An IDENTITY field, ROWVERSION field, SERIAL field, or UNIQUEIDENTIFIER (GUID) field is *not* identified in the Notes column.

1.3.5 Average Field Size

Running Tune Table calculates the average field size (in characters) for all non-Stream fields, based on the current table data set. This is (unless otherwise noted) the same as `AVG($LENGTH(field))`, rounded to two decimal places. You can change this average field size for individual fields to reflect the anticipated average size of the field's data.

- NULL: Because the **$LENGTH** function treats NULL fields as having a length of 0, NULL fields are averaged in, with a length 0. This may result in an Average Field Size of less than one character.
- Empty column: If a column contains no data (no field values for all of the rows), the average field size value is 1, not 0. The `AVG($LENGTH(field))` is 0 for a column that contains no data.
- ExtentSize=0: When you set ExtentSize to 0, Average Field Size for all fields is reset to 0.
- Logical field values: Average Field Size is always calculated based on the field's Logical (internal) value.
- List fields: InterSystems IRIS List fields are calculated based on their Logical (internal) encoded value. This encoded length is longer than the total length of the elements in the list.
- Container fields: A container field for a collection is larger than the total length of its collection objects. For example, in Sample.Person the Home container field Average Field Size is larger than the total of the average field sizes of Home_Street, Home_City, Home_State, and Home_Zip. For further details, refer to Controlling the SQL Projection of Collection Properties.
- Stream fields: A stream field does not have an average field size.

If the property parameter CALCSELECTIVITY is set to 0 for a property/field, Tune Table does not calculate the Average Field Size for that property/field.

You can modify an *Average Field Size* calculated value by selecting an individual field from the Tune Table display. This displays the values for that field in the **Details** area to the right of the display. You can modify the *Average Field Size* to a value that better fits the anticipated full data set. Because Tune Table performs no validation when you set this value, you should make sure that the field is not a Stream field, and that the value you specify is not larger than the maximum field size (MaxLen).

The Average Field Size is also displayed in the Management Portal **Catalog Details** tab **Fields** option table. Tune Table must have been run for the **Fields** option table to display Average Field Size values. For further details, refer to Catalog Details Tab.

1.3.6 Map BlockCount Tab

The Tune Table **Map BlockCount** tab displays the **SQL Map Name**, **BlockCount** (as a positive integer), and **Source of BlockCount**. The **Source of BlockCount** can be `Defined in class definition`, `Estimated by class compiler`, or `Measured by TuneTable`. Running Tune Table changes `Estimated by class compiler` to `Measured by TuneTable`; it does not affect `Defined in class definition` values.

You can modify a *BlockCount* calculated value by selecting an individual SQL Map Name from the Tune Table display. This displays the block count for that Map Name in the **Details** area to the right of the display. You can modify the *BlockCount* to a value that better fits the anticipated full data set. Because Tune Table performs no validation when you set this value, you should make sure that the block count is a valid value. Modifying *BlockCount* changes the **Source of BlockCount** to `Defined in class definition`. For further details, refer to BlockCount.

1.4 Exporting and Re-importing Tune Table Statistics

You can export Tune Table statistics from a table or group of tables and then import these Tune Table statistics into a table or group of tables. The following are three circumstances in which you might want to perform this export/import. (For simplicity, these describe the export/import of statistics from a single table; in actual use, export/import of statistics from multiple inter-related tables is often performed):

- To model a production system: A production table is fully populated with actual data and optimized using Tune Table. In a test environment you create a table with the same table definition but far less data. By exporting the Tune Table statistics from the production table and importing them into the test table, you can model the production table optimization on the test table.
- To replicate a production system: A production table is fully populated with actual data and optimized using Tune Table. A second production table with the same table definition is created. (For example, a production environment and its backup environment, or a multiple identical table definitions with each table containing the patient records for a different hospital.) By exporting the Tune Table statistics from the first table and importing them into the second table, you can give the second table the same optimization as the first table without the overhead of running Tune Table a second time or waiting for the second table to be populated with representative data.
- To revert to a prior set of statistics: You can create optimization statistics for a table by running Tune Table or by explicitly setting statistics. By exporting these statistics you can preserve them while experimenting with other statistics settings. Once you have determined the optimal set of statistics, you can import them back into the table.

You can export Tune Table statistics to an XML file using the **$SYSTEM.SQL.Stats.Table.Export()** method. This method can export the Tune Table statistics for one, more than one, or all tables within a namespace, as shown in the following examples:

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 */
```

You can re-import Tune Table statistics that were exported using **$SYSTEM.SQL.Stats.Table.Export()** by using the **$SYSTEM.SQL.Stats.Table.Import()** method.

$SYSTEM.SQL.Stats.Table.Import() has a KeepClassUpToDate boolean option. If TRUE (and update is TRUE), **$SYSTEM.SQL.Stats.Table.Import()** will update the class definition with the new EXTENTSIZE and SELECTIVITY values, but the class definition will be kept as up-to-date. In many cases, however, it is desirable to recompile the class after its table has been tuned so that queries in the class definition can be recompiled and the SQL query optimizer can use the updated data statistics. The default is FALSE (0). Note that if the class is deployed the class definition will not be updated.

$SYSTEM.SQL.Stats.Table.Import() has a ClearCurrentStats boolean option. If TRUE, **$SYSTEM.SQL.Stats.Table.Import()** will clear any prior EXTENTSIZE, SELECTIVITY, BLOCKCOUNT and other Tune Table statistics from the existing table before importing the stats. This can be used if you want to completely clear those table stats that are not specified in the import file, instead of leaving them defined in the persistent class for the table. The default is FALSE (0).

If **$SYSTEM.SQL.Stats.Table.Import()** does not find the corresponding table, it skips that table and proceeds to the next table specified in the import file. If a table is found, but some of the fields are not found, those fields will simply be skipped.

The BlockCount for a map in a class storage definition cannot be inherited. The BlockCount can only appear in the storage definition of the class where the map originated. **$SYSTEM.SQL.Stats.Table.Import()** only sets the projected table's BlockCount metadata and not the class storage BlockCount metadata if the map originated in a super class.

2

Define and Build Indexes

2.1 Overview

An index is a structure maintained by a persistent class that InterSystems IRIS® data platform can use to optimize queries and other operations.

You can define an index on the values of a field within a table, or the corresponding property within a class. (You can also define an index on the combined values of several fields/properties.) The same index is created, regardless of whether you defined it using SQL field and table syntax, or class property syntax. InterSystems IRIS automatically defines indexes when certain types of fields (properties) are defined. You can define additional indexes on any field in which data is stored or for which data can be reliably derived. InterSystems IRIS provides several types of indexes. You can define more than one index for the same field (property), providing indexes of different types for different purposes.

InterSystems IRIS populates and maintains indexes (by default) whenever a data insert, update, or delete operation is carried out against the database, whether using SQL field and table syntax, or class property syntax. You can override this default (by using the %NOINDEX keyword) to rapidly make changes to the data, and then build or rebuild the corresponding index as a separate operation. You can define indexes before populating a table with data. You can also define indexes for a table that is already populated with data and then build the index as a separate operation.

InterSystems IRIS makes use of available indexes when preparing and executing SQL queries. By default it selects which indexes to use to optimize query performance. You can override this default to prevent the use of one or more indexes for a specific query or for all queries, as appropriate. For information about optimizing index usage, refer to Using Indexes.

2.1.1 Index Attributes

Every index has a unique name. This name is used for database administrative purposes (reporting, index building, dropping indexes, and so on). Like other SQL entities, an index has both an SQL index name and a corresponding index property name; these names differ in permitted characters, case-sensitivity, and maximum length. If defined using the SQL **CREATE INDEX** command, the system generates a corresponding index property name. If defined using a persistent class definition, the SqlName keyword allows the user to specify a different SQL index name (SQL map name). The Management Portal SQL interface Catalog Details displays the SQL index name (**SQL Map Name**) and the corresponding index property name (**Index Name**) for each index.

The index type is defined by two index class keywords, Type and Extent. The types of indexes available with InterSystems IRIS include:

- Standard Indexes (Type = index) — A persistent array that associates the indexed value(s) with the RowID(s) of the row(s) that contains the value(s). Any index not explicitly defined as a bitmap index, bitslice index, or extent index is a standard index.

- Bitmap Indexes (Type = bitmap) — A special kind of index that uses a series of bitstrings to represent the set of RowID values that correspond to a given indexed value; InterSystems IRIS includes a number of performance optimizations for bitmap indexes.
- Bitslice Indexes (Type = bitslice) — A special kind of index that enables very fast evaluation of certain expressions, such as sums and range conditions. Certain SQL queries automatically use bitslice indexes.
- Columnar Indexes (Type = columnar) — A special kind of index that enables very fast queries, especially ones involving filtering and aggregation operations, on columns whose underlying data is stored across rows. Columnar indexes are an experimental feature for 2022.2.
- Extent Indexes — An index of all of the objects in an extent. For more information, see the Extent index keyword page.

The maximum number of indexes for a table (class) is 400.

2.1.2 Storage Type and Indexes

The index functionality described here applies to data stored in a persistent class. InterSystems SQL supports index functionality for data stored using the InterSystems IRIS default storage structure, %Storage.Persistent (%Storage.Persistent-mapped classes), and for data stored using %Storage.SQL (%Storage.SQL-mapped classes). You can define an index for a %Storage.SQL-mapped class using a functional index type. The index is defined in the same manner as an index in a class using default storage, with the following special considerations:

- The class must define the IdKey functional index, if it is not automatically system assigned. See Master Map, below.
- This functional index must be defined as an INDEX.

Refer to %Library.FunctionalIndex for further details.

2.1.3 Index Global Names

The name of the subscripted global that stores index data is determined by the name of the global that stores the data in the table. These names are dependent on the values of the USEEXTENTSET and DEFAULTGLOBAL class parameters that define the table, either in a persistent class or by using the %CLASSPARAMETER key word in a **CREATE TABLE** statement. For more information about USEEXTENTSET and DEFAULTGLOBAL, see "Hashed Global Names" and "User-Defined Global Names" respectively.

2.1.4 Master Map

The system automatically defines a Master Map for every table. The Master Map is not an index, it is a map that directly accesses the data itself using its map subscript field(s). By default, the master map subscript field is the system-defined RowID field. By default, this direct data access using the RowID field is represented with the SQL Map Name (SQL index name) IDKEY.

By default, a user-defined primary key is not the IDKEY. This is because Master Map lookup using RowID integers is almost always more efficient than lookup by primary key values. However, if you specify that the primary key is the IDKEY, the primary key index is defined as the Master Map for the table and SQL Map Name is the primary key SQL index name.

For a single-field primary key/IDKEY, the primary key index is the Master Map, but the Master Map data access column remains the RowID. This is because there is a one-to-one match between a record's unique primary key field value and its RowID value, and RowID is the presumed more efficient lookup. For a multi-field primary key/IDKEY, the Master Map is given the primary key index name, and the Master Map data access columns are the primary key fields.

You can view the Master Map definition through the Management Portal SQL Catalog Details tab. This displays, among other items, the global name where the Master Map data is stored. For SQL and default storage, this Master Map global

defaults to ^package.classnameD and the namespace is recorded to prevent ambiguity. For custom storage, no Master Map data storage global is defined; you can use the DATALOCATIONGLOBAL class parameter to specify a data storage global name.

For SQL and default storage, the Master Map data is stored in a subscripted global named either `^package.classnameD` or `^hashpackage.hashclass.1` (refer to Index Global Names for more information). Note that the global name specifies the persistent class name, not the corresponding SQL table name, and that the global name is case-sensitive. You can supply the global name to ZWRITE to display the Master Map data.

Data access using a Master Map is inefficient, especially for large tables. For this reason, it is recommended that the user define indexes that can be used to access data fields specified in WHERE conditions, JOIN operations, and other operations.

2.2 Automatically-Defined Indexes

The system automatically defines certain indexes when you define a table. The following indexes are automatically generated when you define a table and populated when you add or modify table data. If you define:

- A primary key that is not an IDKEY, the system generates a corresponding index of type Unique. The name of the primary key index may be user-specified or derived from the name of the table. For example, if you define an unnamed primary key, the corresponding index will be named tablenamePKEY#, where # is a sequential integer for each unique and primary key constraint.
- A UNIQUE field, InterSystems IRIS generates an index for each UNIQUE field with the name tablenameUNIQUE#, where # is a sequential integer for each unique and primary key constraint.
- A UNIQUE constraint, the system generates an index for each UNIQUE constraint with the specified name, indexing the fields that together define a unique value.
- A shard key, the system generates an index on the shard key field(s) named ShardKey.

You can view these indexes through the Management Portal SQL Catalog Details tab. The CREATE INDEX command can be used to add a UNIQUE field constraint; the DROP INDEX command can be used to remove a UNIQUE field constraint.

By default, the system generates the IDKEY index on the RowID field. Defining an IDENTITY field does not generate an index. However, if you define an IDENTITY field and make that field the primary key, InterSystems IRIS defines the IDKEY index on the IDENTITY field and makes it the primary key index. This is shown in the following example:

SQL

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

Similarly, if you define an IDENTITY field and give that field a UNIQUE constraint, InterSystems IRIS explicitly defines an IdKey/Unique index on the IDENTITY field. This is shown in the following example:

SQL

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

These IDENTITY indexing operations only occur when there is no explicitly defined IdKey index and the table contains no data.

2.2.1 Bitmap Extent Index

A bitmap extent index is a bitmap index for the rows of the table, not for any specified field of the table. In a bitmap extent index, each bit represents a sequential RowID integer value, and the value of each bit specifies whether or not the corresponding row exists. InterSystems SQL uses this index to improve performance of COUNT(*), which returns the number of records (rows) in the table. A table can have, at most, one bitmap extent index. Attempting to create more than one bitmap extent index results in an SQLCODE -400 error with the %msg `ERROR #5445: Multiple Extent indexes defined: DDLBEIndex`.

- A table defined using CREATE TABLE automatically defines a bitmap extent index. This automatically-generated index is assigned the Index Name DDLBEIndex and the SQL MapName %%DDLBEIndex.
- A table defined as a persistent class does not automatically define a bitmap extent index. If you add a bitmap index to a persistent class that does not have a bitmap extent index, InterSystems IRIS automatically generates a bitmap extent index. This generated bitmap extent index has an Index Name and SQL MapName of $*ClassName* (where *ClassName* is the name of the table's persistent class).

You can use the **CREATE INDEX** command with the BITMAPEXTENT keyword to add a bitmap extent index to a table, or to rename an automatically-generated bitmap extent index. For further details, refer to CREATE INDEX.

You can view a table's bitmap extent index through the Management Portal SQL Catalog Details tab. Though a table can have only one bitmap extent index, a table that inherits from another table is listed with both its own bitmap extent index and the bitmap extent index of the table it extends from. For example, the Sample.Employee table extends the Sample.Person table; in the **Catalog Details** Maps/Indices Sample.Employee lists both a `$Employee` and `$Person` bitmap extent index.

In a table that undergoes many **DELETE** operations the storage for a bitmap extent index can gradually become less efficient. You can rebuild a Bitmap Extent index either by using the BUILD INDEX command or from the Management Portal by selecting the table's **Catalog Details** tab, Maps/Indices option and selecting **Rebuild Index**.

The %SYS.Maint.Bitmap utility methods compress the bitmap extent index, as well as bitmap indexes and bitslice indexes. For further details, see Maintaining Bitmap Indexes.

Invoking the **%BuildIndices()** method builds an existing bitmap extent index in any of the following cases: the **%BuildIndices()** *pIndexList* argument is not specified (build all defined indexes); *pIndexList* specifies the bitmap extent index by name; or *pIndexList* specifies any defined bitmap index. See Building Indexes Programmatically.

2.3 Defining Indexes

There are two ways to define indexes:

- Defining Indexes Using DDL
- Defining Indexes Using a Class Definition, which includes:
  - Properties That Can Be Indexed
  - Indexes on Combinations of Properties
  - Index Collation
  - Using the Unique, PrimaryKey, and IdKey Keywords with Indexes
  - Storing Data with Indexes

- Indexing a NULL
- Indexing Collections
- Indexing Array Collections
- Indexing Data Type Properties with (ELEMENTS) and (KEYS)
- Indexing an Embedded Object (%SerialObject) Property

2.3.1 Defining Indexes Using DDL

If you are using DDL statements to define tables, you can also use the following DDL commands to create and remove indexes:

- CREATE INDEX
- DROP INDEX

The DDL index commands do the following:

1. They update the corresponding class and table definitions on which an index is being added or removed. The modified class definition is recompiled.
2. They add or remove index data in the database as needed: The CREATE INDEX command populates the index using the data currently stored within the database. Similarly, the DROP INDEX command deletes the index data (that is, the actual index) from the database.

2.3.2 Defining Indexes Using a Class Definition

You can add index definitions to a %Persistent class definition by editing the text of the class definition. An index is defined on one or more index property expressions optionally followed by one or more optional index keywords. It takes the form:

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

where:

- *index_name* is a valid identifier.
- *index_property_expression_list* is a list of the one or more comma-separated property expressions that serve as the basis for the index.
- *index_keyword_list* is an optional comma-separated list of index keywords, enclosed in square brackets. Used to specify the index Type for a bitmap or bitslice index. Also used to specify a Unique, IdKey, or PrimaryKey index. (An IdKey or PrimaryKey index is, by definition, also a Unique index.) The complete list of index keywords appears in the *Class Definition Reference*.

The *index_property_expression_list* argument consists of one or more index property expressions. An index property expression consists of:

- The name of the property to be indexed.
- An optional `(ELEMENTS)` or `(KEYS)` expression, which provide a means of indexing on collection subvalues. If the index property is not a collection, the user can use the **BuildValueArray()** method to produce an array containing keys and elements. For more information on keys and elements, see Indexing Collections.
- An optional collation expression. This consists of a collation name followed optionally by a list of one or more comma-separated collation parameters. You cannot specify an index collation for a Unique, IdKey, or PrimaryKey index. A

Unique or PrimaryKey index takes its collation from the property (field) that is being indexing. An IdKey index is always EXACT collation. For a list of valid collation names, see Collation Types.

Note: Adding an index to a class definition does not automatically build the index at compile time, unlike using the CREATE INDEX command. For information on building an index, see Building Indexes.

For example, the following class definition defines two properties and an index based on each of them:

Class Definition

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

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

A more complex index definition might be:

Class Member

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

2.3.2.1 Properties That Can Be Indexed

Properties that can be indexed are:

- Those that are stored in the database.
- Those that can be reliably derived from stored properties.

A property that can be reliably derived (and is not stored) must be defined with the SQLComputed keyword as true; the compute code of the property must be the only way to derive the property's value and the property cannot be set directly.

As a general rule, only derived properties defined as Calculated and SQLComputed can be indexed. There is, however, an exception for derived collections. That is, a collection defined with the SQLComputed and Transient keywords, but not the Calculated keyword, can be indexed.

If a property has a long text limit, data may be ingested in into the property, but the contents of the property may be too long to fit in the subscript of the index. The length limit may also manifest for composite indexes where the combination of multiple fields makes the property too long for the subscript. However, indexes on long strings generally have limited usefulness, as they are rarely identical to other values. Therefore, in these cases, an error message is provided and the data is not stored in the index. If you still would like a long text field to be included in an index, you can define TRUNCATE collation with a limit of 128 characters on the relevant properties.

Note: There must not be a sequential pair of vertical bars (||) within the values of any property used by an IdKey index, unless that property is a valid reference to an instance of a persistent class. This restriction is required by the InterSystems SQL internal mechanism. The use of || in IdKey properties can result in unpredictable behavior.

2.3.2.2 Indexes on Combinations of Properties

You can define indexes on combinations of two or more properties (fields). Within a class definition, use the On clause of the index definition to specify a list of properties, such as:

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);
}
```

An index on multiple properties may be useful if you need to perform queries that use a combination of field values, such as:

SQL

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

2.3.2.3 Index Collation

A Unique, PrimaryKey, or IdKey index cannot specify a collation type. For other types of indexes, each property specified in an index definition can optionally have a collation type. The index collation type should match the property (field) collation type when the index is applied.

1. If an index definition includes an explicitly specified collation for a property, the index uses that collation.
2. If an index definition does not include an explicitly specified collation for a property, the index uses the collation explicitly specified in the property definition.
3. If the property definition does not include an explicitly specified collation, then the index uses the collation that is the default for the property data type.

For example, the Name property is defined as a string, and therefore has, by default, SQLUPPER collation. If you define an index on Name, it takes, by default, the property's collation, and the index would also be defined with SQLUPPER. The property collation and the index collation match.

However, if a comparison applies a different collation, for example, `WHERE %EXACT(Name)=%EXACT(:invar)`, the property collation type in this usage no longer matches the index collation type. A mismatch between the property comparison collation type and the index collation type may cause the index to not be used. Therefore, in this case, you might wish to define the index for the Name property with collation EXACT. If an ON clause of a JOIN statement specifies a collation type, for example, `FROM Table1 LEFT JOIN Table2 ON %EXACT(Table1.Name) = %EXACT(Table2.Name)`, a mismatch between the property collation type specified here and the index collation type may cause InterSystems IRIS to not use the index.

The following rules govern collation matches between an index and a property:

- Matching collation types always maximize use of an index.
- A mismatch of collation types, where the property is specified with EXACT collation (as shown above) and the index has some other collation allow the index to be used, but its use is less effective than matching collation types.
- A mismatch of collation types, where the property collation is not EXACT and the property collation does not match the index collation, causes the index to not be used.

To explicitly specify a collation for a property in an index definition, the syntax is:

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

where

- *IndexName* is the name of the index

- *PropertyName* is the property being indexed
- *CollationName* is the type of collation being used for the index

For example:

Class Member

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

Different properties can have different collation types. For example, in the following example the F1 property uses SQLUPPER collation while F2 uses EXACT collation:

Class Member

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

For a list of recommended collation types, see Collation Types.

Note: An index specified as Unique, PrimaryKey, or IdKey cannot specify an index collation. The index takes its collation from the property collations.

2.3.2.4 Using the Unique, PrimaryKey, and IdKey Keywords with Indexes

As is typical with SQL, InterSystems IRIS supports the notions of a unique key and a primary key. InterSystems IRIS also has the ability to define an IdKey, which is one that is a unique record ID for the instance of a class (row of a table). These features are implemented through the Unique, PrimaryKey, and IdKey keywords:

- Unique — Defines a UNIQUE constraint on the properties listed in the index's list of properties. That is, only a unique data value for this property (field) can be indexed. Uniqueness is determined based on the property's collation. For example, if the property collation is EXACT, values that differ in letter case are unique; if the property collation is SQLUPPER, values that differ in letter case are not unique. However, note that the uniqueness of indexes is not checked for properties that are undefined. In accordance with the SQL standard, an undefined property is always treated as unique.
- PrimaryKey — Defines a PRIMARY KEY constraint on the properties listed in the index's list of properties.
- IdKey — Defines a unique constraint and specifies which properties are used to define the unique identity of an instance (row). An IdKey always has EXACT collation, even when it is of data type string.

The syntax of such keywords appears in the following example:

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 ];
}
```

Note: The IdKey, PrimaryKey, and Unique keywords are only valid with standard indexes. You cannot use them with bitmap or bitslice indexes.

It is also valid syntax to specify both the IdKey and PrimaryKey keywords together, such as:

Class Member

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

This syntax specifies that the IDPKIDX index is both the IdKey for the class (table), as well as its primary key. All other combinations of these keywords are redundant.

For any index defined with one of these keywords, there is a method that allows you to open the instance of the class where the properties associated with the index have particular values; for more information, see Opening an Instance by Index Key.

For more information on the IdKey keyword, see the IdKey page of the *Class Definition Reference*. For more information on the PrimaryKey keyword, see the PrimaryKey page of the *Class Definition Reference*. For more information on the Unique keyword, see the Unique page of the *Class Definition Reference*.

2.3.2.5 Storing Data with Indexes

You can specify that a copy of one or more data values be stored within an index using the index Data keyword:

Class Definition

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

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

In this case, the index, NameIDX, is subscripted by the collated (uppercase) value of the various Name values. A copy of the actual (uncollated) value of the Name is stored within the index. These copies are maintained when changes are made to the Sample.Person table through SQL or to corresponding the Sample.Person class or its instances through objects.

Maintaining a copy of data along within an index can be helpful in cases where you frequently perform selective (selecting a few rows out of many) or ordered searches that return a few columns out of many.

For example, consider the following query against the Sample.Person table:

SQL

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

The SQL Engine could decide to satisfy this request entirely by reading from the NameIDX and never reading the master data for the table.

Note: You cannot store data values with a bitmap index.

2.3.2.6 Indexing a NULL

If the data has a NULL (no data present) for an indexed field, the corresponding index represents this using an index null marker. By default, the index null marker value is -1E14. Use of an index null marker provides that null values collate before all non-null values. You can change the index null marker value for a specific field by using the INDEXNULL-MARKER property parameter.

2.3.2.7 Indexing Collections

When a property is indexed, the value that is placed in the index is the entire collated property value. For collections, it is possible to define index properties that correspond to the element and key values of the collection by appending (ELEMENTS) or (KEYS) to the property name. (ELEMENTS) and (KEYS) allow you to specify that multiple values are produced from a single property value and each of these sub-values is indexed. When the property is a collection, then the ELEMENTS

token references the elements of the collection by value and the KEYS token references them by position. When both ELEMENTS and KEYS are present in a single index definition. the index key value includes the key and associated element value.

For example, suppose there is an index based on FavoriteColors property of the Sample.Person class. The simplest form of an index on the items in this property's collection would be either of:

Class Member

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

or

Class Member

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

where `FavoriteColors(ELEMENTS)` refers to the elements of the FavoriteColors property and `FavoriteColors(KEYS)` refers to the keys of the FavoriteColors property. The general form is *propertyName*`(ELEMENTS)` or *propertyName*`(KEYS)`, where that collection's content is the set of elements contained in a property defined as a `List Of` or an `Array Of` some data type. For information on collections, see Working with Collections.

To index literal properties (described in Defining and Using Literal Properties), you can create an index value array as produced by a *propertyName***BuildValueArray()** method (described in the following section). As with collections proper, the `(ELEMENTS)` and `(KEYS)` syntax is valid with index value arrays.

If property-collection is projected as array, then the index must obey the following restrictions in order to be projected to the collection table. The index must include (KEYS). The index cannot reference any properties other than the collection itself and the object's ID value. If a projected index also defines DATA to be stored in the index, then the data properties stored must also be restricted to the collection and the ID. Otherwise the index is not projected. This restriction applies to an index on a collection property that is projected as an array; it does not apply to an index on a collection that is projected as a list. For further details, refer to Controlling the SQL Projection of Collection Properties.

Indexes that correspond to element or key values of a collection can also have all the standard index features, such as storing data with the index, index-specific collations, and so on.

InterSystems SQL can use a collection index by specifying the FOR SOME %ELEMENT predicate.

2.3.2.8 Indexing Data Type Properties with (ELEMENTS) and (KEYS)

For the purposes of indexing data type properties, you can also create index value arrays using the **BuildValueArray()** method. This method parses a property value into an array of keys and elements. It does this by producing a collection of element values derived from the value of the property with which it is associated. This method does not work on existing collection properties.

When you use **BuildValueArray()** to create an index value array, its structure is suitable for indexing.

The **BuildValueArray()** method has the name *propertyName***BuildValueArray()** and its signature is:

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

where

- The name of the **BuildValueArray()** method derives from the property name in the typical way for composite methods.
- The first argument is the property value.
- The second argument is an array that is passed by reference. This is an array containing key-element pairs where the array subscripted by the key is equal to the element.
- The method returns a %Status value.

2.3.2.9 Indexing Array Collections

To specify an index on an array collection property, you must include the keys in the index definition. For example:

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));
}
```

These keys identify the RowID of the array element's child table row. Without this key, the parent table does not project an index to the child table. Because this projection does not occur, INSERT operations into the parent table fail.

2.3.2.10 Indexing an Embedded Object (%SerialObject) Property

To index a property in an embedded object, you create an index in the persistent class referencing that embedded object. The property name must specify the name of the referencing field in the table (%Persistent class) and the property in the embedded object (%SerialObject), as shown in the following example:

Class Definition

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

Here *Home* is a property in Sample.Person that references the embedded object Sample.Address, which contains the *State* property, as shown in the following example:

Class Definition

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

Only the data values in the instance of the embedded object associated with the persistent class property reference are indexed. You cannot index a %SerialObject property directly. The SqlCategory for %Library.SerialObject (and all sub-classes of %SerialObject that do not define the SqlCategory explicitly) is STRING.

You can also define an index on an embedded object property using the SQL CREATE INDEX statement, as shown in the following example:

SQL

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

For more details, see Introduction to Serial Objects and Embedded Object (%SerialObject).

2.4 Summary of Index Types

InterSystems SQL provides a number of different index types that are useful in different circumstances. To optimize your schema, you should define the index that best serves your use case. The table below summarizes these index types; see the linked sections for further information about each type.

Note that adding any index increases both the amount of storage space a table takes up and the number of operations performed when updating or adding data.

Table 2–1:

Index Type	Description	Field Types	Example Use Cases
Standard	Groups rows that share a value in a specific column	Any	Efficient for most circumstances
Bitmap	For each unique value in a column, stores a bitstring that indicates which rows contain the value When a bitmap index is defined, a Bitmap Extent index is automatically created	Numeric	Queries that use: AND or OR for multiple conditions on a single table RANGE conditions
Bitmap Extent	An existence bitmap created automatically when a bitmap index is created or when a table is defined with CREATE TABLE	Numeric (see Bitmap)	Queries that use: COUNT commands
Bitslice	Converts numeric values to binary and stores that binary value as a bitmap	Numeric	Queries that use: SUM, COUNT, or AVG commands
Columnar	Stores a copy of the field's data in a compressed, vectorized format	Numeric; short strings with low cardinality	Queries that use: SUM or AVG commands RANGE conditions

2.5 Bitmap Indexes

A bitmap index is a special type of index that uses a series of bitstrings to represent the set of ID values that correspond to a given indexed data value.

Bitmap indexes have the following important features:

- Bitmaps are highly compressed: bitmap indexes can be significantly smaller than standard indexes. This reduces disk and cache usage considerably.
- Bitmaps operations are optimized for transaction processing: you can use bitmap indexes within tables with no performance penalty as compared with using standard indexes.

- Logical operations on bitmaps (counting, AND, and OR) are optimized for high performance.
- The SQL Engine includes a number of special optimizations that can take advantage of bitmap indexes.

The creation of bitmap indexes depends upon the nature of the table's unique identity field(s):

- If the table's ID field is defined as a single field with positive integer values, you can define a bitmap index for a field using this ID field. This type of table either uses a system-assigned unique positive integer ID, or uses an IdKey to define custom ID values where the IdKey is based on a single property with type %Integer and MINVAL > 0, or type %Numeric with SCALE = 0 and MINVAL > 0.
- If the table's ID field is not defined as single field with positive integer values (for example, a child table), you can define a %BID (bitmap ID) field that takes positive integers which acts as a surrogate ID field; this allows you to create bitmap indexes for fields in this table.

Subject to the restrictions listed below, bitmap indexes operate in the same manner as standard indexes. Indexed values are collated and you can index on combinations of multiple fields.

This page addresses the following topics related to bitmap indexes:

- Bitmap Index Operation
- Defining Bitmap Indexes Using DDL
- Defining Bitmap Indexes by Using a Class Definition
- Generating a Bitmap Extent Index
- Choosing an Index Type
- Restrictions on Bitmap Indexes
- Maintaining Bitmap Indexes
- SQL Manipulation of Bitmap Chunks

2.5.1 Bitmap Index Operation

Bitmap indexes work in the following way. Suppose you have a Person table containing a number of columns:

Figure 2–1: Person Table

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	...
...	...	...	...	...	...

Each row in this table has a system-assigned RowID number (a set of increasing integer values). A bitmap index uses a set of bitstrings (a string containing 1 and 0 values). Within a bitstring, the ordinal position of a bit corresponds to the RowID of the indexed table. For a given value, say where State is "NY", there is a string of bits with a 1 for every position that corresponds to a row containing "NY" and a 0 in every other position.

For example, a bitmap index on State might look like this:

Figure 2–2: State Bitmap Index

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

While an index on Age might look like this:

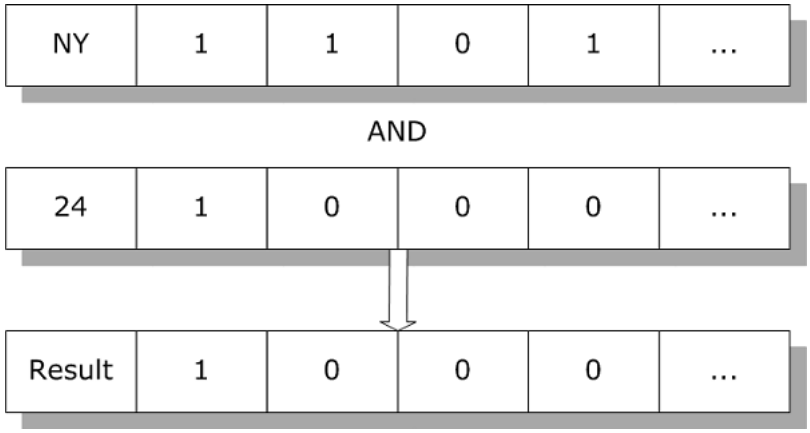
Figure 2–3: Age Bitmap Index

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

Note: The Age field shown here can be an ordinary data field or a field whose value can be reliably derived (Calculated and SQLComputed).

In addition to using bitmap indexes for standard operations, the SQL engine can use bitmap indexes to efficiently perform special set-based operations using combinations of multiple indexes. For example, to find all instances of Person that are 24 years old and live in New York, the SQL Engine can simply perform the logical AND of the Age and State indexes:

Figure 2–4: Using Multiple Indexes



The resulting bitmap contains the set of all rows that match the search criteria. The SQL Engine uses this to return data from these rows.

The SQL Engine can use bitmap indexes for the following operations:

- ANDing of multiple conditions on a given table.
- ORing of multiple conditions on a given table.
- RANGE conditions on a given table.
- COUNT operations on a given table.

2.5.2 Defining Bitmap Indexes Using DDL

If you are using DDL statements to define tables, you can also use the following DDL commands to create and remove bitmap indexes for a table with a positive integer ID:

- CREATE INDEX
- DROP INDEX

This is identical to creating standard indexes, except that you must add the BITMAP keyword to the CREATE INDEX statement:

SQL

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

2.5.3 Defining an IdKey Bitmap Index Using a Class Definition

If the table's ID is a field with values restricted to unique positive integers, you can add bitmap index definitions to a class definition using either the New Index Wizard or by editing the text of the class definition in the same way that you would create a standard index. The only difference is that you need to specify the index Type as being "bitmap":

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 Defining a %BID Bitmap Index Using a Class Definition

If the table's ID is not restricted to positive integers, you can create a %BID field to use to create bitmap index definitions. You can use this option for a table with an ID field of any datatype, as well as an IDKEY consisting of multiple fields (which includes child tables). A %BID bitmap can be created for either data storage type: a default structure table or a %Storage.SQL table. This feature is referred to as "Bitmaps for Any Table," or BAT.

To enable use of bitmap indexes on such a table, you must do the following:

1. Define the %BID field, or identify an existing field as the %BID field. The data type of this field must restrict values to unique positive integers. For example, in this table, the IDKey is a composite of two fields that are not restricted to positive integers. This makes the MyBID field, which has a positive integer data type (%Counter) a candidate for the %BID field.

   ### 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. Define the BIDField class parameter to identify the %BID field for the SQL compiler. Set its value to the SQLFieldName of the %BID field. For example:

   ### 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. Define the BATKey index. This index acts as the master map and data map for the SQL query processor. Typically, the %BID field is the first subscript of the BATKey index. The map data can then include the IDKEY fields and any additional properties that you want to have the fastest access to. You must set the index on the %BID field. For example:

   ### 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. Define the BATKey class parameter identify the BATKey index for the SQL compiler. Set its value to the SQLFieldName of the BATKey index. For example:

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. Define the %BID locator index, or identify an existing index as the %BID locator index. This index ties the %BID index to the IDKey fields of the table. For example:

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 */
}
```

This table now supports bitmap indexes. You can define bitmap indexes as needed using standard syntax. For example:
`Index RegionIDX On Region [Type = bitmap];`

Tables created in this way also support bitslice indexes, which can also be defined using standard syntax.

2.5.5 Generating a Bitmap Extent Index

A bitmap index requires a bitmap extent index. Defining a persistent class only generates a bitmap extent index if one or more bitmap indexes are defined. Therefore, when compiling a persistent class that contains a bitmap index, the class compiler generates a bitmap extent index if no bitmap extent index is defined for that class. Tables defined with the CREATE TABLE DDL statement automatically generate a bitmap extent index.

If you delete all bitmap indexes from the persistent class definition, the bitmap extent index is automatically deleted. However, if you rename the bitmap extent index (for example, using the **CREATE BITMAPEXTENT INDEX** command) deleting the bitmap index does not delete the bitmap extent index.

When building indexes for a class, the bitmap extent index is built either if you explicitly build the bitmap extent index, or if you build a bitmap index and the bitmap extent index is empty.

A class inherits its bitmap extent index from the primary superclass if it exists, either defined or generated. A bitmap extent index inherited from a primary superclass is considered to be a bitmap index and will trigger a bitmap extent index to be generated in the subclass, if no bitmap extent index is explicitly defined in that subclass.A bitmap extent index is defined as follows:

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 does not generate a bitmap extent index in a superclass based on future possibility. This means that InterSystems IRIS does not ever generate a bitmap extent index in a persistent class unless an index whose `type = bitmap` is present. A presumption that some future subclass might introduce an index with `type = bitmap` is not sufficient.

Note: Special care is required during the process of adding a bitmap index to a class on a production system (where users are actively using a particular class, compiling said class, and subsequently building the bitmap index structure for it). On such a system, the bitmap extent index may be populated in the interim between the compile completing and the index build proceeding. This can cause the index build procedure to not implicitly build the bitmap extent index, which leads to a partially complete bitmap extent index.

2.5.6 Choosing an Index Type

The following is a general guideline for choosing between bitmap and standard indexes. In general, use standard indexes for indexing on all types of keys and references:

- Primary key
- Foreign key
- Unique keys
- Relationships
- Simple object references

Otherwise, bitmap indexes are generally preferable (assuming that the table uses system-assigned numeric ID numbers).

Other factors:

- Separate bitmap indexes on each property usually have better performance than a bitmap index on multiple properties. This is because the SQL engine can efficiently combine separate bitmap indexes using AND and OR operations.
- If a property (or a set of properties that you really need to index together) has more than 10,000-20,000 distinct values (or value combinations), consider standard indexes. If, however, these values are very unevenly distributed so that a small number of values accounts for a substantial fraction of rows, then a bitmap index could be much better. In general, the goal is to reduce the overall size required by the index.

2.5.7 Restrictions on Bitmap Indexes

All bitmap indexes have the following restrictions:

- You cannot define a bitmap index on a UNIQUE column.
- You cannot store data values within a bitmap index.
- You cannot define a bitmap index on a field unless the SqlCategory of the ID field is INTEGER, DATE, POSIXTIME, or NUMERIC (with scale=0).
- For a table containing more than 1 million records, a bitmap index is less efficient than a standard index when the number of unique values exceeds 10,000. Therefore, for a large table it is recommended that you avoid using a bitmap index for any field that contains (or is likely to contain) more than 10,000 unique values; for a table of any size, avoid using a bitmap index for any field that is likely to contain more than 20,000 unique values. These are general approximations, not exact numbers.

You must create a %BID property to support bitmap indexes on a table that:

- Uses a non-integer field as the unique ID key.
- Uses a multi-field ID key.

- Is a child table within a parent-child relationship.

You can use the **$SYSTEM.SQL.Util.SetOption()** method `SET status=$SYSTEM.SQL.Util.SetOption("BitmapFriendlyCheck",1,.oldval)` to set a system-wide configuration parameter to check at compile time for this restriction, determining whether a defined bitmap index is allowed in a %Storage.SQL class. This check only applies to classes that use %Storage.SQL. The default is 0. You can use **$SYSTEM.SQL.Util.GetOption("BitmapFriendlyCheck")** to determine the current configuration of this option.

2.5.7.1 Application Logic Restrictions

A bitmap structure can be represented by an array of bit strings, where each element of the array represents a "chunk" with a fixed number of bits. Because undefined is equivalent to a chunk with all 0 bits, the array can be sparse. An array element that represents a chunk of all 0 bits need not exist at all. For this reason, application logic should avoid depending on the $BITCOUNT(str,0) count of 0-valued bits.

Because a bit string contains internal formatting, application logic should never depend upon the physical length of a bit string or upon equating two bit strings that have the same bit values. Following a rollback operation, a bit string is restored to its bit values prior to the transaction. However, because of internal formatting, the rolled back bit string may not equate to or have the same physical length as the bit string prior to the transaction.

2.5.8 Maintaining Bitmap Indexes

In a volatile table (one that undergoes many **INSERT** and **DELETE** operations) the storage for a bitmap index can gradually become less efficient. To maintain bitmap indexes, you can run the %SYS.Maint.Bitmap utility methods to compress the bitmap indexes, restoring them to optimal efficiency. You can use the **OneClass()** method to compress the bitmap indexes for a single class or the **Namespace()** method to compress the bitmap indexes for an entire namespace. These maintenance methods can be run on a live system.

The results of running the %SYS.Maint.Bitmap utility methods are written to the process that invoked the method. These results are also written to the class %SYS.Maint.BitmapResults.

2.5.9 SQL Manipulation of Bitmap Chunks

InterSystems SQL provides the following extensions to directly manipulate bitmap indexes:

- %CHUNK function
- %BITPOS function
- %BITMAP aggregate function
- %BITMAPCHUNK aggregate function
- %SETINCHUNK predicate condition

All of these extensions follow the InterSystems SQL conventions for bitmap representation, representing a set of positive integers as a sequence of bitmap chunks, of up to 64,000 integers each.

These extensions enable easier and more efficient manipulation of certain conditions and filters, both within a query and in embedded SQL. In embedded SQL they enable simple input and output of bitmaps, especially at the single chunk level. They support the processing of complete bitmaps, which are handled by %BITMAP() and the %SQL.Bitmap class. They also enable bitmap processing for non-RowID values, such as foreign key values, parent-reference of a child table, either column of an association, etc.

For example, to output the bitmap for a specified chunk:

SQL

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

To output all the chunks for the whole table:

SQL

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

2.5.9.1 %CHUNK function

%CHUNK(f) returns the chunk assignment for a bitmap indexed field *f* value. This is calculated as f\64000+1. **%CHUNK(f)** for any field or value *f* that is not a bitmap indexed field always returns 1.

2.5.9.2 %BITPOS function

%BITPOS(f) returns the bit position assigned to a bitmap indexed field *f* value within its chunk. This is calculated as f#64000+1 . **%BITPOS(f)** for any field or value *f* that is not a bitmap indexed field returns 1 more than its integer value. A string has an integer value of 0.

2.5.9.3 %BITMAP aggregate function

The aggregate function **%BITMAP(f)** combines many *f* values into a %SQL.Bitmap object, in which the bit corresponding to *f* in the proper chunk is set to 1 for each value *f* in the result set. *f* in all of the above would normally be a positive integer field (or expression), usually (but not necessarily) the RowID.

2.5.9.4 %BITMAPCHUNK aggregate function

The aggregate function %BITMAPCHUNK(f) combines many values of the field *f* into an InterSystems SQL standard bitmap string of 64,000 bits, in which bit f#64000+1=%BITPOS(f) is set to 1 for each value f in the set. Note that the bit is set in the result regardless of the value of %CHUNK(f) . %BITMAPCHUNK() yields NULL for the empty set, and like any other aggregate it ignores NULL values in the input.

2.5.9.5 %SETINCHUNK predicate condition

The condition (f %SETINCHUNK bm) is true if and only if ($BIT(bm,%BITPOS(f))=1) . bm could be any bitmap expression string, e.g. an input host variable :bm, or the result of a %BITMAPCHUNK() aggregate function, etc. Note that the <bm> bit is checked regardless of the value of %CHUNK(f) . If <bm> is not a bitmap or is NULL, the condition returns FALSE. (f %SETINCHUNK NULL) yields FALSE (not UNKNOWN).

2.6 Bitslice Indexes

A bitslice index is used for a numeric data field when that field is used for certain numeric operations. A bitslice index represents each numeric data value as a binary bit string. Rather than indexing a numeric data value using a boolean flag (as in a bitmap index), a bitslice index represents each value in binary and creates a bitmap for each digit in the binary value to record which rows have a 1 for that binary digit. This is a highly specialized type of index that can substantially improve performance of the following operations:

- **SUM**, **COUNT**, or **AVG** aggregate calculations. (A bitslice index is *not* used for **COUNT(*)** calculations.) Bitslice indexes are not used for other aggregate functions.
- A field specified in a `TOP n ... ORDER BY field` operation.

- A field specified in a range condition operation, such as `WHERE field > n` or `WHERE field BETWEEN lownum AND highnum`.

The SQL optimizer determines whether a defined bitslice index should be used. Commonly, the optimizer only uses a bitslice index when a substantial number of rows (thousands) are being processed.

You can create a bitslice index for a string data field, but the bitslice index will represent these data values as canonical numbers. In other words, any non-numeric string, such as “abc” will be indexed as 0. This type of bitslice index could be used to rapidly **COUNT** records that have a value for a string field and not count those that are NULL.

In the following example, Salary would be a candidate for a bitslice index:

SQL

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

A bitslice index can be used for an aggregate calculation in a query that uses a WHERE clause. This is most effective if the WHERE clause is inclusive of a large number of records. In the following example, the SQL optimizer would probably use a bitslice index on Salary, if defined; if so, it would also use a bitmap index on Region, either using a defined bitmap or generating a bitmap tempfile for Region:

SQL

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

However, a bitslice index is not used when the WHERE condition cannot be satisfied by an index, but must be performed by reading the table that contains the field being aggregated. The following example would not use the bitslice index on Salary:

SQL

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

A bitslice index can be defined for any field containing numeric values. InterSystems SQL uses a scale parameter to convert fractional numbers into bitstrings, as described in the ObjectScript $FACTOR function. A bitslice index can be defined for a field of data type string; in this case, non-numeric string data values are treated as 0 for the purposes of the bitslice index.

A bitslice index can be defined for fields in a table that has system-assigned row Ids with positive integer values, or a table defined with a %BID property to support bitmap (and bitslice) indexes.

A bitslice index can only be defined for a single field name, not a concatenation of multiple fields. You cannot specify a WITH DATA clause.

The following example compares a bitslice index to a bitmap index. If you create a bitmap index for values 1, 5, and 22 for rows 1, 2, and 3, it creates an index for the values:

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

If you create a bitslice index for values 1, 5, and 22 for rows 1, 2, and 3, it first converts the values to bit values:

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

It then creates an index for the bits:

```
^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"
```

In this example, the value 22 in a bitmap index required setting 1 global node; the value 22 in a bitslice index required setting 3 global nodes.

Note that an **INSERT** or **UPDATE** requires setting a bit in all *n* bitslices, rather than setting a single bitstring. These additional global set operations can affect performance of **INSERT** and **UPDATE** operations that involve populating bitslice indexes. Populating and maintaining a bitslice index using **INSERT**, **UPDATE**, or **DELETE** operations is slower than populating a bitmap index or a regular index. Maintaining multiple bitslice indexes, and/or maintaining a bitslice index on a field that is frequently updated may have a significant performance cost.

In a volatile table (one that undergoes many **INSERT**, **UPDATE**, and **DELETE** operations) the storage for a bitslice index can gradually become less efficient. The %SYS.Maint.Bitmap utility methods compress both bitmap indexes and bitslice indexes, restoring efficiency. For further details, see Maintaining Bitmap Indexes.

2.7 Columnar Indexes

A columnar index is used for a field that is frequently queried but whose table has an underlying row storage structure. By default, each row of a table is stored as a **$LIST** in a separate global subscript. A columnar index stores data for a specific field in a compressed, vectorized format.

To define a columnar index using InterSystems SQL DDL, use the `CREATE COLUMNAR INDEX` syntax of CREATE INDEX:

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

To define a columnar index in a persistent class, specify the `type = columnar` keyword on the index you define:

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

This sample DDL shows how to define a columnar index on a specific column in a table:

```
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)
```

Suppose you are performing an **AVG** aggregate calculation on the Amount column of this table and filtering the result to include only deposit amounts.

SQL

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

This calculation requires loading each **$LIST** global into memory when you need only a subset of rows (`WHERE Type = 'Deposit'`) for a single column (`Amount`). Performing an **AVG** calculation on a field with a columnar index, the query plan accesses this information from the index rather than directly from the **$LIST** globals.

A columnar index may not be defined on a serial property or subclass. However, you may define an index on a non-serial property of a subclass, such as an %Integer.

A columnar index is similar to a bitmap index but is slightly less efficient for equality conditions. The bitmap index already has the bitstrings for each value, whereas a columnar index takes a vectorized operation to get them from the columnar index. A columnar index is often more efficient for range conditions. With a bitmap index, multiple bitstrings need to be combined, whereas in a columnar index, the same computation can be carried it in a single vectorized operation.

For more details on columnar indexes and defining table storage layouts, see Choose an SQL Table Storage Layout.

2.8 Building Indexes

You can build/re-build indexes as follows:

- Using the BUILD INDEX SQL command to build specified indexes, or build all indexes defined for a table, a schema, or the current namespace. This option is safe to use on a live system.
- Using the Management Portal to rebuild all of the indexes for a specified class (table).
- Using the **%BuildIndices()** (or **%BuildIndicesAsync()**) method.

The preferred way of building indexes systems is to use the BUILD INDEX SQL command. Building an index does the following:

1. Removes the current contents of the index.
2. Scans (reads every row) of the main table and adds index entries for each row in the table. As appropriate, low-level optimizations with respect to parallel execution and efficient batch sorting are applied using $SortBegin and $SortEnd.

If you use BUILD INDEX on a live system, the index is temporarily labeled as not selectable, meaning that queries cannot use the index while it is being built. Note that this will impact the performance of queries that use the index.

2.8.1 Building Indexes with BUILD INDEX

After creating an index at the class or DDL level, you should build it using the BUILD INDEX command. This statement can be used to build all of the indexes in a namespace, all the indexes in a schema, or only the indexes specified in the command. By default, it acquires an extent lock on each table prior to building its indexes and releases it when it has finished, making it safe to use on an active system. Queries will not be able to use an index while it is being built. Any data that is inserted or updated in the table using the INSERT or UPDATE commands while the BUILD INDEX command is running will be included in the building process.

BUILD INDEX uses the journaling setting for the current process to log any errors. You can turn off locking and journaling behavior by supplying the %NOLOCK and %NOJOURN options, respectively.

The following examples build indexes for the `MyApp.Salesperson` class:

Class Member

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

The first statement builds all of the indexes for the specified table (or class) name. The second statement builds only the NameIdx and SSNKey indexes.

2.8.2 Building Indexes with the Management Portal

You can build existing indexes (rebuild indexes) for a table by doing the following:

1. From the Management Portal select **System Explorer**, then **SQL**. Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces. After selecting a namespace, select the **Schema** drop-down list on the left side of the screen. This displays a list of the schemas in the current namespace with boolean flags indicating whether there are any tables or any views associated with each schema.

2. Select a schema from this list; it appears in the **Schema** box. Just above it is a drop-down list that allows you to select Tables, System Tables, Views, Procedures, or All of these that belong to the schema. Select either Tables or All, then open the Tables folder to list the tables in this schema. If there are no tables, opening the folder displays a blank page. (If you have not selected Tables or All, opening the Tables folder lists the tables for the entire namespace.)
3. Select one of the listed Tables. This displays the **Catalog Details** for the table.
   - To rebuild all indexes: click the **Actions** drop-down list and select **Rebuild Table's Indices**.
   - To rebuild a single index: click the **Indices** button to display the existing indexes. Each listed index has the option to **Rebuild Index**.

CAUTION: Do not rebuild indexes in this manner while the table's data is being accessed by other users. To rebuild indexes on an active system, you should use BUILD INDEX or, alternatively, the programmatic approach.

2.8.3 Building Indexes Programmatically

You can also use the **%BuildIndices()** and **%BuildIndicesAsync()** methods provided by the %Persistent class for the table to build indexes. Note that these methods are only provided for classes that use InterSystems IRIS default storage structure. Calling these methods requires that at least one index definition has been added to a specified class and compiled. Read about the methods more fully in the class reference pages linked below.

- **%Library.Persistent.%BuildIndices()**: **%BuildIndices()** executes as a background process but the caller has to wait for **%BuildIndices()** to complete before receiving control back.
- **%Library.Persistent.%BuildIndicesAsync()**: **%BuildIndicesAsync()** initiates **%BuildIndices()** as a background process and the caller immediately receives control back. The first argument to **%BuildIndicesAsync()** is the *queueToken* output argument. The remaining arguments are the same as **%BuildIndices()**.

2.9 Index Validation

You can validate indexes using the either of the following methods:

- **$SYSTEM.OBJ.ValidateIndices()** validates the indexes for a table, and also validates any indexes in collection child tables for that table.
- **%Library.Storage.%ValidateIndices()** validates the indexes for a table. Collection child table indexes must be validated with separate **%ValidateIndices()** calls.

Both methods check the data integrity of one or more indexes for a specified table, and optionally correct any index integrity issues found. They perform index validation in two steps:

1. Confirm that an index entity is properly defined for every row (object) in the table (class).
2. Traverse each index and for every entry indexed, make sure there is a value and matching entry in the table (class).

If either method finds discrepancies, it can optionally correct the index structure and/or contents. It can validate, and optionally correct, standard indexes, bitmap indexes, bitmap extent indexes, and bitslice indexes. By default, both methods validate indexes, but do not correct indexes.

%ValidateIndices() can only be used to correct an index on a READ and WRITE active system if **SetMapSelectability()** is used and the **%ValidateIndices()** arguments include both *autoCorrect*=1 and *lockOption*>0. Because **%ValidateIndices()** is significantly slower, **%BuildIndices()** is the preferred method for building indexes on an active system.

%ValidateIndices() is commonly run from the Terminal. It displays output to the current device. This method can be applied to a specified %List of index names, or to all indexes defined for the specified table (class). It operates only on those indexes that originated in specified class; if an index originated in a superclass, that index can be validated by calling **%ValidateIndices()** on the superclass. It is not supported for READONLY classes.

2.9.1 Validating Indexes on Sharded Classes

%ValidateIndices() is supported for sharded classes and for shard-master class tables (Sharded=1). You can invoke **%ValidateIndices**, either directly as a class method, or from **$SYSTEM.OBJ.ValidateIndices** on the shard master class. Index validation is then performed on the shard local class on each shard, and the results are returned to the caller on the shard master. When using **%ValidateIndices()** on a sharded class, the verbose flag is forced to 0. There is no output to the current device. Any issues found/corrected are returned in the byreference errors() array.

2.10 Using Indexes in Query Processing

Indexing provides a mechanism for optimizing queries by maintaining a sorted subset of commonly requested data. Determining which fields should be indexed requires some thought: too few or the wrong indexes and key queries will run too slowly; too many indexes can slow down **INSERT** and **UPDATE** performance (as the index values must be set or updated).

2.10.1 What to Index

To determine if adding an index improves query performance, run the query from the Management Portal SQL interface and note in Performance the number of global references. Add the index and then rerun the query, noting the number of global references. A useful index should reduce the number of global references. You can prevent use of an index by using the %NOINDEX keyword as preface to a WHERE clause or ON clause condition.

You should index fields (properties) that are specified in a JOIN. A LEFT OUTER JOIN starts with the left table, and then looks into the right table; therefore, you should index the field from the right table. In the following example, you should index T2.f2:

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

An INNER JOIN should have indexes on both ON clause fields.

Run Show Plan and follow to the first map. If the first bullet item in the Query Plan is “Read master map”, or the Query Plan calls a module whose first bullet item is “Read master map”, the query first map is the master map rather than an index map. Because the master map reads the data itself, rather than an index to the data, this almost always indicates an inefficient Query Plan. Unless the table is relatively small, you should create an index so that when you rerun this query the Query Plan first map says “Read index map.”

You should index fields that are specified in a WHERE clause equal condition.

You may wish to index fields that are specified in a **WHERE** clause range condition, and fields specified in **GROUP BY** and **ORDER BY** clauses.

Under certain circumstances, an index based on a range condition could make a query slower. This can occur if the vast majority of the rows meet the specified range condition. For example, if the query clause `WHERE Date < CURRENT_DATE` is used with a database in which most of the records are from prior dates, indexing on Date may actually slow down the query. This is because the Query Optimizer assumes range conditions will return a relatively small number of rows, and optimizes for this situation. You can determine if this is occurring by prefacing the range condition with %NOINDEX and then run the query again.

If you are performing a comparison using an indexed field, the field as specified in the comparison should have the same collation type as it has in the corresponding index. For example, the Name field in the WHERE clause of a **SELECT** or in the ON clause of a **JOIN** should have the same collation as the index defined for the Name field. If there is a mismatch between the field collation and the index collation, the index may be less effective or may not be used at all. For further details, refer to Index Collation.

For details on how to create an index and the available index types and options, refer to the CREATE INDEX command, and Defining and Building Indexes.

2.10.2 Index Configuration Options

The following system-wide configuration methods can be used to optimize use of indexes in queries:

- To use the PRIMARY KEY as the IDKey index, set the **$SYSTEM.SQL.Util.SetOption()** method, as follows: `SET status=$SYSTEM.SQL.Util.SetOption("DDLPKeyNotIDKey",0,.oldval)`. The default is 1.
- To use indexes for **SELECT DISTINCT** queries set the **$SYSTEM.SQL.Util.SetOption()** method, as follows: `SET status=$SYSTEM.SQL.Util.SetOption("FastDistinct",1,.oldval)`. The default is 1.

For further details, refer to SQL and Object Settings Pages listed in *System Administration Guide*.

2.10.3 Using %ALLINDEX, %IGNOREINDEX, and %NOINDEX

The FROM clause supports the %ALLINDEX and %IGNOREINDEX *optimize-option* keywords as hints. These *optimize-option* keywords govern all index use in the query.

You can use the %NOINDEX condition-level hint to specify exceptions to the use of an index for a specific condition. The %NOINDEX hint is placed in front of each condition for which no index should be used. For example, `WHERE %NOINDEX hiredate < ?`. This is most commonly used when the overwhelming majority of the data is selected (or not selected) by the condition. With a less-than (<) or greater-than (>) condition, use of the %NOINDEX condition-level hint is often beneficial. With an equality condition, use of the %NOINDEX condition-level hint provides no benefit. With a join condition, %NOINDEX is supported for ON clause joins.

The %NOINDEX keyword can be used to override indexing optimization established in the **FROM** clause. In the following example, the %ALLINDEX optimization keyword applies to all condition tests except the E.Age condition:

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 Analyzing Index Usage

There are two tools you can use to analyze the usage of indexes you have defined by SQL cached queries.

- The Management Portal Index Analyzer SQL performance tool.
- The %SYS.PTools.UtilSQLAnalysis methods **indexUsage()**, **tableScans()**, **tempIndices()**, **joinIndices()**, and **outlierIndices()**.

2.11.1 Index Analyzer

You can analyze index usage for SQL queries from the Management Portal using either of the following:

- Select **System Explorer**, select **Tools**, select **SQL Performance Tools**, then select **Index Analyzer**.
- Select **System Explorer**, select **SQL**, then from the **Tools** drop-down menu select **Index Analyzer**.

The **Index Analyzer** provides an SQL Statement Count display for the current namespace, and five index analysis report options.

2.11.1.1 SQL Statement Count

At the top of the **SQL Index Analyzer** there is an option to count all SQL statements in the namespace. Press the **Gather SQL Statements** button. The **SQL Index Analyzer** displays "Gathering SQL statements ...." while the count is in progress, then "Done!" when the count is complete. SQL statements are counted in three categories: a Cached Query count, a Class Method count, and a Class Query count. These counts are for the entire current namespace, and are not affected by the **Schema Selection** option. The corresponding method is **getSQLStmts()** in the %SYS.PTools.UtilSQLAnalysis class.

You can use the **Purge Statements** button to delete all gathered statements in the current namespace. This button invokes the **clearSQLStatements()** method.

2.11.1.2 Report Options

You can either examine reports for the cached queries for a selected schema in the current namespace, or (by not selecting a schema) examine reports for all cached queries in the current namespace. You can skip or include system class queries, INSERT statements, and/or IDKEY indexes in this analysis. The schema selection and skip option check boxes are user customized.

The index analysis report options are:

- **Index Usage**: This option takes all of the cached queries in the current namespace, generates a Show Plan for each and keeps a count of how many times each index is used by each query and the total usage for each index by all queries in the namespace. This can be used to reveal indexes that are not being used so they can either be removed or modified to make them more useful. The result set is ordered from least used index to most used index.
- **Queries with Table Scans**: This option identifies all queries in the current namespace that do table scans. Table scans should be avoided if possible. A table scan can't always be avoided, but if a table has a large number of table scans, the indexes defined for that table should be reviewed. Often the list of table scans and the list of temp indexes will overlap; fixing one will remove the other. The result set lists the tables from largest Block Count to smallest Block Count. A Show Plan link is provided to display the Statement Text and Query Plan.
- **Queries with Temp Indices**: This option identifies all queries in the current namespace that build temporary indexes to resolve the SQL. Sometimes the use of a temp index is helpful and improves performance, for example building a small index based on a range condition that InterSystems IRIS can then use to read the in order. Sometimes a temp index is simply a subset of a different index and might be very efficient. Other times a temporary index degrades performance, for example scanning the master map to build a temporary index on a property that has a condition. This situation indicates that a needed index is missing; you should add an index to the class that matches the temporary index. The result set lists the tables from largest Block Count to smallest Block Count. A Show Plan link is provided to display the Statement Text and Query Plan.
- **Queries with Missing JOIN Indices**: This option examines all queries in the current namespace that have joins, and determines if there is an index defined to support that join. It ranks the indexes available to support the joins from 0 (no index present) to 4 (index fully supports the join). Outer joins require an index in one direction. Inner joins require an index in both directions. By default, the result set only contains rows that have a JoinIndexFlag < 4. JoinIndexFlag=4 means there is an index that fully supports the join.

- **Queries with Outlier Indices**: This option identifies all queries in the current namespace that have outliers, and determines if there is an index defined to support that outlier. It ranks the indexes available to support the outlier from 0 (no index present) to 4 (index fully supports the outlier). By default, the result set only contains rows that have a OutlierIndexFlag < 4. OutlierIndexFlag=4 means there is an index that fully supports the outlier.

When you select one of these options, the system automatically performs the operation and displays the results. The first time you select an option or invoke the corresponding method, the system generates the results data; if you select that option or invoke that method again, InterSystems IRIS redisplays the same results. To generate new results data you must use the **Gather SQL Statements** button to reinitialize the Index Analyzer results tables. Changing the **Skip all system classes and routines** or **Skip INSERT statements** check box option also reinitializes the Index Analyzer results tables. To generate new results data for the %SYS.PTools.UtilSQLAnalysis methods, you must invoke **getSQLStmts()** to reinitialize the Index Analyzer results tables.

2.12 Listing Indexes

The INFORMATION.SCHEMA.INDEXES persistent class displays information about all column indexes in the current namespace. It returns one record for each indexed column. It provides a number of index properties, including the name of the index, table name, and column name that the index maps to. Each column record also provides the ordinal position of that column in the index map; this value is 1 unless the index maps to multiple columns. It also provides the boolean properties PRIMARYKEY and NONUNIQUE (0=index value must be unique).

The following example returns one row for each column that participates in an index for all non-system indexes in the current namespace:

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 '%'
```

You can list indexes for a selected table using the Management Portal SQL interface Catalog Details **Maps/Indices** option. This displays one line for each index, and displays index information not provided by INFORMATION.SCHEMA.INDEXES.

2.13 Open, Exists, and Delete Methods

The InterSystems IRIS indexing facility supports the following operations:

- Opening an Instance by Index Key
- Checking If an Instance Exists
- Deleting an Instance

2.13.1 Opening an Instance by Index Key

For ID key, primary key, or unique indexes, the *indexname***Open()** method (where *indexname* is the name of the index) allows you to open the object whose index property value or values match supplied value or values. Because this method has one argument corresponding to each property in the index, the method has three or more arguments:

- The first argument(s) each correspond to the properties in the index.

- The penultimate argument specifies the concurrency value with which the object is to be opened (with the available concurrency settings listed in Object Concurrency).
- The final argument can accept a %Status code, in case the method fails to open an instance.

The method returns an OREF if it locates a matching instance.

For example, suppose that a class includes the following index definition:

Class Member

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

then, if the referenced object has been stored to disk and has a unique ID value, you can invoke the method as follows:

ObjectScript

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

Upon successful completion, the method has set the value of *person* to the OREF of the instance of Sample.Person whose SSN property has a value of `111-22-3333`.

The second argument to the method specifies the concurrency value, which here is 2 (shared). The third argument holds an optional %Status code; if the method does not find an object that matches the supplied value, then an error message is written to the status parameter *sc*.

This method is implemented as the **%Compiler.Type.Index.Open()** method; this method is analogous to the **%Persistent.Open()** and **%Persistent.OpenId()** methods, except that it uses the properties in the index definition instead of the OID or ID argument.

2.13.2 Checking If an Instance Exists

The *indexname***Exists()** method (where *indexname* is the name of the index) checks if an instance exists with the index property value or values specified by the method's arguments. The method has one argument corresponding to each property in the index; its final, optional argument can receive the object's ID, if one matches the supplied value(s). The method returns a boolean, indicating success (1) or failure (0). This method is implemented as the **%Compiler.Type.Index.Exists()** method.

For example, suppose that a class includes the following index definition:

Class Member

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

then, if the referenced object has been stored to disk and has a unique ID value, you can invoke the method as follows:

ObjectScript

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

Upon successful completion, *success* equals 1 and *id* contains the ID matching the object that was found.

This method returns values for all indexes except:

- bitmap indexes, or a bitmap extent index.
- when the index includes an `(ELEMENTS)` or `(KEYS)` expression. For more information on such indexes, see Indexing Collections.

2.13.3 Deleting an Instance

The *indexname***Delete()** method (where *indexname* is the name of the index) is meant for use with a Unique, PrimaryKey, and or IdKey index; it deletes the instance whose key value matches the supplied key property/column values. There is one optional argument, which you can use to specify a concurrency setting for the operation. The method returns a %Status code. It is implemented as the **%Compiler.Type.Index.Delete()** method.

3

Choose an SQL Table Storage Layout

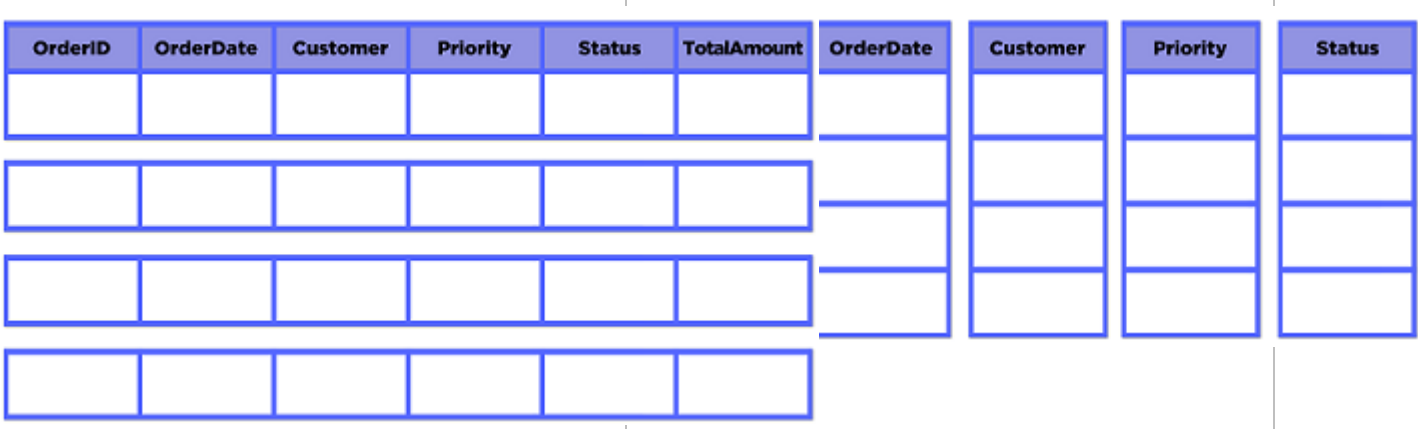
In InterSystems IRIS®, a relational table, such as the one shown here, is a logical abstraction. It does not reflect the underlying physical storage layout of the data.

OrderID	OrderDate	Customer	Priority	Status	TotalAmount

Using the flexibility inherent to globals, the lower-level InterSystems IRIS storage structure, you can specify whether to store the data in *rows*, *columns*, or a mixture of both. Depending on the size of your data and the nature of your queries and transactions, making the right storage layout choice can increase query performance or transaction throughput by an order of magnitude.

The choice of storage format has no effect on how you author your queries and other SQL statements, such as INSERTs. It is complementary to other table-level storage options such as sharding, which can further improve query performance for large tables. You can also define indexes on tables of any storage layout to gain additional performance benefits. For more details, see the Indexes on Storage Layouts section.

This table summarizes the row-based and column-based (*columnar*) storage formats.

Row Storage (Default)	Columnar Storage
Primary data is stored in one global. Each row of data is stored in a separate global subscript, using a list encoding that supports elements with different data types. In general, transactions affect individual rows and process efficiently on data stored by row, but analytical queries might be slower.	Primary data is stored in one global per column. Sequences of 64,000 data elements are stored in separate global subscripts. Data is encoded using a vector encoding that is optimized for storing elements of the same data type. In general, analytical queries run quickly but transactions might be slower.
	
Use row storage for: • Online transaction processing (OLTP), where you are processing transactional data in real time. • Frequent inserts, updates, and deletes of the data. • Queries where you want to select entire rows at a time and materialize them quickly.	Use columnar storage for: • Online analytical processing (OLAP), where you are filtering and aggregating data in specific columns to perform analytical queries. • Data in which updates, inserts, and deletes are infrequent or done in bulk, such as by using LOAD DATA.

Choosing a storage layout is not an exact science. You might need to experiment with multiple layouts and run multiple query tests to find the optimal one. For more of an overview on deciding between row and columnar storage layouts, along with sample use cases for choosing each layout, see the What Is Columnar Storage? video.

Note: Once you define a storage layout for your data, you currently cannot change it without reloading the data.

3.1 Row Storage Layout

When you define a table, either in the InterSystems SQL DDL language or in a persistent class, InterSystems IRIS defaults to using row storage.

3.1.1 Define Row Storage Table Using DDL

In InterSystems SQL DDL, the CREATE TABLE command defines tables in a row storage layout by default. **CREATE TABLE** does provide an optional `WITH STORAGETYPE = ROW` clause that you can specify after the column definitions, but it can be omitted. These two syntaxes are equivalent:

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


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

The following **CREATE TABLE** command creates a table of bank transactions. The table contains columns for the account number, transaction date, transaction description, transaction amount, and transaction type (for example, "deposit", "withdrawal", or "transfer"). Since the WITH STORAGETYPE clause is omitted, the table defaults to row storage.

SQL

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

3.1.2 Define Row Storage Table Using a Persistent Class

As with tables created using DDL, tables created by using a persistent class also use the row storage layout by default. You can optionally define the STORAGEDEFAULT parameter with the value "row" or the empty string (""), but both can be omitted. These syntaxes are equivalent:

```
Parameter STORAGEDEFAULT = "row";


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

The following persistent class shows the definition of a row storage table similar to the DDL-defined table created in the previous section. The USEEXTENTSET parameter organizes the table storage into a more efficient set of globals. The bitmap extent index creates an index of all IDs in the extent set, which makes counting and other operations more efficient. When you define a table using DDL commands, InterSystems SQL applies these settings automatically and includes them in the projected persistent class. For more details, see Defining a Table by Creating a Persistent Class.

```
  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 ];
  }
```

Note: The DEFAULT in the parameter name STORAGEDEFAULT implies that InterSystems IRIS uses this parameter value as the default when generating storage definition entries for this class. As with most class parameters that impact storage, such as USEEXTENTSET and DEFAULTGLOBAL, this value is considered only when generating storage, upon the initial compilation of the class or when adding new properties. Changing these parameters for a class that already has a storage definition (saved in a Storage XData block) has no effect on that storage, including on data already stored for the class extent.

3.1.3 Row Storage Details

In a table with row storage, all data is stored in a single global. Each subscript of this global contains a $LIST value that stores the data for a single row of table column values. The **$LIST** data type stores elements of varying types and encodes empty string (' ') and NULL values efficiently. These characters make **$LIST** suitable for storing rows of data, where columns usually have varying types and might contain NULL values.

Suppose the `BankTransaction` table from the previous section contains these transaction records:

SQL

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

AccountNumber	TransactionDate	Description	Amount	Type
10001234	02/22/2022	Deposit to Savings	40.00	Deposit
10001234	03/14/2022	Payment to Vendor	-20.00	Withdrawal
10002345	07/30/2022	Transfer to Checking	-25.00	Transfer
10002345	08/13/2022	Deposit to Savings	30.00	Deposit

You can optionally examine the global storage structure from the Management Portal by clicking **System Explorer** and then **Globals**. In the namespace containing the table, you can then select **Show SQL Table Name** and find the Data/Master global corresponding to your table. For more details, see Managing Globals. This code shows a sample Data/Master global that is representative for most standard tables on InterSystems IRIS.

```
^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` is the name of the global. The name shown here is for illustrative purposes. In tables created using DDL, or in a persistent class with the USEEXTENTSET=1 parameter specified, InterSystems IRIS generates more efficient, hashed globals with names such as `^EW3K.Cku2.1`. If a persistent class table does not specify USEEXTENTSET=1, then the global has a name of the format `^TableNameD`. In the projected persistent class for an SQL table, the global is stored in the `<DataLocation>` element of the `Storage` class member. For example:

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

- The top-level global node's value is the value of the highest subscript in the table, in this case 4. This integer subscript is used as the row ID.
- Each global subscript is a **$LIST** containing the column values for one row. The order of element values is defined in the storage definition and usually corresponds to column order. In this example, the second element of each row corresponds to the `TransactionDate` column, which stores the data in $HOROLOG format (number of days since December 31, 1840).

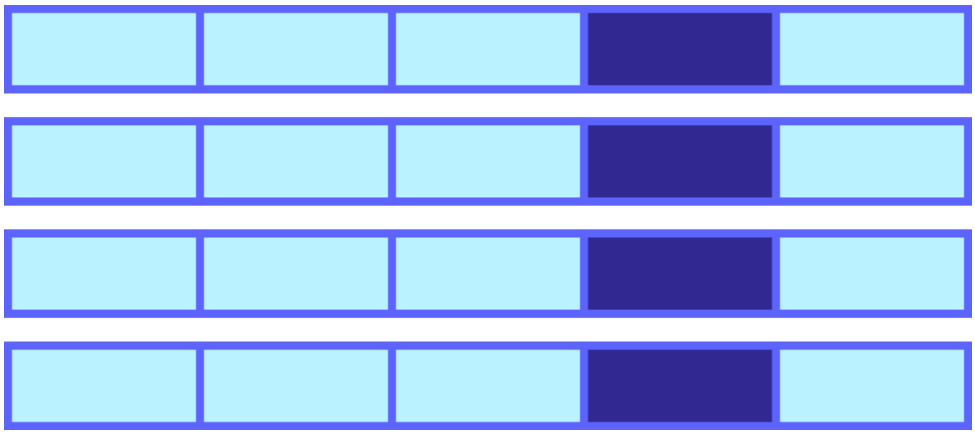
3.1.4 Analytical Query Processing with Row Storage

To show how row storage can be less efficient for analytical processing, suppose you query the average size of all transaction amounts in the `BankTransaction` table:

SQL

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

With this query, only the values in the `Amount` column are relevant. However, to access these values, the query must load each row into memory entirely, because the smallest unit of storage that InterSystems IRIS can read is a global node.



To check whether a query accesses each row individually, you can analyze the query execution plan. In the Management Portal, select **System Explorer** then **SQL** then **Show Plan**. Alternatively, use the **EXPLAIN** *query* SQL command. If the query plan includes a statement such as “Read master map ... looping on IDKEY”, then the query reads each row of the table, regardless of how relevant each column value is to the query.

To improve analytical query performance on tables with row storage, you can define indexes on fields that are frequently used for filtering in the WHERE clause. For more details, see Indexes for Different Storage Types.

3.1.5 Transaction Processing with Row Storage

To show how row storage can be efficient for transaction processing, suppose you insert a new row into the `BankTransaction` table.

SQL

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

An **INSERT** operation creates a new **$LIST** global without needing to load any existing **$LIST** globals into memory.

3.2 Columnar Storage Layout

You can define an entire table as having columnar storage using either the InterSystems SQL DDL language or a persistent class.

3.2.1 Define Columnar Storage Table Using DDL

To define a table with a columnar storage in InterSystems SQL DDL, use the `WITH STORAGETYPE = COLUMNAR` after the column definitions in CREATE TABLE.

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

When a table defined to use columnar storage is created via DDL, the system considers the lengths of VARCHAR columns and reverts them to row storage when they are longer than 12 characters (the current internal string length limit for columnar storage). All other columns, including any additional VARCHAR columns with lengths of 12 or less, are still stored in columnar storage, resulting in a table with a mixed storage layout. When all columns are fit for columnar storage, every field is stored in the columnar layout, offering the highest-performance ingestion, querying, and most efficient storage.

This command creates a `TransactionHistory` table containing historical data of account transactions performed. It contains the same columns as the `BankTransaction` table created in the Row Storage Layout section. Note that in this example, the `Description` column is stored using row storage, while the other columns are stored using columnar storage.

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

3.2.2 Define Columnar Storage Table Using a Persistent Class

To create a columnar storage table by using a persistent class, set the `STORAGEDEFAULT` parameter to the value `"columnar"`.

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

Unlike defining a table to use columnar storage through DDL, there is no check to validate that strings stored in this class meet the internal string length limit for columnar storage. As a result, you should exercise extra care with string lengths stored in tables defined to use columnar storage set with the STORAGEDEFAULT parameter, as you may encounter errors when ingesting and storing large numbers of unique strings that are longer than limit (which is 12 characters). Alternatively, you can set the MAXLEN of string-typed fields in your class to be 12 or less

The following persistent class defines a columnar storage table similar to the DDL-defined table created in the previous section. As with the BankTransaction table created in Define Row Storage Table Using a Persistent Class, this table defines a USEEXTENTSET parameter and bitmap extent index. For details on these settings, see Defining a Table by Creating a Persistent Class.

```
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 ];
}
```

You can declare any table as columnar. However, tables that use columnar as the default storage layout must specify either the Final class keyword or the NoExtent class keyword, with any immediate subclasses defined explicitly as Final.

As described earlier, the STORAGEDEFAULT parameter specifies which storage type InterSystems IRIS uses when generating the storage definition for a new table or column. However, some column types cannot be properly encoded into the optimized vector data types used for columnar storage. Serials, arrays, and lists are some types that cannot be properly encoded; a compilation error will arise when attempting to use a these data types with columnar storage. InterSystems IRIS automatically reverts to row storage in these cases:

- A column type is incompatible with columnar storage. Streams, arrays, and lists are examples of incompatible types.
- A column type is generally a poor fit for columnar storage. Strings longer than 12 characters are an example of a poor fit. In these cases, you can override the storage type by setting the STORAGETYPE = COLUMNAR clause on a column. For details, see the Mixed Storage Layout section. An example of when you might want to use columnar storage for strings longer than 12 characters anyway is when the cardinality of the strings is low. This means that there are only a few possible string values. If all the strings are different however, row storage is the better choice.

3.2.3 Columnar Storage Details

3.2.3.1 Global Structure

In a table with columnar storage, each column of a dataset is stored in a separate global. Within each column global, all row value elements are of the same data type and “chunked” into separate subscripts per 64,000 rows, similar to how $BIT values are stored. For example, if a table has 100,000 rows, then each column global has two subscripts. The first subscript contains the first 64,000 row values. The second subscript contains the remaining 36,000 row values. InterSystems IRIS uses a specialized vector encoding to efficiently store data of the same data type.

Suppose the TransactionHistory table defined in the previous section contains these records.

SQL

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

AccountNumber	TransactionDate	Description	Amount	Type
10001234	02/22/2022	Deposit to Savings	40.00	Deposit
10001234	03/14/2022	Payment to Vendor	-20.00	Withdrawal
10002345	07/30/2022	Transfer to Checking	-25.00	Transfer
10002345	08/13/2022	Deposit to Savings	30.00	Deposit

You can optionally examine the global storage structure from the Management Portal by clicking **System Explorer** and then **Globals**. In the namespace containing the table, you can then select **Show SQL Table Name** and find the globals corresponding to your table. For more details, see Managing Globals.

The Data/Master global contains a subscript for each row and is used to reference data involving row operations. Each subscript row is empty, because the data is stored by column in separate globals. This code shows a sample Data/Master global.

```
^THist = 4
```

`^THist` is the name of the global. The name shown here is for illustrative purposes. In tables created using DDL, or in a persistent class with the USEEXTENTSET=1 parameter specified, InterSystems IRIS generates more efficient, hashed globals with names such as `^EW3K.B3vA.1`. If a persistent class table does not specify USEEXTENTSET=1, then the global has a name of the format `^`*`TableName`*`D`. In the projected persistent class for an SQL table, the global is stored in the `<DataLocation>` element on the `Storage` class member. For example:

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

The table includes five additional globals, one per column, with names of the form, `^THist.V1`, `^THist.V2`, and so on. Each global stores a column of row values in a vector encoding, an internal data type designed to work with values of the same type and efficiently encode sparse data. The actual encoding is internal, but the **Globals** page and informational commands such as **ZWRITE** present a more readable format that describes:

- the type of the data
- the number of non-NULL elements in the column
- the length of the vector

Because this table has fewer than 64,000 rows, each column global contains only a single subscript. The data in the globals shown here have been truncated for readability.

```
^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",...]}
```

In this column global for a table with 200,000 rows, the data is spread across four global subscripts containing 64,000 + 64,000 + 64,000 + 8,000 elements. The count of elements is lower than the length, because the column includes NULL values.

```
^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 String Collation with Columnar Storage

When a string-typed field is defined in a table that uses columnar storage by default, its collation will be defined as EXACT, unless otherwise specified. EXACT collation is used for string-typed fields even if the MAXLEN of the field exceeds the 12 characters, causing the field to revert to row storage.

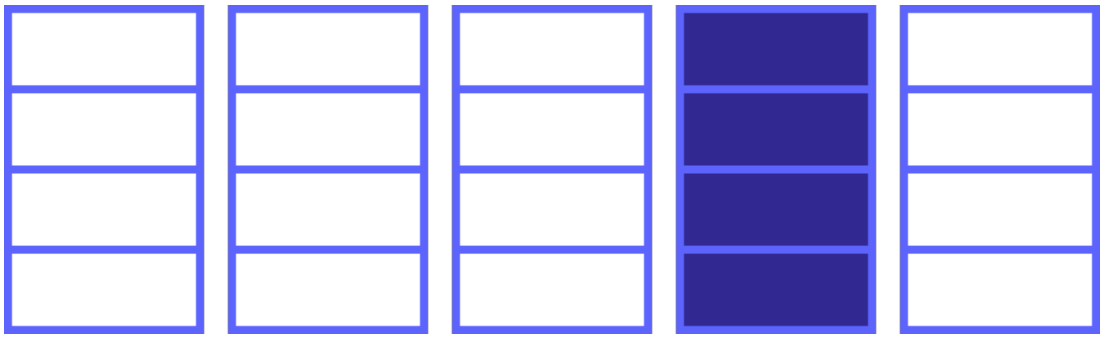
3.2.4 Analytical Query Processing with Columnar Storage

To show how columnar storage can be efficient for analytical processing, suppose you query the average size of all transaction amounts in the `TransactionHistory` table:

SQL

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

With columnar storage, this query loads only the `Amount` column global into memory and computes the average using the data in that column. None of the data from the other columns are loaded into memory, resulting in a more efficient query than if the data was stored in rows. Also, the optimized vector encoding comes with a set of dedicated vectorized operations that execute efficiently on an entire vector at a time, rather than on individual values. For example, calculating the sum of all elements inside a vector is several orders of magnitude faster than adding them up one by one, especially if each value needs to be extracted from a $list holding row data. Many of these vectorized operations leverage low-level SIMD (Single Instruction, Multiple Data) chipset optimizations.



You can check whether a query takes advantage of columnar storage efficiencies by analyzing the query execution plan. In the Management Portal, select **System Explorer** then **SQL** then **Show Plan**. Alternatively, use the **EXPLAIN** *query* SQL command. If the query plan includes statements such as "read columnar index", "apply vector operations" or "columnar data/index map", then the query is accessing data from column globals.

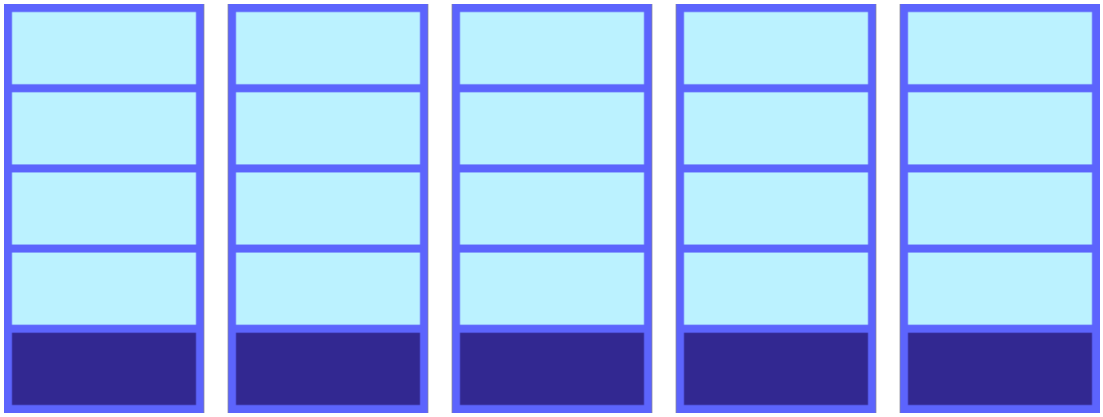
3.2.5 Transaction Processing with Columnar Storage

To show how columnar storage can be less efficient for transaction processing, suppose you insert a new row into the `TransactionHistory` table.

SQL

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

Because row data is distributed across all column globals, an **INSERT** operation must load the last chunk for each of these globals into memory to perform the insert.



Because inserts into columnar storage layouts can be so memory inefficient, perform them infrequently or in bulk, such as by using the LOAD DATA command. InterSystems IRIS includes optimizations that buffer INSERTs for columnar tables in memory before writing chunks to disk.

3.3 Mixed Storage Layout

For additional flexibility, you can define a table as having a mixture of row and columnar storage. In these tables, you specify an overall storage type for the table and then set specific fields as having a different storage type.

Mixed storage can be useful in transaction-based tables that have a few columns that you want to perform analytical queries on, such as fields that are often aggregated. You can store the bulk of the table data in rows, but then store the columns you frequently aggregate in the columnar format. Columns that are usually returned as is in row-level query results, without any filtering or grouping, might also be a good fit for row storage to save on the cost of materializing those rows before including them in the result. This enables you to perform transactional and analytical queries on a single table.

3.3.1 Define Mixed Storage Using DDL

To define a table with mixed storage in InterSystems SQL DDL, specify the `WITH STORAGETYPE = ROW` or `WITH STORAGETYPE = COLUMNAR` clause on individual columns in a CREATE TABLE command.

This syntax creates a table with the default, row-based storage layout but with the third column using columnar storage.

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

This syntax creates a table with a column-based storage layout but with the third column stored in row layout.

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

This **CREATE TABLE** command creates a `BankTransaction` table that stores all data in row layout except for the data in the `Amount` column, which uses columnar storage.

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

3.3.2 Define Mixed Storage Table Using a Persistent Class

To create a table with mixed storage by using a persistent class, specify the STORAGEDEFAULT parameter on the individual properties. Valid values are "columnar" and "row" (default).

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

This persistent class shows the definition of a columnar storage table. This table is similar to the DDL-defined table created in the previous section.

```
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 Mixed Storage Details

3.3.3.1 Global Storage

A table with mixed storage uses a combination of global storage structures, where:

- Data with a row storage layout is stored in $list format in the Data/Master global.
- Data with a columnar storage layout is stored in a vector encoding in separate column globals.

Consider this BankTransaction table:

```
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 ];
}
```

Notice that it is mostly similar to the example in Define Mixed Storage Table Using a Persistent Class, but uses columnar storage for the Amount column and row storage for everything else. The sample logical abstraction of the table data, shown here, is identical to the tables shown in Row Storage Details and Columnar Storage Details.

SQL

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

AccountNumber	TransactionDate	Description	Amount	Type
10001234	02/22/2022	Deposit to Savings	40.00	Deposit
10001234	03/14/2022	Payment to Vendor	-20.00	Withdrawal
10002345	07/30/2022	Transfer to Checking	-25.00	Transfer
10002345	08/13/2022	Deposit to Savings	30.00	Deposit

In the Management Portal, the **Globals** page shows how this data is stored. The Data/Master global stores the data for the rows. This format is similar to the format shown in Row Storage Details, but the data for the `Amount` column is not present.

```
^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")
```

The table includes an additional global that stores the `Amount` column data. This format is similar to the format shown in Columnar Storage Details.

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

The table metadata contains information about column order. InterSystems IRIS uses this information to construct the relational table using the data stored in the row and column globals.

3.3.3.2 String Collation

When mixing storage layouts, any string-typed field that uses columnar storage, either by using the table's default storage or as an explicit setting on a particular field, is defined to have EXACT collation.

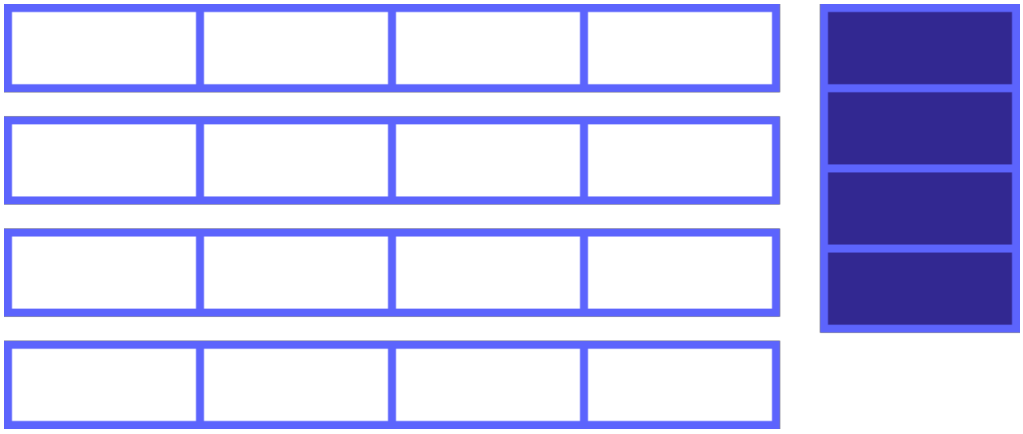
3.3.4 Analytical Query Processing with Mixed Storage

The efficiency of analytical queries depends on the data you access. Consider the `BankTransaction` table created in the previous section, where only the `Amount` column uses columnar storage. Querying the average size of all transaction amounts is efficient, because only the `Amount` column is accessed.

SQL

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

In this diagram, the `Amount` column is separated from the columns that are stored in rows.



However, if a query performed additional aggregations on other columns stored as rows, the performance gains might not be as noticeable.

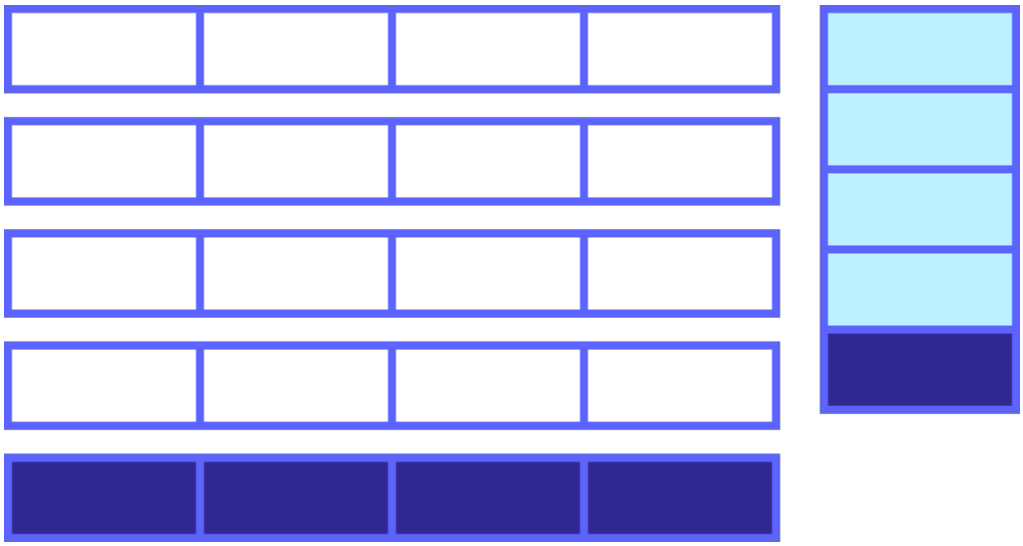
3.3.5 Transaction Processing with Mixed Storage

If a mixed storage table requires frequent updates and insertions, performance can be slower that pure row storage, but not as slow as pure columnar storage. For example, suppose you insert a new row into the `BankTransaction` table that uses columnar storage only for the `Amount` column.

SQL

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

The **INSERT** operation does not load any existing row globals, but to insert the new `Amount` value, the entire last chunk of the `Amount` column global must be loaded into memory. Depending on your data, this overhead on transaction processing might be preferable to maintaining separate tables for transactions and analytics.



3.4 Indexes on Storage Layouts

The type of storage format that you choose does not preclude you from defining indexes on your tables. The benefits gained from indexes can vary depending on the storage format.

3.4.1 Indexes on Row Storage Layouts

As shown in the Analytical Processing with Row Storage section, filter and aggregate operations on columns in tables with a row storage layout can be slow. Defining a bitmap or columnar index on such tables can help improve the performance of these analytical operations.

A *bitmap index* uses a series of bitstrings to represent the set of ID values that correspond to a given indexed data value. This format is highly compressed and can reduce the number of rows that you look up. Also, different bitmap indexes can be combined using Boolean logic for efficient filtering involving multiple fields or field values. For more details on working with bitmaps, see Bitmap Indexes.

Using the `BankTransaction` table from earlier sections, suppose you create this bitmap index on the `Type` column:

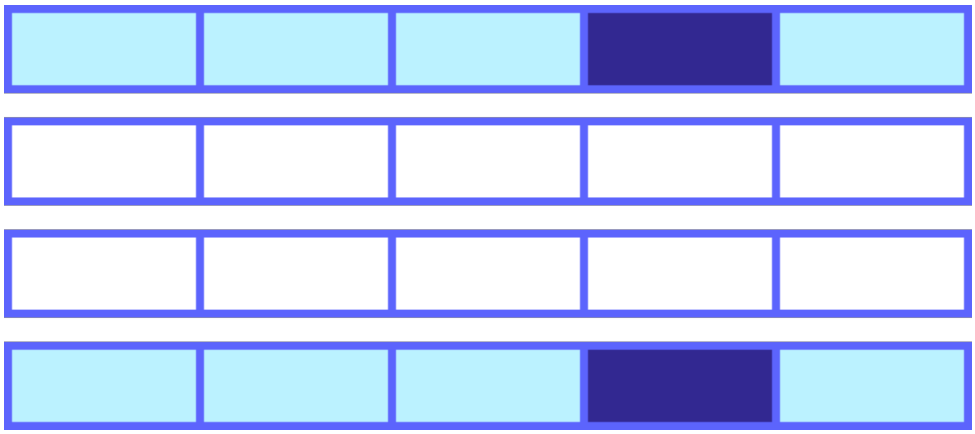
```
CREATE TABLE Sample.BankTransaction (
  AccountNumber INTEGER,
  TransactionDate DATE,
  Description VARCHAR(100),
  Amount NUMERIC(10,2),
  Type VARCHAR(10))

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

Suppose you then perform an aggregate query in which you limit rows based on one of the transaction types.

SQL

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

The bitmap index ensures that the query iterates only the rows for the selected transaction type, as shown by this diagram.



However, as shown by this diagram, after using the bitmap index to find eligible rows, InterSystems IRIS still needs to fetch the entire row even if you need only a single element per row. If your table has millions of rows, even a filtered set of rows can incur a heavy performance cost.

Alternatively, you can define a *columnar index* on a column that is frequently queried. A columnar index stores the same vectorized column data described in Columnar Storage Details. Use this index to improve analytical query performance on row storage tables at the expense of the storage costs of an additional index.

To define a columnar index using InterSystems SQL DDL, use the `CREATE COLUMNAR INDEX` syntax of CREATE INDEX:

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

To define a columnar index in a persistent class, specify the `type = columnar` keyword on the index you define:

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

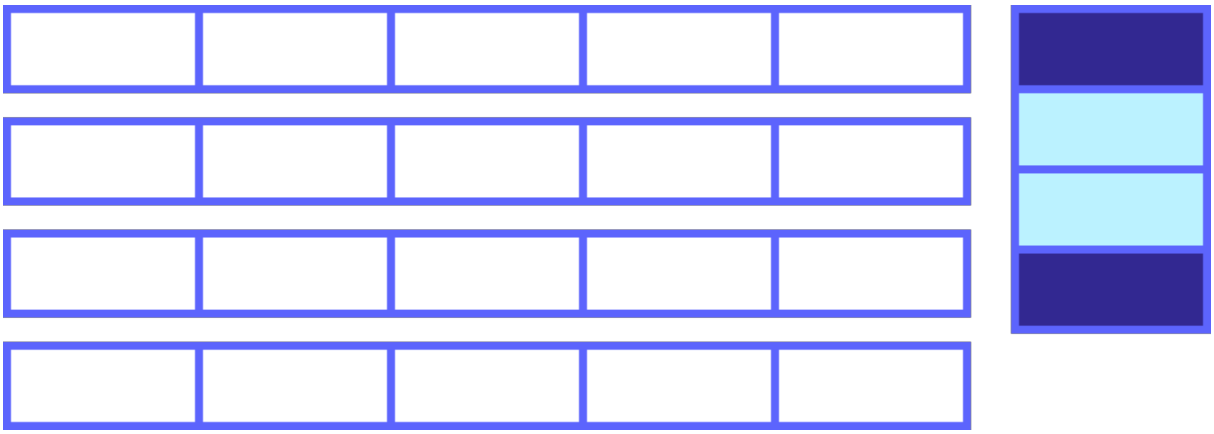
Using the `BankTransaction` table again, suppose you create a bitmap index on the `Type` column and a columnar index on the `Amount` column:

```
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)
```

The aggregate query from earlier now combines the use of both indexes to access only the data being queried. First the query looks up which rows to access based on the `TypeIndex` bitmap index. Then it accesses the `Amount` values for those rows from the `AmountIndex` columnar index.

SQL

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



3.4.2 Indexes on Columnar Storage Layouts

If the primary purpose of the columnar storage table is aggregation and filter operations, then additional indexes might not provide many performance gains. For example, although a bitmap index can improve filtering performance, the performance gains might not justify the additional storage they take up and the ingestion overhead they cause. If your query workload involves lookups on highly selective fields or unique keys, then defining regular indexes might still be worthwhile. Determining whether such indexes are worth defining requires query experimentation and analyzing the trade-offs. For more details on defining indexes, see Defining and Building Indexes.

3.5 Suggested Application of Row and Columnar Storage

While InterSystems has no prescriptive formula for whether to use columnar or row-wise storage in your tables, there are general guidelines that may help you when defining the your InterSystems SQL schema structure. In general, adhere to the following guidelines:

- If your InterSystems IRIS SQL tables contain less than one million rows, there is no need to consider columnar storage. The benefits of vectorized storage are unlikely to make a significant difference on smaller tables.
- Use the default row-wise storage layout for applications that leverage InterSystems SQL or Objects, such as a transaction processing application. Most queries issued for applications or programmatic transactions only retrieve or update a limited number of rows and rarely use aggregate functions. In such cases, the benefits offered by columnar storage and vectorized query processing do not apply.
- If such an application employs operational analytics, add columnar indexes if the performance of analytical queries is not satisfactory. In these cases, look for numeric fields used in aggregations, like quantities or currencies, or fields with high cardinality used in range conditions, like timestamps. Columnar indexes can be used in conjunction with bitmap indexes to avoid excessive read operations from the master map or regular index maps.
- Use the columnar storage layout if you are deploying an InterSystems IRIS SQL schema for analytical use cases. Star schemas, snowflake schemas, or other de-normalized table structures, as well as broad use of bitmap indexes and batch ingestion, are good indicators of these use cases. Analytical queries benefit the most from columnar storage when they aggregate values across rows. When defining a columnar table, InterSystems IRIS automatically reverts to a row-wise storage layout for columns that are not a good fit for columnar storage, including streams, lengthy strings, or serial fields. InterSystems IRIS SQL fully supports mixed table layouts and uses vectorized query processing for eligible parts of the query plan. On columnar tables, you may omit a bitmap index, as their value on such tables is limited.

These suggestions may be impacted by both data-related factors and the environment in which your application runs. Therefore, InterSystems recommends that customers tests different layouts in a representative setup to determine which layouts will provide the best performance.

4

Define SQL Optimized Tables Through Persistent Classes

In InterSystems IRIS SQL, you can define persistent classes in Object Script that will present themselves as SQL tables, instead of writing Data Definition Language (DDL) statements. This page describes a number of class features you can use within a class definition to ensure high performance for SQL statements accessing the table. These features can be applied to a new or existing class definition, but some require additional attention if data already exists for this table.

4.1 Global Naming Strategy

The name of the global that stores data is determined by the value of the USEEXTENTSET and DEFAULTGLOBAL class parameters that define the table. The global names also determine how indexes are named. The relationship between the two parameters is summarized below:

- If USEEXTENTSET=0, the global name will consist of a user-specified name and an appended letter code. For example, a class named `Sample.MyTest` will correspond with a global named `^Sample.MyTestD` for the master map and `^Sample.MyTestI` for all index maps. If DEFAULTGLOBAL is specified, the specified global name is substituted for the persistent class name.
- If USEEXTENTSET=1, hashed global names will be created for the master map and each of the separate index maps. This involves hashing both the package name and the class name and appending an incrementing integer for each master and index map. These names are less user readable, but lead to better low-level efficiencies for storing and traversing the globals. Using separate globals for each index map also offers performance and operational benefits. If DEFAULTGLOBAL is specified, that name is substituted for the hashed package and class names.

Both the USEEXTENTSET and DEFAULTGLOBAL class parameters drive how a storage definition is generated. Changing them for a class that already has a storage definition will have no impact until you reset the storage definition, as described in Resetting the Storage Definition. Note that this will render any existing data inaccessible; therefore, these parameters should only be updated in a development environment, prior to loading any import data.

InterSystems recommends setting USEEXTENTSET to 1, which is the default when creating tables with CREATE TABLE. For reasons of backwards compatibility, the default for the USEEXTENTSET parameter in class inheriting from %Persistent is still 0. Therefore, InterSystems recommends setting this parameter to 1 for new classes, before compiling for the first time.

For more information about the USEEXTENTSET and DEFAULTGLOBAL parameters, refer to Hashed Global Names and User-Defined Global Names respectively.

4.2 Decide Storage Layout

You can define persistent classes to take advantage of columnar storage on either all properties of the class or a subset of the properties of the class. The benefits and drawbacks of these approaches are further explained in Choose an SQL Table Storage Layout.

By default, persistent classes use the row storage layout for all properties in a class. However, you can use the STORAGEDEFAULT parameter to set this default to columnar. In addition, you can define a mixed storage layout that uses a row storage layout for some properties and a columnar storage layout for others.

Note: InterSystems IRIS uses the value of the STORAGEDEFAULT parameter as the default when generating storage definition entries for the class. This value is only considered upon the initial compilation of the class or when adding new properties. Changing this parameter for a class that already has a storage definition (saved in a Storage XData block) has no effect on that storage, including on data already stored for the class extent. As a result, you should decide your storage layout before compiling your class for the first time.

4.3 Indexes

You can define an index for a table field or group of fields. You can define several different type of indexes: standard, bitmap, bitslice, and columnar. SQL optimization uses the defined indexes to access specific records for a query, update, or delete operation based on predicates that involve the fields covered by those indexes.

One of the key differences between defining indexes in a class definition and in a DDL statement is that indexes defined in a class must be built separately from being created, as class compilation only affects its definition and never its data (including index entries). You must manually build the defined indexes to make use of them in future SQL queries.

See Defining Indexes Using a Class Definition for examples of how to define an index in a class definition.

For more information about what fields to index, refer to the What to Index.

4.4 The Extent Index

An Extent Index is a special type of index that does not index any particular fields, but only contains the ID entries for each row in the master map. When a class has a bitmap-compatible IDKEY, the extent index can be implemented as a bitmap extent index, which offers extremely efficient existent checking and counting. For example, the query `SELECT COUNT(*) FROM t` is an order of magnitude more efficient on a table with a bitmap extent index than it is on a table without such an index.

To define a bitmap extent index in a class, write the following:

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

The type keyword can be left out in the rare case that your class does not have a bitmap-compatible IDKEY.

A bitmap extent index will automatically be created when creating a table using the `CREATE TABLE` DDL statement. InterSystems recommends adding a bitmap extent index to any persistent class. As with standard indexes, the extent index must be built separately from being created.