

# Astronomy Pixel-Processing Use Cases

These pixel-processing use cases are better suited for an array data model than for a relational table model. The syntax used here is arbitrarily invented for pseudocode purposes and is unlikely to reflect actual SciDB usage.

The first is nearly trivial for an array database, the second involves some sequential processing of array elements, and the third uses complex math that would take a while to translate into pseudocode.

## Simple Instrument Signature Removal example

```
CREATE ARRAY Pixels [  
    ccdId INT,  
    x INT,  
    y INT,  
    time TIMESTAMP  
]  
(  
    flux FLOAT  
);
```

```
CREATE ARRAY DarkFlat [  
    ccdId INT,  
    x INT,  
    y INT,  
    time TIMESTAMP  
]  
(  
    flux FLOAT  
);
```

```
CREATE ARRAY CalibImage [  
    ccdId INT,  
    x INT,  
    y INT,  
    time TIMESTAMP  
]  
(  
    flux FLOAT  
);
```

```
INSERT INTO CalibImage  
SELECT Pixels.ccdId, Pixels.x, Pixels.y, Pixels.time,  
    Pixels.flux - DarkFlat.flux  
FROM Pixels, DarkFlat  
WHERE Pixels.time = :time AND DarkFlat.time = CLOSEST(DarkFlat.time, :time)  
    -- DarkFlat.time could be later than :time  
    AND Pixels.ccdId = DarkFlat.ccdId  
    AND Pixels.x = DarkFlat.x  
    AND Pixels.y = DarkFlat.y;
```

---

## Detection example

```
CREATE ARRAY Footprint [
    ccdId INT,
    objectId INT,
    time TIMESTAMP
]
(
    x INT,
    y INT
);

CREATE ARRAY Source [
    sourceId BIGINT,
    time TIMESTAMP
]
(
    minX INT,
    maxX INT,
    minY INT,
    maxY INT,
    centroidX FLOAT,
    centroidY FLOAT
);

WHERE time == :time { -- Select a particular image
    FOREACH ccd { -- Iterate over all CCDs
        c = CONVOLVE(PsfKernel, CalibImage[ccd])
        -- PsfKernel is a product of calibration and is a
        -- small two-dimensional array
        id = ARRAY(DIM(c), INT, 0)
        -- New array to hold object ids with same
        -- dimensions as array c, filled with zeros
        fixup = ARRAY([*, 2], INT)
        -- New vector holding pairs of ids that are part of
        -- the same object, initially empty
        nextId = 1 -- Id of the next object to be found
        FOR x, y { -- Iterate over all pixels, y fastest
            IF c[x, y] > :threshold { -- We have an object pixel
                IF id[x, y-1] { id = id[x, y-1] }
                ELSIF id[x-1, y-1] { id = id[x-1, y-1] }
                ELSIF id[x-1, y] { id = id[x-1, y] }
                ELSIF id[x-1, y+1] { id = id[x-1, y+1] }
                ELSE { id = nextId; nextId = nextId + 1 }
                id[x, y] = id
                -- If it's adjacent to another object
                -- pixel, use that id; otherwise,
                -- assign a new one
                IF id[x-1, y+1] && id[x-1, y+1] != id {
                    -- If we're touching an existing
                    -- object, either by bridging or by
                    -- projecting to one side, remember
                    -- that we have to fix this up
                    APPEND(fixup, (id, id[x-1, y+1]))
                }
            }
        }
    }
}
```

```

FOR x, y {
  -- Fix up object ids
  oldid = id[x, y]
  newid = LOOKUP(fixup, oldid)
  WHILE newid {
    oldid = newid
    newid = LOOKUP(fixup, oldid)
  }
}
FOR x, y {
  -- Fix up object ids
  oldid = id[x, y]
  newid = LOOKUP(fixup, oldid)
  WHILE newid {
    oldid = newid
    newid = LOOKUP(fixup, oldid)
  }

  INSERT INTO Footprint[ccd] VALUES (oldid, x, y)
  -- Remember pixels that are part of an object
}
INSERT INTO Source SELECT
  GENERATE_ID(),
  MIN(x), MAX(x), MIN(y), MAX(y),
  SUM(c[x, y] * x) / SUM(c[x, y]),
  SUM(c[x, y] * y) / SUM(c[x, y])
FROM Footprint[ccd] GROUP BY id
}
}

```

## Difference imaging example

There is an additional use case from difference imaging that is much more complex. It involves extracting a template sub-array, re-gridding the template from spherical to rectangular coordinates, extracting footprints (similar to the detection algorithm above), solving linear equations to determine convolution kernels, finding principle components of those kernels (eigen-kernels), fitting the coefficients of the eigen-kernels using spatially varying functions, and then convolving an image with the fitted spatially varying kernels, producing a final output image.