# 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. Versioning
4. 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.

![A description...](data:None;base64,)

#### 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:

hashvali+1 = hashfunc(hashvali || UFID)

Collision resolution would implemented by rehashing until the collision is eliminated

while hashvali == hashvalj :

hashvali = hashfunc(hashvali)

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.

A description...

Reuse criteria will be calculated using difference threshold formula

Polygon

ST\_Area([f1,f2,f3...]) - ST\_Area([f1',f2',f3'...]) > THRESHOLDArea &&

ST\_Perimeter([f1,f2,f3...]) - ST\_Perimeter([f1',f2',f3'...])>THRESHOLDPerimeter

LineString

ST\_Length([f1,f2,f3...]) - ST\_Length([f1',f2',f3'...]) > THRESHOLDLength

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(

WeightArea\*ST\_Area([f1,f2,f3...])+WeightPerimeter\*ST\_Perimeter([f1,f2,f3...]), WeightArea\*ST\_Area([f5,f6,f7...])++WeightPerimeter\*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.

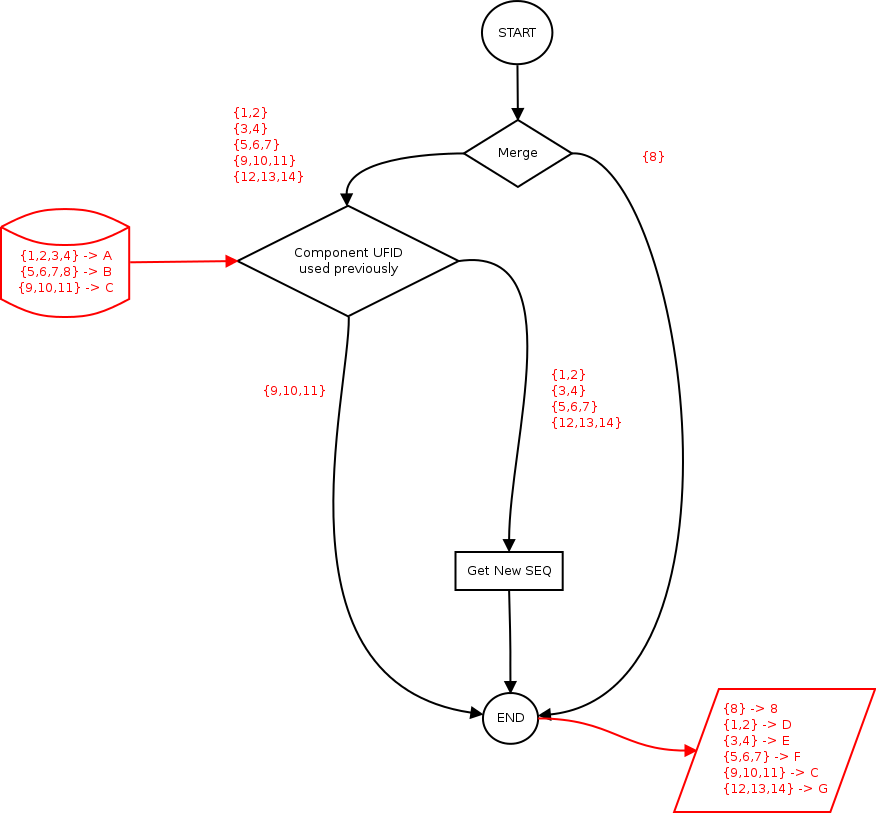
Difficulty tracking merged geometries across UFID changes.

Time consuming UFID change metric calculations.

#### Proposal 3 – Simplified Sequence

This proposal uses sequence id generation without requiring geometry characterisation or comparison.

Stated simply; new LDS UFIDS will be generated off a PostgreSQL sequence and stored alongside an array of their component UFIDs. Subsequent feature merges will query this list matching component UFIDs returning the stored LDS UFID if found or will generate a new sequence LDS UFID.



### Versioning

Versioning is achieved in the python import script by wrapping database queries in calls to turn versioning on and off.

While the mechanism for reblocking remains import shape to PG, build merged PG table, export merged PG table to Shape maintaining versioning isn't required and is disabled by default.

Note.

1. Testing required to see how table replacement affects versioning
2. Test versioning over multiple table updates, “SELECT reblockall();”.

### 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.

![A description...](data:None;base64,)

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***

|  |  |
| --- | --- |
| A description...  Original Composite Feature.  count-count' = 100%  area-area' = 100%  perimeter-perimeter'=100% | A description...  Section A Removed.  count-count' = 75%  area-area' = 95%  perimeter-perimeter'=100% |
| A description...  Section A,B Removed.  count-count' = 50%  area-area' = 85%  perimeter-perimeter'=90% | A description...  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