

How can I create new data-sets from existing tables for use with MapInfo

Being able to access data in a usable form is fast becoming a key-stone productive computing - particularly given the growing amounts of data being collated by organisations the world over.

It would be useful to be able to extract **only** the data we need for a particular job **without** having to wade through superfluous fields of data. This becomes even more of a consideration when this data may be sitting across several different tables.

Add to this the GIS mapping requirement and there's a major potential headache developing...

This is where iShare can help!

In this example, we are going to create a MapInfo TAB file based on data extracted from two tables (containing a fair number of fields each) in order to produce a mapable 'Council Tax Reference' file associated with addresses from our main Address Lookup Table.

The process will be split into three parts. First we extract the data we need from our source tables into a third, which we will then use to generate the spatial geometry and finally export into a TAB file that can be used in iShare - or 'exported' for use elsewhere.

In this way we can provide users/clients with **only** the data they need - in a form that is of practicable use to them.

The first two of these steps we perform within Postgres, the last we do in iShareMaps Studio:

The Postgres Bits

1. Data Extraction

The two tables we will be using to source our data are shown below and have the following formats:

AstunLocationLookup														
Fields	UniqueId	Parent	Name	X	Y	Postcode	Suffix	Type	idxfti	pcodes	postcode_nospace	wkb_geometry	idxftimp	idxmpall
BLPU_APP_CROSS_REF														
Fields	UPRN	ENTRY_DATE	SOURCE	CROSS_REFERENCE	START_DATE	LAST_UPDATE_DATE	PRO_ORDER	CHANGE_TYPE	MAPINFO_ID	IE_NO	NE			

The actual "useful" data that we want is only two columns (Address & Council Tax Reference). In addition to these we'll also need X-Y co-ordinates (for mapping) and a unique ID for the property.

The extracted table will look like this:

CTAX_MI_VIEW						
Fields	Name	X	Y	UPRN	SOURCE	CROSS_REFERENCE

The SQL statement used to extract these fields from the source tables is as follows:

```
SELECT "AstunLocationLookup"."Name",
       "AstunLocationLookup"."X",
       "AstunLocationLookup"."Y",
       "AstunLocationLookup"."UniqueId",
       "BLPU_APP_CROSS_REF"."SOURCE",
       "BLPU_APP_CROSS_REF"."CROSS_REFERENCE"
INTO "CTAX_MI_VIEW"
FROM "AstunLocationLookup",
     "BLPU_APP_CROSS_REF"
WHERE "AstunLocationLookup"."UniqueId" = "BLPU_APP_CROSS_REF"."UPRN"
      AND "BLPU_APP_CROSS_REF"."SOURCE" = 'CTAX';
```

For those that don't know SQL, the above is broken down as follows:

The SELECT Statement

```
SELECT "AstunLocationLookup"."Name",
       "AstunLocationLookup"."X",
       "AstunLocationLookup"."Y",
       "AstunLocationLookup"."UniqueId",
       "BLPU_APP_CROSS_REF"."SOURCE",
       "BLPU_APP_CROSS_REF"."CROSS_REFERENCE"
```

The ***SELECT*** statement specifies the data we want to extract.

From the ***AstunLocationLookup*** table:

- Name** - the full Address for the property
- X** - X co-ordinate of the property
- Y** - Y co-ordinate of the property
- UniqueId** - unique identifier for the property (the UPRN)

From the **BLPU_APP_CROSS_REF** table:

- SOURCE** - used to 'filter' the table for Council Tax-specific records)
- CROSS_REFERENCE** - the Council Tax Reference

The INTO Statement

```
INTO "CTAX_MI_VIEW"
```

This specifies the name of the new table we wish to create from the extracted data.

The FROM Statement

```
FROM "AstunLocationLookup",
     "BLPU_APP_CROSS_REF"
```

This simply specifies the tables from which we are to extract the data. You will notice that the **SELECT** statement also specifies the source table for each field by using it as a prefix (eg "AstunLocationLookup"."X"). Strictly speaking this isn't always necessary; however it does prevent any 'confusion' and/or data-errors caused by having fields of the same name in both tables.

The WHERE Statement

```
WHERE "AstunLocationLookup"."UniqueId" = "BLPU_APP_CROSS_REF"."UPRN"
      AND "BLPU_APP_CROSS_REF"."SOURCE" = 'CTAX';
```

This is the 'filter' used to extract **only** those records where:

- The **UniqueId** in one table matches the ***UPRN*** from the other (ie the record in each table corresponds to the same property)
- The **SOURCE** field is ***CTAX*** (ie the record is Council Tax related)

2. The Spatial Field

For this we simply call the function **at_wkf_geo_createpointsfromxy** that is supplied as part of the DataShare installation. What this function does is generate the spatial geometry required by MapInfo for plotting points on a map from x-y co-ordinates.

Following the use of this function, the final layout of our table is as follows:

CTAX_MI_VIEW							
Fields	Name	X	Y	UPRN	SOURCE	CROSS_REFERENCE	wkb_geometry

Our Data Explained

If you look at the data in this table, you will spot that, for each **"UniqueId"** (UPRN) there are -or can be- multiple records for each UPRN.

WHY? This is supposed to be a **Unique ID**...

The answer is, actually, quite simple - and if one was to use the **BLPU** table, rather than **AstunLocationLookup** in the above procedure, you would find that this is, indeed, the case. The UPRN of a property refers to the building itself - ie "The Property". In the case of multiple records for a UPRN, you will notice that 'properties' that share a UPRN all exist within the same (physical) building, or complex. In this example for Council Tax, it could well be that we have a number of 'properties' within a single building; each with its own Council Tax Reference. For this reason it makes sense to use **AstunLocationLookup** as this allows us to differentiate between these discrete addresses rather than the physical buildings.

Creating a Function

Obviously we don't want to be lumbered with having to perform all these steps manually each time we have to generate this table - for example when the data changes. So we can create a function within postgres to store this procedure for later use.

The only additions we need to make to the code we have so far are:

Drop an existing table

```
SELECT at_sys_drop_table('CTAX_MI_VIEW');
```

This is another function supplied with DataShare that will allow a table to be dropped (or not - if it doesn't exist), prior to being created (or re-created) without causing issues with Postgres itself - which can occasionally happen when simply using the **DROP** statement on its own. It also handles things like table/index dependencies that could prevent a **DROP** statement from exiting cleanly. By using the Astun function, one is able to simply provide the name of the table and let the function deal with everything else required to drop the table.

Postgres Functions

In order to add our procedure to Postgres as a valid function, we need to top-n-tail our procedure with some Postgres pre-requisites.

Preceding our code we have:

```
-- Function: at_tcbc_create_ctax_mi_view()
-- DROP FUNCTION at_tcbc_create_ctax_mi_view();

CREATE OR REPLACE FUNCTION at_tcbc_create_ctax_mi_view()
  RETURNS text AS
  $BODY$
```

The first two lines are commented-out (via the "--") as the first is simply the name of our function (which doesn't get executed) and the second is there purely to save typing should we ever need to make amendments/test our function at any time.

The next two lines are used to define this as a function when this code is executed.

The last line (\$BODY\$) is used to define the start of the code which forms the function itself.

Following our code we have:

```
$BODY$
  LANGUAGE 'sql' VOLATILE;
```

```
ALTER FUNCTION at_tcbc_create_ctax_mi_view() OWNER TO postgres;  
COMMENT ON FUNCTION at_tcbc_create_ctax_mi_view() IS  
'Astun Technology custom function for Torfaen to build a Council Tax view in MapInfo format';
```

The first line (\$BODY\$) defines the end of our code and the lines after it define various aspects of the function - such as language, owner/perms and a comment/description

The final function looks like this:

```
-- Function: at_tcbc_create_ctax_mi_view()  
-- DROP FUNCTION at_tcbc_create_ctax_mi_view();  
  
CREATE OR REPLACE FUNCTION at_tcbc_create_ctax_mi_view()  
  RETURNS text AS  
  $BODY$  
  
  SELECT at_sys_drop_table('CTAX_MI_VIEW');  
  
  SELECT "AstunLocationLookup"."Name",  
         "AstunLocationLookup"."X",  
         "AstunLocationLookup"."Y",  
         "AstunLocationLookup"."UniqueId",  
         "BLPU_APP_CROSS_REF"."SOURCE",  
         "BLPU_APP_CROSS_REF"."CROSS_REFERENCE"  
  INTO "CTAX_MI_VIEW"  
  FROM "AstunLocationLookup",  
       "BLPU_APP_CROSS_REF"  
  WHERE "AstunLocationLookup"."UniqueId" = "BLPU_APP_CROSS_REF"."UPRN"  
        AND "BLPU_APP_CROSS_REF"."SOURCE" = 'CTAX';  
  
  SELECT at_wkf_geo_createpointsfromxy('CTAX_MI_VIEW', 'EASTING', 'NORTHING');  
  
  $BODY$  
  LANGUAGE 'sql' VOLATILE;  
ALTER FUNCTION at_tcbc_create_ctax_mi_view() OWNER TO postgres;  
COMMENT ON FUNCTION at_tcbc_create_ctax_mi_view() IS  
'Astun Technology custom function for Torfaen to build a Council Tax view in MapInfo format';
```

Once this function has been saved within Postgres (which is done simply by executing the code-block above as an SQL statement within the pgAdmin III utility), we are able to call it from within iShare Studio; just as with any other pre-loaded/installed function.

The iShare Studio Bits

3. Export to MapInfo

Once we have created our table, we now need to export it into a TAB file for use in iShareMaps (or elsewhere). For this we