LDS Reblock Process

Process

1. Topo Shapefile Reader
2. Reblock Layers
    1. Merge features across layers
    2. Generate/Associate/Persist LDS UFIDs
3. Output Shapefiles (for LDS Import)
3. Create Revision
4. Insert Differences
5. Complete Revision
6. SourceAPI changes to LDS

Connectivity

Topo Shapefile Reader

Shapefile reader functionality will include the ability to manually initiate the reblocking process or be set to poll/periodically-check content changes in a reserved directory when it will read any changed shapefiles into PostgreSQL operating database.
NB. A current OGR limitation requires that layers must be converted row-by-row to preserve SRID. This is the slowest part of the process and will need to be optimised somehow

Reblock Layers

Feature Reblock

Layer features first identified as reblocking candidates needing to be joined where they have been split along a mapsheet boundary.

Criteria for reblocking

Features are considered to be reblock candidates if they touch on a mapsheet boundary. The principle method to identify these candidates is the PostGIS function ST_RELATE(f1,f2,'<relate-string>') where relate-string is one of [FF2F11212, FF1F00102] for Polygon and Linestring repectively.
Candidate features must share a NS/EW boundary coincident with a mapsheet edge. (Candidates with a potential Mapsheet Edge are filtered using the formula.

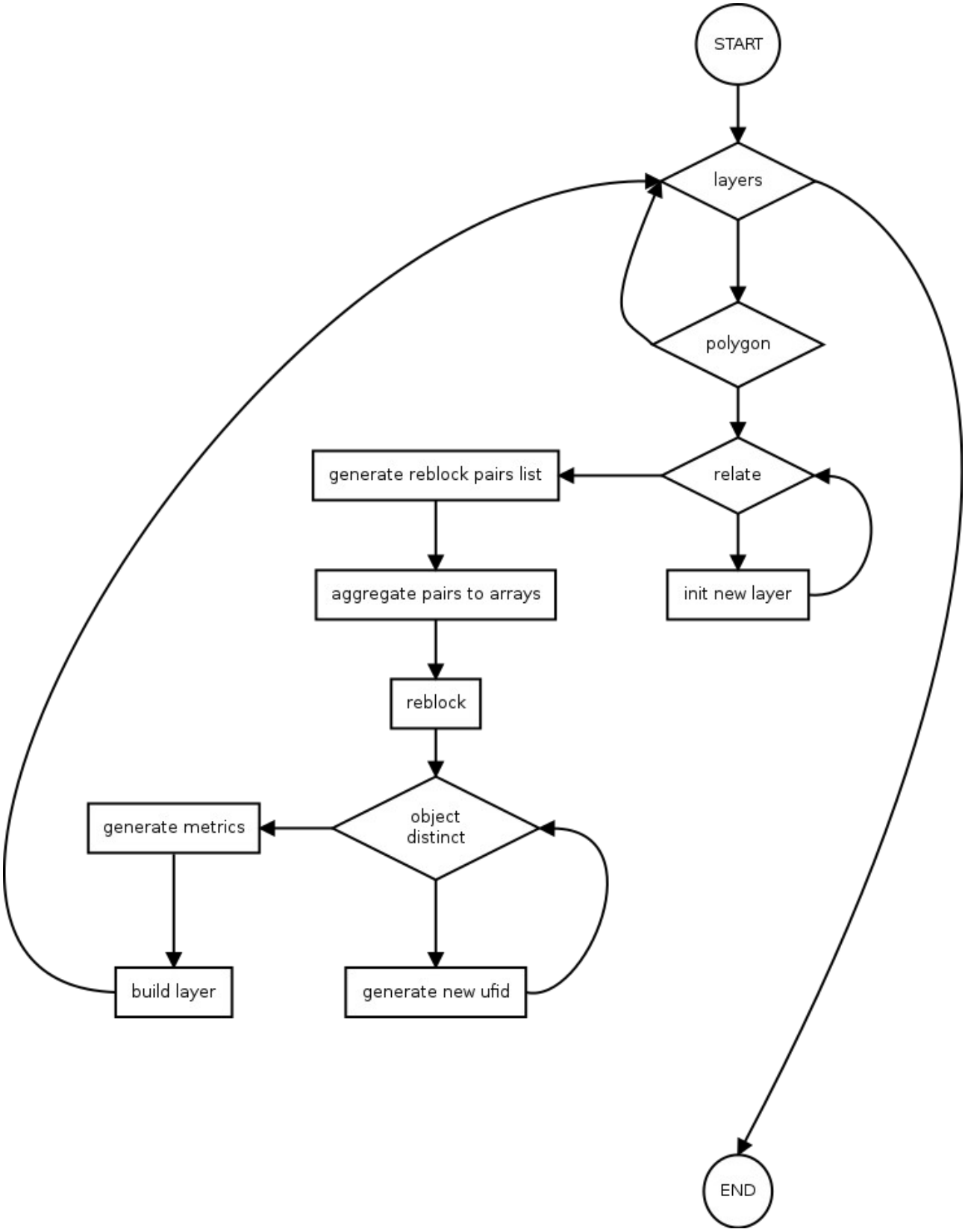
```
ST_Extent(feature::geometry) % 1000 == 0
```

and identified if they meet the following criteria

```
NScoordinate == 6000 + 36000i  : 133 < i < 172
EWcoordinate == 4000 + 24000i :  45 < i < 87
```

Candidate features must share 'common' non-null attribute values.

A list of feature pairs is created and aggregated into a combined list containing all feature components required to reconstruct an original feature allowing PostGIS to employ the Cascaded Union algorithm when actioning the ST_UNION function.



LDS UFIDS

Unchanged features will retain their original Topo UFID. In the case of reblocked features a new composite UFID (LDS UFID) will have to be constructed and maintained.

As a rule, features shoud not be renamed as long as they are “logiclly consistent”. That is, less significant changes to a feature should not result in UFID re-assignment.
Significant changes and in cases where original features are split and a new UFID has to be generated. To facilitate this the UFID's of the composite parts of a merged feature will be stored. This UFID list will be used for identification of composite features between releases to track changes or the extent of changes.

Rules governing changes to a composite UFID will be based on an agreed measure of change to the orignal feature or the features making up the final composite.

Proposal 1

Topo control source UFIDs and determine whether a feature has changed to an extent requiring a new ID. Given that composite features, though split across mapsheet boundaries, are considered individual objects, UFID changes should propogate through the split/merge process. Generating a LDS UFID will take into account only UFIDs in the composite feature set. When a component ID or the number of components changes this will trigger a corresponding automatic change in LDS UFID. This has the desirable affect of seperating the merge process from the generation, tracking and maintenance of UFIDs.

```
LDSUFID   = func(f1.UFID,f2.UFID, … fn.UFID)
LDSUFID_1 = func(f1.UFID,f2.UFID, … fn.UFID,fn+1.UFID)
LDSUFID_2 = func(f1.UFID,f2.UFID, … fn.UFIDchange1)
```

The undocumented postgres function “hash_numeric(int)” would be used for this purpose, applied sequentially to each component UFID bitwise OR'd with the previous hash result.

```
for UFID in UFID_LIST:
      hashval_{i+1} = func(hashval_i || UFID)
```

Collision resolution would implemented by rehashing until the collision is eliminated

```
while hashval_i == hashval_j :
      hashval_i = func(hashval_i)
```

Pros
Low maintenance.
Fast calculation.
Deterministic and repeatable.

Cons
Only changes to feature UFIDs will trigger a LDSUFID update. (Unless also hashing geometry)
Hashing has a (small) chance of causing data collisions.
Collision resolution has a (v.small) chance of causing cascading ID changes.

Proposal 2

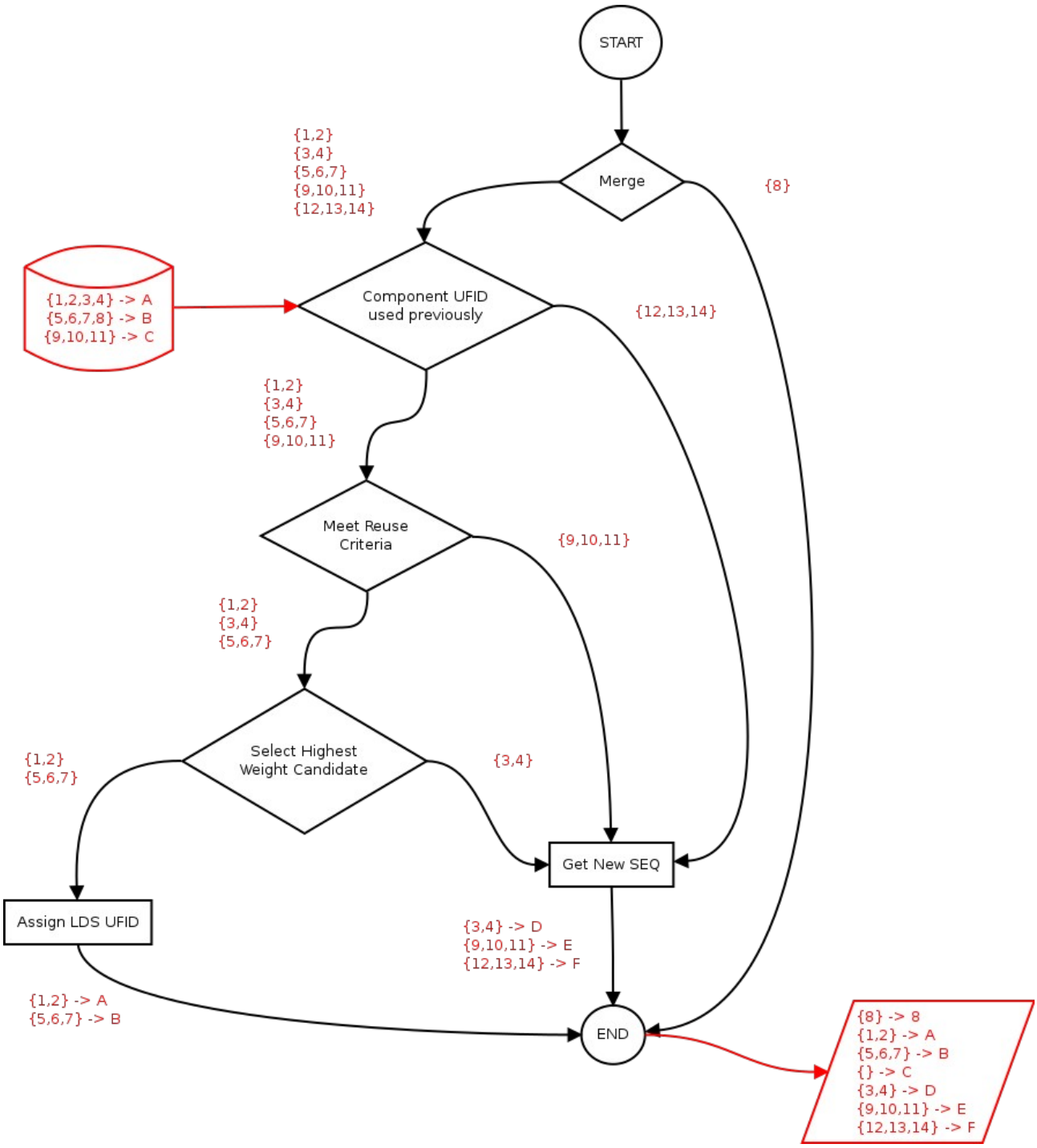
The LDS team will track changes by measuring feature differences and issuing sequential LDS UFIDs to any composite feature meeting some predefined change threshold.

Changes monitored will be different for each feature type. For Polygons, Area and Perimeter will be monitored. For LineStrings only Length will be monitored. Differences between these respective measures for each composite feature will be saved and used for comparison purposes at each release. (For easily/fast calculated metrics these can be regenerated inline and features compared directly)

Provisionally UFID generation will be triggered on the count of composite UFIDs changes and/or a percent change caluclation as outlined.

To identify features that both change but also reuse an existing LDS UFID the following criteria will be used.

1. Reuse candidates must merge[†] across a mapsheet boundary.
2. UFIDs are reuse candidates if their UFID component sets intersect AND their geometries meet the Area-Threshold criteria AND their geometries meet the Perimeter-Threshold criteria. (For LineStrings see Length-Threshold)
3. If a number of candidates match a particular reuse criteria the candidates will be ranked using a weighting function and selected according to the maximum value this returns.



Reuse criteria will be calculated using difference threshold formula

Polygon

$$ST\_Area([f1,f2,f3...]) - ST\_Area([f1',f2',f3'...]) > THRESHOLD_{Area} \;\&\&$$
$$ST\_Perimeter([f1,f2,f3...]) - ST\_Perimeter([f1',f2',f3'...]) > THRESHOLD_{Perimeter}$$

LineString

$$ST\_Length([f1,f2,f3...]) - ST\_Length([f1',f2',f3'...]) > THRESHOLD_{Length}$$

When a LDS UFID is to be split the same threshold function will be used to decide on reuse

elegibility for a particular feature set.

Reuse Selection
MAX(
$Weight_{Area}$*ST_Area([f1,f2,f3...])+$Weight_{Perimeter}$*ST_Perimeter([f1,f2,f3...]),
$Weight_{Area}$*ST_Area([f5,f6,f7...])++$Weight_{Perimeter}$*ST_Perimeter([f5,f6,f7...])
)

If a new LDS UFID is required a new value will be generated from a sequence over a unique non intersecting range.

Notes.
UFID changes are in many cases determined by Topo workload. That is, when it is easier to re-sample an entire area instead of editing existing features, this will be preferred. This will cause all UFIDs to be re-generated. In addiion, certain minor edits such as attribute changes will trigger UFID changes. Only in limited situations where boundaries/lines are undergoing simple realignment will UFIDs persist. The implication here is that any significant feature changes will be accompanied by new UFID and accurate reuse will be difficult (requiring geometry comparisons) and largly unnecessary.

Pros
Sensitive to observable change condtions.
Well defined change criteria.
No chance of collision

Cons
Requires long term maintenance of UFID tables.
Difficulty tracking merged geometries across UFID changes.
Time consuming UFID change metric calculations.

Create Revision

Comparing a reblocked layer with previous snapshots using BDE versioning tools will be used to generate changesets.

Insert Diffs

Differences resulting from comparisons can be used to create revisions

Complete Revision

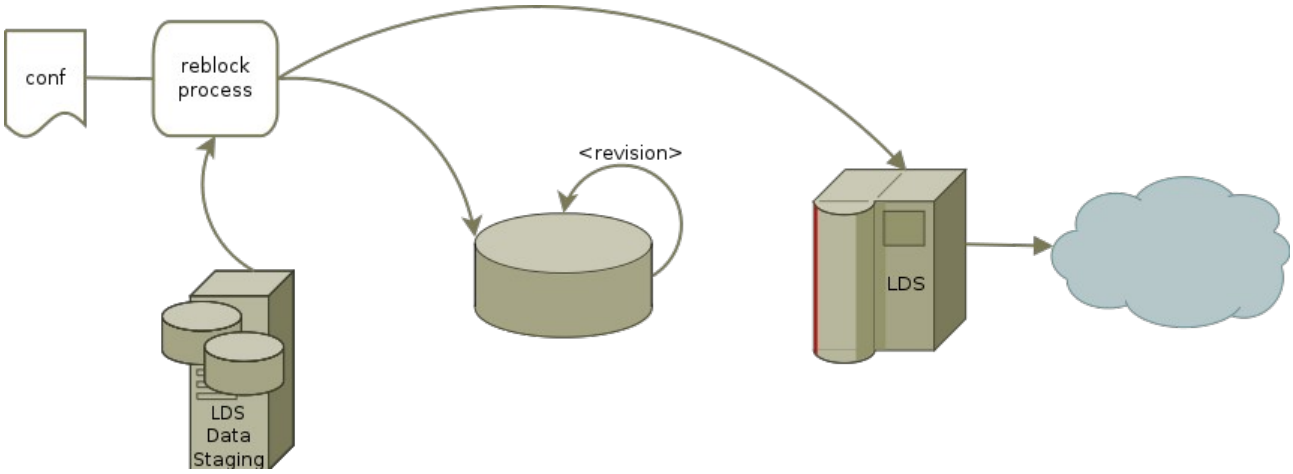
Complete the revision

SourceAPI Upload

Newly minted layers are uploaded to LDS using the SourcesAPI

Connectivity

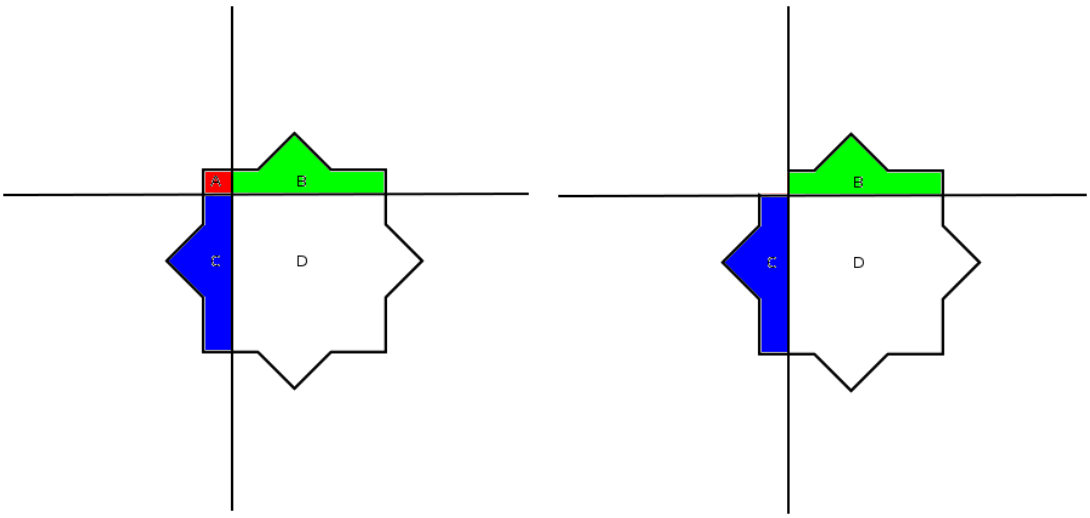
Centering around the reblocking processor, source data will be periodically dropped into lds_data_staging. Polling on a named directory will inform the reblocker of data to be processed and trigger the file conversion stage of the provisioning operation.



Reblocked data will be stored into temporary tables on a PostgreSQL database and revisioned in situ. Since BDE processing provides revisioning capability the reblocking functionality would be most easily located here.
Uploading to the LDS servers can occur anytime thereafter or during a predetermined upload window. Using the SourcesAPI this client side initiated process and can be built into the reblock process.

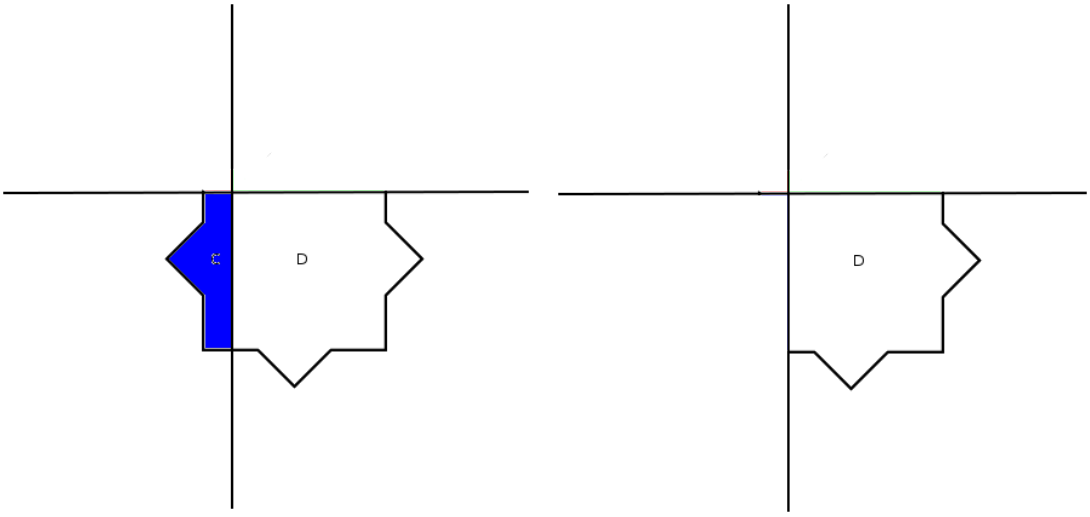
Appendices

Example of Feature Dis-Assembly



Original Composite Feature.
count-count' = 100%
area-area' = 100%
perimeter-perimeter'=100%

Section A Removed.
count-count' = 75%
area-area' = 95%
perimeter-perimeter'=100%



Section A,B Removed.
count-count' = 50%
area-area' = 85%
perimeter-perimeter'=90%

Section A,B,C Removed.
count-count' = 25%
area-area' = 75%
perimeter-perimeter'=80%

Example of Processor Pseudo Code

```
start

for shapefile in directory:
      import layers from shapefile

for layer in layers:
      for features in layer:
            pairs_list = function query_features(features)
            aggregates = function aggregate_pairs(pairs_list)
      singles = features with feature.id NOT IN aggregates
      multis = features with feature.id IN aggregates
      output = function merge_features(multis)

      function test_for_collisions(output)
      revised_output = function save_associations(output)


for new_layer in revided_output:
      export new_layer to shapefile

end
-----

function query_features(features):
      filter NIfeatures from features:
            pairs_list += function match_features(NIfeatures)
      filter SIfeatures from features:
            pairs_list += function match_features(SIfeatures)
```

```
      return pairs_list

function match_features(features):
      for feature1 in features:
            for feature2 in features:
            select feature1 with bound on mapsheet North
            select feature2 with bound on mapsheet South
            if feature1 touches feature2:
                  pairs_list += (feature1,feature2)

            select feature1 with bound on mapsheet East
            select feature2 with bound on mapsheet West
            if feature1 touches feature2:
                  pairs_list += (feature1,feature2)
      return pairs_list:

function aggregate_pairs(pairs):
      for feature1, feature2 in pairs:
            for feature3, feature4 in pairs:
                  if feature2 == feature3:
                        aggregated_pairs_list += (feature1, feature2, feature4)
      return  aggregated_pairs_list

function merge_features(multis):
      for multi in multis:
            merge = function merge_multi(multi)
            if merge NOT error:
                  new_ufid = function gen_ufid(multi)
                  combined += (new_ufid,merge)
            else:
                  log (failed merge)
                  for part in multi
                        combined += (part.ufid, part)
      return combined

function merge_multi(multi):
      return *st_union(multi)

function gen_ufid(multi):
      new_ufid = *hash_numeric(multi)
      OR
      new_ufid = function compare_saved_merge(multi)
return new_ufid

# blue : save option
function compare_saved_merge(multi):
      new_metric = function generate_metric(multi)
      old_metric = function read_saved_metric(multi)
      comparison = function compare_metrics(new_metric,old_metric)
      if comparison:
```

```
            new_ufid = max(multi)
            function write_new_multi(new_ufid, new_metric, multi)
      else:
            new_ufid = function read_saved_ufid(multi)
      return new_ufid

function compare_metrics(m1, m2):
      if m1.c <> m2.c:
            if m1.p - m2.p > perimiter_threshold
            and m1.a - m2.a > area_threshold:
                  return TRUE
      return FALSE

function read_saved_metric(multi):
      ?something like?
      for saved in DB.saved_multis:
            inter =  multi.ufid_list intersection saved.ufid_list
            if count(inter) / count(saved.ufid_list) > match_threshold:
                  return generate_metric(saved.multi)
      return 0

function generate_metric(multi):
      c = *count(multi)
      a = *st_area(multi)
      p = *st_perimeter(multi)
      return (c,a,p)

# green : hash option
function test_for_collisions(output):
      if count(ufid) in output>1:
            output.ufid = gen_hash(ufid)

function save_associations(output):
      for ufid, ufid_list in split(output):
            function save_to_associations_table(ufid,ufid_list)
      revised_output = output - ufid_list
      return revised_output
```