Implementing ADQL in Postgres using PgSphere.

PgSphere is a free add-on to Postgres which extends the standard GIS capabilities of Postgres to geometric objects on a sphere. PgSphere was intended to address many of the same issues as the ADQL extension to SQL. This memo discusses how PgSphere can be used within Postgres so that ADQL queries can be handled without any special actions by the user. It describes the HEASARC's experience over the past couple of weeks, what worked and the issues that remain. Overall this approach seems very promising. This approach is one that implementers of ADQL systems in TAP or elsewhere may wish to consider.

Installing PgSphere.

PgSphere is available as free download. The build script worked without problems on our Linux Postgres installation. PgSphere 1.1 is required to support Postgres 8.4. Variadic functions, a feature available only in Postgres 8.4 were used in the ADQL implementations

Implementing ADQL.

PgSphere implements points, circles and polygons on the standard unit sphere. It provides a transformation object that will transform any of these objects from one coordinate system to another. PgSphere also supports a box object corresponding to a range in both latitude and longitude (i.e., the 'rect' object in some discussions). This type is not used in the ADQL implementation.

ADQL supports points, circles, boxes and polygons where each object is associated with a coordinate frame and where different objects may be in different frames. The ADQL box object is a special case of a polygon.

PgSphere and ADQL support functions for the distance between objects, containment and intersection and the extraction of coordinates from points. ADQL supports a function for the extraction of the description of the coordinate system from each object.

Two separate approaches were tried in implementing ADQL using PgSphere, implementing a standard geometry type, xgeom, which included delegates to the various PgSphere types, and direct translation of the ADQL constructs to native PgSphere objects. E.g., in the first approach the function

point('j2000', 10.,10.)

returns an xgeom object. The xgeom object has a PgSphere strans object representing the J2000 coordinate frame, and an spoint representing the coordinates. Each xgeom object has an strans, spoint, scircle and spoly delegates internally. Only some of these are set. E.g., when we render a circle function we do not set the spoly delegate in the object returned. Another field indicates which of the PgSphere delegates to use for the current object.

In the second approach the point('j2000', 10,10) returns an spoint object appropriately rotated by an strans object if needed. The ADQL point, circle, polygon and box functions are factories for the existing PgSphere types.

In this discussion we do not consider the REGION type because it is currently ill defined. Implementing regions requires more complex parsers. Simple regions can also be implemented using the scircle or spoly PgSphere functions. Complex regions that are described by unions or intersections of polygons probably could be done easily enough extension of the xgeom type. They will require a new compound class in the native PgSphere implementation.

Both of these approaches are straightforward to implement and generally work quite well. Query constraints are easily represented and ADQL functions can be used anywhere functions are legal in SQL. Users who wish to implement ADQL interfaces to smallish (<10K row) tables may find either approach to be entirely adequate to their needs.

There are a few special implementation details:

ADQL defines a variadic (variable number of arguments) Polygon function. SQL does not, in general, implement variadic functions, but Postgres 8.4 does allow them. PgSphere uses a clever trick to allow a function with a variable number of arguments: the constructor for polygons is an aggregate function. So when one needs to create a PgSphere polygon the implementation creates a temporary table of the points defining its edges and then executes a query against that temporary table. Some care is needed since multiple polygons may be created in a single SQL statement. The box function also populates a temporary table to create the polygon object it is transformed to. The implementation of the polygon and box functions is the only non-trivial code in the entire system. A region implementation would likely involve a significant parser. Everything else is just a wrapper on PgSphere.

The Box function is implemented in both five and six argument versions, with an optional rotation angle.

In the xgeom implementation, some functions like coord1 and distance that are defined only for points are defined for more kinds of objects.

Issues.

Two issues arise in both implementations: the representation of the geometry objects when they are in the select clause, and ensuring the positional indexes are used. Each approach almost solves one of these problems.

Select clause representations.

Postgres does not allow users to control the representation of composite types. So a query like

```
select circle('j2000', 10, 10, 10);
```

returns a string that represents either the concatenation of the elements of the xgeom type, or the native representation of a scircle object. The xgeom implementation does include a field which has the string specified in SQL. While a query run in PSQL will not show the right string output, it is very easy to fix this in the library that is doing the database query, e.g., in a TAP implemention. In Java JDBC it is just a couple of lines to do this very robustly.

Using native PgSphere objects the situation is less satisfactory. The information about the user-specified coordinate system is lost in the query results. E.g., suppose a system uses J2000 as its native coordinate system.

```
select point('galactic', 0., 90.)
```

will return an spoint object that points to the position of the Galactic pole but in J2000 coordinates. The same approach for filtering results may used to change the spoint format to the ADQL format, but non-native coordinates will not be preserved. I do not believe that always returning objects in some standard coordinate system is technically a violation of ADQL as written, though I feel that it is certainly a violation of the spirit. In this implementation the ADQL coordsys function will always return the native coordinate system.

Using indexes.

For tables too large to be conveniently kept entirely in memory the use of indexes is essential for rapid query response. The ability of a query optimizer to use an index depends critically upon how the fields used to define the index show up in the query. E.g., something as simple as

```
x+1 < 0
```

might not be able to use an index on x that

```
x<-1
```

could. In all database systems there is a certain degree of tweaking that is required. We would like an implementation where common patterns will use indexes when they are available.

PgSphere supports GIST indexes which can speed spatial queries. In PgSphere a typical query might look like:

```
select … from … where spoint(radians(ra),radians(dec)) @
                        scircle(spoint(radians(10),radians(10)),radians(1))
```

where we assume that RA and Dec are in degrees and we are looking for objects within a degree of 10.,10. PgSphere uses radians internally. The contains operator is '@'.

A user can create an index on a table using

```
create index … on … using GIST(spoint(radians(ra),radians(dec))
```

If they have done so, then the common query idiom above will use the index and should run rapidly assuming a reasonably small radius.

The corresponding ADQL query is

```
select … from … where contains(point('J2000', ra,dec), circle('J2000', 10,10, 1)) = 1
```

In the xgeom approach the query optimizer has no hope of recognizing that it can use the index. The delegation from the xgeom proxy to the spoint,scircle and spoly elements is not something that it can predict. If we are going to use indexes in this approach, then the query needs to be analyzed either by software or by the user and additional constraints that the query optimizer can recognized put in. E.g., if our databases have indexes on Dec then either the user or software could add

```
… and dec between  9 and 11
```

to the previous query. This simple constraint is recognized and used by the query planner.

Since PgSphere includes support for GIST indices, they almost work in the native PgSphere implementation of ADQL. If we have created a GIST index as above, then it will be used in a query like

```
select … from … where point('J2000', ra,dec) @ circle('J2000', 10, 10, 1)
```

a hybrid approach using a little bit of PgSphere syntax. The index would also work if *contains* were a boolean rather than a integer valued functions. Booleans are in the SQL 99 standard but not in SQL92. The little =1 seems to be the kiss of death with regard to getting the index used.

There may be ways of playing with the index creation to get around this, but none seem immediately obvious. Again some kind of revision of the SQL by the user or ingest engine seems to be needed.

Getting indexes used is a much harder problem than the rendering of the selected fields. A variety of very different schemes are used in our various databases and users cannot be expected to understand

what these are. For medium sized databases (<10M rows) indexing directly on coordinates is probably adequate, and it is unclear if the GIST indexes used in PgSphere provide any advantage. For very large systems the input parsers will need to translate the user specified constraints to something that the system can take advantage of.

In current DBMS's and presumably in ADQL implementations only certain idiomatic ways of expressing constraints can use indexes. It behooves us to make sure that we agree on what those idioms are regardless of how the data is indexed underneath.