Article
Benjamin De Boe · Sep 9, 2016 4m read

# Using complex filters

In a conference call earlier this week, a customer described how they built an iKnow domain with clinical notes and now wanted to filter the contents of that domain based on the patient's diagnosis codes. With such filters, they wanted to explore the corellations between iKnow entities and certain diagnosis codes, first through the Knowledge Portal to get a good sense of the sort of entities and then through more analytical means with the aim of eventually building smart early warning systems.

In an iKnow domain, you can easily define metadata fields that, for every source in your domain, can have a value by which queries can be filtered using the %iKnow.Filters.SimpleMetadataFilter. For example, a clinical note may have an author, encounter date and patient ID, so that you can restrict your results to a particular clinician, date range and/or patient. Now, the regular metadata API was designed to support single-valued fields only, because building an infrastructure to cope with arbitrarily complex metadata structures *next to* the existing data that's typically in a table structure would just make things cumbersome. So, if you would want to filter your notes based on diagnosis codes (plural) that are associated with the patient level, the simple metadata infrastructure (what's in a name) couldn't do this for you.

That's where the %iKnow.Filters.SqlFilter comes in. It allows you to express your filter criteria in SQL, querying the data model your iKnow-indexed text stemmed from, in its entirety. For example, a query implementing that diagnosis filter idea could look like this:

```
SELECT ...
FROM MyEMR.Note n
WHERE n.PatientID IN (
    SELECT d.PatientID
    FROM MyEMR.Diagnosis d
    WHERE d.DiagnosisCode IN ('824.8', '824.9')
)
```

Now, what goes into those three dots we need to return to our SqlFilter? Obviously, it needs to somehow reference our iKnow sources, which correspond to records in the MyEMR.Note table. The documentation for the SqlFilter class describes three options:

1. In some cases, you might just JOIN your data models tables to iKnow-fed ones in the %iKnow.Objects package or those created by %iKnow.Tables.Utils. From these, you can obtain the **SourceID** column, giving you the unique, iKnow-generated identifier for the record that the filter is looking for. Note that this identifier is generated by iKnow upon building your domain and may change after rebuilding it.
2. More typically, you'll be using the **ExternalID** mechanism. This is a string iKnow created while loading to give you an externally referencable identifier, which remains the same upon rebuild operations. It consists of a prefix identifying where the data came from (":SQL:" when loading from tables), a group name and an ID field, both of which you configure as part of your domain definition. The combination of these three elements should be unique within your domain, so you could for example have used the group name to refer to the table name and let the ID field refer to some unique row identifier of your data model. Rebuilding this ExternalID is the second way to inform the SqlFilter of which records to retain.
3. As sort of a shorthand for the ExternalID mechanism, and assuming the data was loaded through a SQL query anyhow, you can simply let your filter query retrieve an **IdField** and **GroupField** column, which will then be concatenated automatically into the ExternalID.

So, assuming the data location our data was loaded from looks like this:

```
...
<data>
<table tableName="MyERP.Note" idField="NoteID" groupField="NoteType"
dataFields="NoteText" />
</data>
...
```

Our query for the SqlFilter could look like this:

```
SELECT NoteType AS GroupField, NoteID AS IdField
FROM MyEMR.Note n
WHERE n.PatientID IN (
SELECT d.PatientID
  FROM MyEMR.Diagnosis d
  WHERE d.DiagnosisCode IN ('824.8', '824.9')
)
```

## And more ahead

It's clear the SqlFilter can help you cover highly complex filter criteria, as long as things can be expressed by SQL. Of course, you can shield this complexity from your application users by hiding predefined SQL queries behind a button in your GUI, optionally parameterizing things as you see fit. In a future version of the Aviation demo included in the SAMPLES namespace, we'll elaborate such a scenario with a simple table tracking "common" queries:

```
Class Aviation.UI.SavedFilter Extends %Persistent
{
Property Name As %String(MAXLEN = 100);
Property SQL As %String(MAXLEN = "");
Index NameIdx On Name [ Unique ];
}
```

This table will then be populated with a couple of queries, aligning with the group name and ID field settings of the Aviation.ReportDomain. For example, the following filter will select all events involving pilots aged 60 or more:

```
SELECT e.EventId AS IdField, YEAR(e.EventDate) AS GroupField
FROM Aviation.Event e
WHERE EXISTS (
    SELECT *
    FROM Aviation.Crew c
    WHERE c.EventID = e.EventID AND Age > 60 AND c.Category IN ('Pilot', 'Flight Inst
ructor')
)
```

A future release of the Knowledge Portal will also include hooks so you can extend the list of filters shown there to include these sorts of custom creations, but that's for a future article.

#Best Practices #iKnow

**Source URL:** https://community.intersystems.com/post/using-complex-filters