MultiResolve

Layout and groundwork

The fundamental workings of Resolve are, quite correctly, handled by stored procedures executing within the database server. While some of them may be hampered in performance due to extensive logic and looping, implementing the same at the application level would likely require much more time, be more error prone and much more difficult to correct.

It has been shown that Resolve currently does not put undo load on the production database server, even when supplied with a lengthy list of instructions[[1]](#footnote-2). Also shown, in a test environment, is that at least five instances can run simultaneously. Neither the test server nor the desktop client running these Resolves was particularly taxed.

What follows is an overview of a proposed overhaul of the Resolve facility, with light detail on expected implementation. It begins with a list of issues requiring more research. While the results of that research may have an impact on one or more of the proposed features, the project is expected to be straight forward and highly likely to achieve the desired result of improve throughput.

Typography: C# code snippets and class names use monospace font and references to database components are underlined.

# Parts unknown

1. Is it sufficient to do the look-ahead as part of the insertion trigger? Early in Resolve's processing of an instruction, all the APIDs are collected based on the personIds in the instruction. It is entirely conceivable that these could be collected during instruction creation.
2. How expensive is it to do AB instruction then later BC instruction? Upwards of 2 seconds per, apparently, but how often does this happen.
3. Full mechanism for creating instructions and all implementations. There's a trigger somewhere…
4. Yes/No to additional allowed value for mappinginstruction.success? Maybe even rename that column?
5. The current (manual) start mechanism is a faint memory at this point but it did strike me as odd, misplaced.
6. Idiomatic .Net code: Is the "Repo" a common DAO model in C#-land?
7. Are any of the Resolve stored procedures used in other code? Or does any other code use Resolve.APIDs
8. How is Command = 'Extract' handled. The string extract is not found anywhere within the code base for Resolve. Strictly an Extractomerge issue.

# Hurdles (and vaults)

## Resolve.APIDS

This single, real table designed to hold all the archivePersonIds associated with the two ids within the instruction is the primary bottleneck to running multiple Resolves. Any and all Resolves want to read, write and delete this small table: contention is guaranteed and locks inevitable.

Replaced this single point of contention by having each run of resolve establish its own temporary table of the exact definition as Resolve.APIDS. This then required a consistent edit in the store procedures. This was done for the test environment. An excellent task for sed, er, string-select or some such…

## Queue Manipulations

Handling a single instruction can affect the MappingInstruction queue in either or both of two ways.

### Related instruction

Early in the processing of an instruction Resolve looks ahead in the queue to see any other instruction(s) deal with an input. If so these are incorporated in to the current instruction.

It will be preferable to restructure the queuing mechanism such that the act of adding an instruction does the work of finding any associated entry or establishing a new one. This should obviate the need for the back-up queue (MappingQueue). See 4

### Parent/Child interaction

A change in any temporal dimension of a parent can potentially affect the child's history. When such a change is detected the child needs to be added to the queue as it's own instruction. And vice versa?

This does not need to be handled any differently than to make use of the new queuing mechanism.

# C# Retooling

The existing C# components are well aligned with database structures and can be extended to play nicely in a multi-threaded environment.

## MappingInstruction queue

The existing MappingInstructionRepo class knows how to read/update instructions and hands them off to Resolve one after another. We'll isolate it a little more, making a MappingInstructionFactory to read batches of instructions, marking those as inprogress and hand each batch off to ResolveManager as Collection <MappingInstruction>. It will always have at least one batch prepared, at most two.

Note that the proof of concept made changes to this code base, as necessary to avoid dead-lock situations. Anything having to do with instruction injection had to be d/circumvented./

### Plan for mapping instruction handling

* separate db connection
* tight transaction loop, reading MappingInstruction in batches
	+ set pending:

#drop table if exists #miworkingset;

creat table #miworkingset as

select top :resblock miid from MappingInstruction

where success is null order by priorityval, requestDate;

update mappinginstruction m set working = 'W'

from #miworking u where m.miid = u.miid;

* :collect instructions:

select \*

from MappingInstruction m join #miworkingset u on m.miid = u.miid

where status = 'W'

order by miid

* build instances of MappingInstruction
* hand off the collection of instructions to ResolveManager

## Resolve execution

The current Resolve class is the workhorse of the current code base. It calls all the stored procedures, harvesting those results then deleting and recreating the (single) person. We'll extend this slightly by shifting its interaction with the queue into ResolveManager and move the stored procedure work into Resolver working on each specific instruction in a batch. It is instances of this class, Resolver, which will be run concurrently, each on it's own connection to the database, each with its private resolveAPIDS table.

### Plan for resolve execution

* ResolveManager defines and maintains N Resolvers in a ring structure
* per new Resolver
	+ connect
	+ make "proprietary" id table (create table #resolveAPIDS(..))
* manager requests block of instructions from MappingInstructionFactory
* iterate over instruction block
	+ get next available Resolver
		- Resolver.isAvailable(false)
	+ use connection from current Resolver
	+ hand instruction to Resolver
	+ truncate table #resolveAPIDS
	+ write MappingInstructionEntry (APIDs) to #resolveAPIDs
	+ transaction boundary
		- Resolver.run() exec stored procedures, (re)-write person as usual
			* minus no look-ahead
			* re-work getting all APIDs
			* with or without timing?
		- update mappinginstruction.status to 'S' (or 'F')
	+ Resolver.isAvailable(true)

# Database Retooling

## Mapping instructions

A feature (or is it a bug) of the current processing is the "look ahead" feature which scans the MappingInstruction table for instructions "related" to the one in hand. Any found are processed along with the current instruction. It was found that looking ahead when there too many instruction became too expensive. To guard against that, "extra" records are moved to MappingQueue table (and back again) to keep MappingInstruction under approximately 5000 records. This to-and-fro happens on a five minute interval, external to the Resolve application.

One shortcoming of this is that "related" instructions may be split between the two tables, with those in the MappingQueue not seen by the look-ahead within Resolve and hence a duplication of effort once the erstwhile related instructions are returned to MappingInstruction. To avoid this, effort must be put into preprocessing instruction insertion.

### Plan

Add a dependent table to MappingInstruction to collect overlapping ids (maybe MappingInstructionEntry). The dependent table will hold the accumulated (person, archivePerson?) ids up to the point of execution. Naturally, once the instruction is picked up by Resolve, nothing can be added to it. There's a style question: keep the person1,person2 structure and only add extra ids to MappingInstructionEntry, or break it out completely. (The latter would be my vote.)

This affects any mechanism which generates mapping instructions. It may change Extractomerge as well.

## SQL instuctions

The less the better in this case.

### Plan

Must replace all SQL references to Resolve.APIDs table with a reference to (a similarly named) temporary table. This mainly affects the store procedures in the Resolve schema. Each database connection will have it's own version of this table. Each Resolver will have its own connection. Processing each instruction will clean-out and refill this temporary table and process the instruction in a single database transaction.

# Exit strategy

Since we'll be setting blocks of instructions to some inprogress status, there may be a desire to reset those if Resolve is turned off before finishing the block. There is currently a trap for control-c from the console but hopefully this will run as a service with a configuration file so there won't be a console at which to hit control-c. One (blunt) mechanism is to read a file periodically. A more refined approach would be to trap the WM\_CLOSE message (which is sent by taskkill for instance). Perhaps Extractomerge could be trained to send such a signal. This would be inter-machine interaction but that's what .Net is all about.

On the other hand, the restart of Resolve could simply make the first block of instructions from all those inprogress leftovers.

# Testing, 1 2 3

1. Every stored procedure used by Resolve which deals with\_Resolve.APIDs\_ table will need to be modified. We have test harness for store procedure test in place [[Resolve\_Testing]<https://hci-ppriis-new.hci.utah.edu:8443/Resolve_Testing>]. Along side the original, this harness will need to able to run a slightly modified version of the original.
2. Verify that therr is no negatively impact on store procedure execution speed
3. Repeat test #1 against multi-threaded version, including assessing overall throughput characteristics
1. An instruction is a request to either a) merge two persons into one or b) re-assess

all the known information on a single person. These live in MappingInstruction table. [↑](#footnote-ref-2)