Article
Allyson Gerace · Feb 6, 2019 13m read

Know Your Indices

This is the first in a pair of articles on SQL indices.

Part 1 - Know your indices

What is an index, anyway?

Picture the last time you went to a library. Typically they have books sorted by subject matter (and then author and title), and each shelf has an end-plate with a code describing the subject of its books. If you wanted to collect books of a certain subject, instead of walking across every aisle and reading the inside cover of every book, you could head straight for the bookshelf labelled with your desired subject matter and choose your books.

A SQL index has the same general function: improving performance by giving a quick reference to the value of fields for each row in a table.

Setting up indices is one of the main steps in preparing your classes for optimal SQL performance.

In this article, we ' ll cover:

1. What is an index and why/when should I use them?
2. What types of indices exist and which scenarios are they ideal for?
3. What does an index look like?
4. How do I create one?
5. And when I have indices, what do I do with them?

I will be referring to classes from our Sample schema. These are available through the following Github repository, and they are also provided in the Samples namespace in Caché and Ensemble installations:

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

The basics

You can index any persistent property and any property that can be reliably computed from persistent data.

Say we want to index the property TaxID in Sample.Company. In Studio or Atelier, we would add the following to the class definition:

```
Index TaxIDIdx On TaxID;
```

The equivalent DDL SQL statement would look something like this:

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

The default global index structure is as follows:

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

Note that there are fewer subscripts to read than fields in a typical data global.

Consider the query "SELECT Name,TaxID FROM Sample.Company WHERE TaxID = 'J7349' " . It ' s logically straightforward, and the query plan for executing this query reflects this:

Query Plan

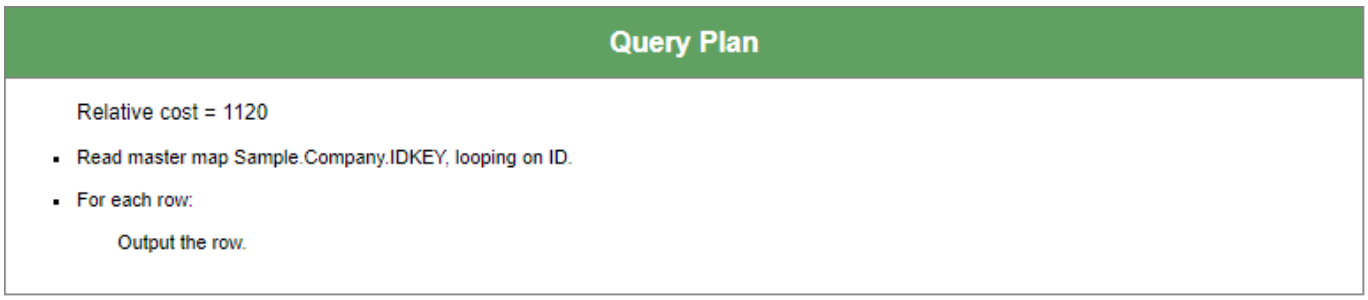
Relative cost = 495.2

- Read index map Sample.Company.TaxIDIdx, using the given %SQLUPPER(TaxID), and looping on ID.
- For each row:

  Read master map Sample.Company.IDKEY, using the given idkey value.
  Output the row.

This plan essentially says we check the index global for rows with the given TaxID value, then refer back to the data global (" master map ") to retrieve the matching row.

Now consider the same query *without* an index on TaxIDX. The resulting query plan is, as expected, less efficient:



Without indices, IRIS's underlying query execution relies on reading into memory and applying the WHERE clause ' s condition to each row of the table – and since we would not logically expect any company to share a TaxID, we ' re doing all this work for just one row!

Of course, having indices means having index *and* row data on disk. Depending on what we have a condition on and how much data our table contains, this can prove to have its own challenges when we create and populate an index.

So when do we add an index to a property?

The blanket case is when we frequently condition on a property. Some examples are identifying information such as a person ' s SSN or a banking account number. You can also consider birth dates or an account ' s funds. Going back to Sample.Company, perhaps the class would benefit from indexing property Revenue if we wanted to collect data on high-earning organizations. Conversely, properties we ' re unlikely to condition on are less fitting to be indexed: say a company slogan or description.

Simple – except we must also consider which type of index is best!

Types of Indices

There are six major types of indices I ' ll go over here: standard, bitmap, compound, collection, bitslice, and data. I ' ll also touch briefly on iFind indices, which are based on streams. There are possible overlaps here, and we ' ve already touched on standard indices with the example above.

I will share examples on how to create indices in your class definition, but adding new indices to a class is more involved than simply adding a line in your class definition. We ' ll go over additional considerations in the next part.

Let ' s use Sample.Person as an example. Note that Person has subclass Employee, which will be relevant in understanding some examples. Employee shares its data global storage with Person, and all of Person ' s indices are inherited by Employee – this means Employee uses Person ' s index global for these inherited indices.

If you ' re unfamiliar, here ' s a general overview of these classes: Person has properties SSN, DOB, Name, Home (an embedded Address object containing State and City), Office (also an Address), and list collection FavoriteColors. Employee has additional property Salary (which I myself defined).

Standard

```
Index DateIDX On DOB;
```

Here I ' m using ' standard ' loosely to refer to indices that store the plain value of a property (as opposed to a binary representation). If the value is a string, it will be stored under some collation – SQLUPPER by default.

Compared to bitmap or bitslice indices, standard indices are more human-readable and relatively effortless to maintain. We have one global node for each row in the table.

Below is how DateIDX is stored at a global level.

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

Note the first subscript after the index ' s name is the date value, the last subscript is the ID of the Person with that DOB, and the value stored at this global node indicates that this Person is also a member of the subclass Sample.Employee. If this Person were not a member of any subclass, the value at the node would be an empty string.

This base structure will be consistent with most non-bit indices, where indices on more than one property create more subscripts in the global, and having more than one value stored at the node produces a $listbuild object, for example:

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

Bitmap – A bitwise representation of the set of IDs corresponding to a property value.

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

Bitmap indices are stored per unique value as opposed to standard indices, which are stored per row.

Going further into the example above, let ' s say the Person with ID 1 lives in Massachusetts, ID 2 in New York, ID 3 in Massachusetts, and ID 4 in Rhode Island. HomeStateIDX is essentially stored as follows:

(…)	0	0	0	0	-

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

If we wanted a query to return data from people living in New England, the system performs a bitwise OR on the bitmap index ' s relevant rows. It ' s quick to see that we must load into memory Person objects with ID 1, 3, and 4 at the very least.

Bitmaps can be efficient for AND, RANGE, and OR operators in your WHERE clauses.

While there is no official cap of how many unique values you can have for a property before a bitmap index will be less efficient than a standard index, the general rule of thumb is up to about 10,000 distinct values. So, while a bitmap index may be effective on a US state, a US city or county bitmap index would not be as useful.

Another concept to consider is storage efficiency. If you ' re planning on adding and removing rows from your table frequently, your bitmap index ' s storage can become less efficient. Consider the example above: say we removed many rows for whatever reason, and we no longer have people in our table who live in less populated states such as Wyoming or North Dakota. The bitmap thus has several rows with only zeroes. On the other side of the coin, creating new rows on large tables can eventually become slower as large bitmap storage must accommodate more unique values.

In these examples, I have about 150,000 rows in Sample.Person. Each global node stores up to 64,000 IDs, so the bitmap index global at value MA is split into three parts:

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

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

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

Special case: Extent Bitmap

An extent bitmap, often named $<ClassName>, is a bitmap index on the IDs of a class – this gives IRIS a quick way of knowing whether a row exists and can be helpful for COUNT queries or queries on subclasses. These indices are generated automatically when a bitmap index is added to the class; you can also manually create a bitmap extent index in a class definition as follows:

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

Or via DDL keyword BITMAPEXTENT:

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

Compound – Indices based on two or more properties

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

The general use case of compound indices is having frequent queries conditioning on two or more properties.

The order of properties in a compound index matters due to how the index is stored on a global level. Having the more selective property first is more performance efficient because it will save initial disk reads of the index global; in this example, Office.City is first because there more unique cities than states in the US.

Having a less selective property first is more space efficient. In terms of global structure, the index tree would be more balanced were State to be first. Think about it: each state contains plenty of cities, but some city names belong to only one state.

You can also consider whether you ' d expect to run frequent queries conditioning on only one of either property – this can save you from defining yet another index.

Here ' s an example of the compound index ' s global structure:

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

Aside: Compound index or bitmap indices?

For queries with conditions on multiple properties, you may also want to consider whether separate bitmap indices would be more effective than a single compound index.

Bitwise operations on two different indices may be more efficient provided that bitmap indices suit each property appropriately.

You can also have compound bitmap indices – these are bitmap indices where the unique value is the intersection of the multiple properties you ' re indexing on. Consider the table given in the previous section, but instead of states we have every possible pair of a state and city (e.g. Boston, MA, Cambridge, MA, even Los Angeles, MA, etc.), and cells get 1 ' s for rows that adhere to both values.

Collection – Indices based on collection properties

Here we have property FavoriteColors defined as follows:

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

With each of the following indices defined for demonstration purposes:

Index fcIDX1 On FavoriteColors(ELEMENTS);

Index fcIDX2 On FavoriteColors(KEYS);

Here I ' m using ' collection ' to refer more broadly to single-cell properties containing more than one value. List Of and Array Of properties are relevant here, and if you want, even delimited strings.

Collection properties are automatically parsed to build their indices. For delimited properties, say a phone number, you ' d need to define this method, <PropertyName>BuildValueArray(value, .valueArray), explicitly.

Given the above example for FavoriteColors, fcIDX1 would look something like this for a Person with favorite colors blue and white:

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

(…)

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

fcIDX2 would look like:

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

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

In this case, since FavoriteColors is a List collection, an index based on its keys is less useful than an index based on its elements.

Please refer to our documentation for more in-depth considerations of creating and managing indices on collection properties:

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

Bitslice – Bitmap representation of the bit string representation of numeric data

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

Unlike bitmap indices, which contain flags representing which rows contain a specific value, bitslice

indices first convert numeric values from decimal to binary, then create a bitmap on each digit of the binary value.

Let ' s take the example above and, for realism ' s sake, simplify Salary as being in units of $1000 – so If an employee ' s salary is stored as 65, it is understood to represent $65,000.

Say we have Employee with ID 1 who has Salary 15, ID 2 Salary 40, ID 3 Salary 64, and ID 4 Salary 130. The corresponding bit values are:

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

Our bit string spans over 8 digits. The corresponding bitmap representation – the bitslice index values – is essentially stored as follows:

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

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

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

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

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

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

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

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

Note that operations modifying Sample.Employee or its rows ' salaries, i.e. INSERTs, UPDATES, and DELETEs, now require each of these global nodes, or bitslices, to be updated. Adding a bitslice index to multiple properties in a table or a frequently modified property can have performance risks. In general, maintaining a bitslice index is more costly than maintaining standard or bitmap indices.

Bitslice indices are highly specialized and thus have specific use cases: queries that need to perform aggregate calculations, e.g. SUM, COUNT, or AVG.

In addition, they can only be effectively used on numeric values – character strings are converted to a binary 0.

Note that if the data table, as opposed to indices, must be read to check a query ' s condition, bitslice indices will not be chosen to execute the query. Let ' s say Sample.Person does not have an index on

Name. If we were to calculate the average salary of employees with the last name Smith – "SELECT AVG(Salary) FROM Sample.Employee WHERE Name %STARTSWITH 'Smith,' " – we would need to read data rows to apply the WHERE condition, and thus the bitslice index would not be used in practice.

There are similar storage concerns for bitslice and bitmap indices on tables where rows are frequently created or removed.

Data - Indices with data stored at their global nodes.

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

In several of the previous examples, you may have observed the string " Sample.Employee~" stored as the value at the node itself. Recall that Sample.Employee inherits indices from Sample.Person. When we query on Employees in particular, we read the value at index nodes matching our property condition to check that said Person is also an Employee.

We can also explicitly define what values to store. Having data defined at your index global nodes can save reads of the data global altogether; this can be useful for frequent selective or ordered queries.

Take the above index as an example. If we wanted to pull identifying information about a person given all or part of their name (e.g. to search for clients ' information in a front-desk application), we could have a query such as "SELECT SSN, Name, DOB FROM Sample.Person WHERE Name %STARTSWITH 'Smith,J' ORDER BY Name " . Since our query conditions on Name and the values we ' re retrieving are all contained within the QuickSearchIDX global nodes, we only need to read our I global to execute this query.

Note that data values cannot be stored with bitmap or bitslice indices.

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

iFind Indices

Ever heard of these? Neither have I. iFind indices are used on stream properties, but to use them you need to specify their names with keywords in the query.

I could explain more, but Kyle Baxter already has a helpful article on this:

https://community.intersystems.com/post/free-text-search-way-search-your-text-fields-sql-developers-are-hiding-you

See Part 2, on managing defined indices, here.

[EDIT 04/16/2020: Adjustments for readability.]

#Best Practices #Indexing #Performance #SQL #Caché #InterSystems IRIS