Salesforce Performance and Scalability patterns

# Introduction

This document is aimed at the developers and attempts to explain some key aspects around obtaining reasonable performance and avoiding governor issues on Salesforce. This is primarily a discussion on programming patterns and concepts. It does not cover issues around higher-level design, which is also important, but is far too wide to cover here, and is also often beyond what an individual developer can influence. Much of what is discussed here should be basic bread and butter stuff for any senior software developers and architects and is aimed primarily at junior/mid-level developers. There is only so much I can cover in a single document; hence it is expected that if developers do not fully understand some aspects they will talk to one of the senior developers or do some further research.

# The Database

There are multiple reasons one can have substandard performance on Salesforce. However, most major performance, scale and governor issues tend to be down to executing far too many DML operations. So far, with Citations Org this has certainly been overwhelmingly true. Even issues that are not directly DML governors tend to occur as a result of too much DML causing a chain reaction of logic; resulting in more queries or CPU time than expected.

This is usually due to poorly thought-out designs – and those designs can be business designs or programming designs. The PMs/BAs can hand you designs that require you hammer the database, and equally you, the developer, can hammer the database when you should not be doing so.

As a developer you can, should and must avoid hammering the database more than necessary. Even where the design provided by a BA/PM demands unnecessary database operations (which they invariably have done) then you still have an obligation to mitigate this with an overarching architecture that deals with this issue as best as possible.

To be clear this is not a Citation specific issue, any Salesforce work requires this effort to maintain the system and avoid a slide towards constant limit exceeded errors and unhappy users. That is not to say that this is the only issue and if you handle this then all performance issues will go away, but it is arguably true to say that if you cannot deal with this then you will never have a performant and scalable set of business processes.

## Salesforce Architecture

There are a couple of key aspects of how Salesforce work that developers need to grasp before engaging in any technical work.

### Business logic timing

At the core of much of Salesforce is the concept that business logic runs at the point of insert, update or delete of records. Most of the technologies you may use are explicit on this, be they Apex triggers, workflows, validation rules, process builders or flows. Salesforce requires that you design for any logic being triggered at the point of DML operations occurring rather than via functional APIs/Services.

It is possible to design very different approaches, for example to try and push for a more Service oriented architecture or move as much logic as possible into asynchronous executions. In terms of addressing performance/scale both of those approaches can be disregarded for now.

Fully functional asynchronous programming is hard, even on Salesforce which may make it look mechanically easy. Pushing logic out into such code does not solve the issue we are discussing here, it simply delays the inevitable by giving a bit more headroom, whilst also adding a lot more complexity to think about and design for. NB async mechanisms will likely form an important part of any highly scalable system eventually, but I am not discussing such here – this document is about the basic coding patterns all developers on Salesforce need to understand and follow, sync or async.

It is also possible to produce service-based APIs that can be called to perform business logic, and such a design is by far and away the most performant and scalable architecture (if designed well). However, that might require an almost total rewrite of everything – for example ditching all native UIs to route through custom UIs and services etc. It is, also, almost impossible to avoid some business logic running at the point of a DML operation, if nothing else you simply cannot avoid all those managed packages and certain Salesforce features doing more direct DML. There are further solutions to that issue as well, but at the cost of more code complexity and architecture. Such an architecture would probably look very different to some of the sample code I show later, but nevertheless the principles discussed are still the same.

In summary, any technical architecture should start on the basis that any DML is expensive and that we need to do everything we can to avoid it as much as possible. At the same time as we need to understand our logic will run on any DML operation, so we need to be reasonably efficient in avoiding unwanted processing when our logic is fired.

### Trigger batch size

The next thing to understand is that, for all practical purposes, any operation that occurs on the database Salesforce executes your logic in chunks of no more than 200 records. If you have written logic for Tasks and something updates all 600 Tasks related to an account (as the account trigger ran) then your Task triggers (or Flows, process builders etc) will run 3 times, once for each group of 200 records.

The importance of that is that it makes database operations even more expensive, as your logic potentially executes many times rather than once, which means that you will potentially run many more queries than expected, or many further updates if your trigger updated further related records, as well just taking longer. It is common to hit query related limit issues due to running the same queries over and again in such bulk operations. If your task trigger performs five queries and updates related Cases for example, then an account with 600 Tasks might run a total of 15 queries and update lists of cases 3 times via that trigger.

### Recursion

This is more a side issue, but it does relate to some later discussion.

At some point you may hit the issue that your logic runs for a second time in the same transaction, because there has been a chain reaction of DML, e.g., your Task trigger updated a Case and the Case Trigger updates the Tasks in turn; causing the Task trigger to fire again mid-way through the initial Task execution.

It is important to understand that whilst this is bad, you should not attempt to avoid rerunning logic using simple static flags. Our code base is littered with these, and every one of them is flawed and almost certainly causing data corruption of some description. Skipping logic means we may not be setting default values or validating values or updating related records etc and is a genuine issue, as it is a silent bug that is often not spotted until long after the event. The approach taken so far results in this risk occurring on a lot of DML where it was not needed. If there is a problem with recursion that is not solved by following the basic rules and patterns here, then it needs to be solved in a focused way for that specific issue.

If you follow some simple guidelines around performant programming on Salesforce, then most of these scenarios will resolve anyway. Not all of them, there are some oddities on Salesforce which make this at times hard, but the simple check for having run once is almost always a far more insidious problem.

## Technology Choices

Salesforce provide a plethora of choices on how to implement logic, from Workflows and process builder to Flows and Apex, or even Formula fields and roll ups. The first thing that needs to be understood is that to put in place a mechanism for controlling database interactions you largely need to route through a single technology choice. As all these technologies run at separate times during a DML operation and in different ways, with limited mechanisms of collaborating then you simply cannot control how you interact with the database within a mixed technology approach. Whilst some of these choices do allow a certain level of collaboration with Apex calls these are not up to the job of doing what is needed, or at least without some serious thought and effort.

As Apex allows us to do almost anything we want, including things we cannot do from other technologies then that is the technology of choice. It also must be said that as things stand Apex is simply faster anyway, even Flows (the latest SF declarative tool) cannot outperform Apex and Flow certainly does not have the flexibility to handle complex systems with many database interactions in the same way Apex can.

Apex therefore should be the choice for almost any business logic. That is not to say the other technologies should be totally avoided, at times they can do things that we care less about and do it easier. Obvious choices for still using wherever reasonable are formula fields and roll ups. Outside of that Flows should be preferred over the Process builders or Workflows.

### One Trigger to Rule them all.

This is really a follow on from the above. Just as it is hard to control your database interactions when you have too many disjointed technologies, so it is also hard to control when you have too many triggers firing independently. Ignoring third party triggers we should nearly always have only a single trigger per object type. This trigger should do as little as possible beyond delegate to some business service class as soon as possible.

There is a more interesting topic around single trigger and packaging, as in if Citation uses packaging to modularise the code base. As we do not do that, and it does not seem likely that we will for long time, if ever, then I am ignoring that discussion.

## Basic Enterprise Patterns

Understanding some basic ‘enterprise’ patterns is critical. The essence of these were covered some time ago, and I am not going to revisit that ground here. A lot of it just comes down to the standard software engineering layering; presentation, service, database. Once you start to layer in a reasonable way then a lot of useful extra patterns open that can make a significant difference to controlling your database interactions.

There are some Trailheads on enterprise patterns and Apex, though be aware that (as with many trailheads) they are often simplistic and a starter into the topic, with code samples etc that may not actually be ‘good’ when taken too literally.

The most important aspect for the purposes of this document is that business logic and database logic are in distinct layers, so services do not interact directly with the database. The importance of this cannot be overstressed – if you follow the naïve approach that has been adopted previously of having all database operations hard coded in line with the business logic, or presentation logic, then it is almost impossible to control for governor usage, performance, and scalability.

### Basic Layering

The following is high level view of how the system should be structured. Any box does not map of necessity to specific classes, but more areas of responsibility which may well involve multiple classes.

Diagram

Description automatically generated

The arrows represent the flow of logic, so a trigger calls a service layer which will call the database layer. The green boxes are the ones we are interested in, but the blue boxes are to represent that by focusing on some business logic layer (Service) we have logic that can be readily ran from anywhere. A calculate quote service could be called via some quote related trigger, or a UI or a batch etc.

I am not going to discuss the trigger layer here, that has been dealt with elsewhere. The rest of this document is focused on the interactions between a service and the database layer. The trigger is highlighted above, as it is the means by which we execute, and hence entry point, into all our trigger-based logic.

# FFlib

Within the Salesforce ecosystem there is a well-known open-source library that is also centred around a lot of what I will discuss, namely the concept of enterprise patterns and layering: FFLib. I would be wary of taking on the FFLib library though, it is arguably heavy weight and over complex for what Citation needs, and it is based on some concepts that over time have been dropped by the authors due to performance issues etc.

The sort of developers who have come from a more ‘configuration’ background with minimal software engineering knowledge will struggle. Developers need a strong understanding of why such patterns exist, when how and why to use them, as well as a good understand of OO principles etc. For example, there is a unit of work class in FFLib that is almost 1000 lines long, with a lot of abstractions covering all sorts of scenarios, and some classes like the factories and query builders can be bad for performance if you use them for the wrong reasons.

*That said, In the interests of transparency, I must admit to strong biases here, which should be borne in mind by anyone else looking to decide. Having worked with the author/company that introduced them and having had to use/implement code based on them I have quite strong views on certain aspects of them. Most of those concerns centre around performance and scalability, although there are also other aspects of it that I think cause issues, especially around its focus on service locators. I have also had concerns in the past about developers getting too hung up on sticking with a one size fits all problems view and failing to apply principles to problems; as that requires more thought and work. A third-party element to this discourages more critical thinking, as there is less room to change how it works.*

# Walkthrough

I will try to walk through an example with explanations as to how we can turn a non-performant group of logic into something a lot more performant and scalable. However, to some extent it cannot get away from theory, as it is hard to cover everything that needs considering in a simple walkthrough.

I have kept this quite simple so that the focus is on the key database aspects and not superfluous logic to this discussion. Therefore, this sample is heavily contrived in terms of fields and what it is really doing. Whilst, at the same time it is based on logic I have already come across that can be found all over our code base. What matters is the structure and patterns, not specific field names or reasoning as to what I am pretending to do.

I have also assumed that the starting point is an Apex based solution, it may be that you will need to convert old Process Builders (PBs) or Flows to Apex first, but that should be sufficiently trivial to do and require no explanation. If you were to simply convert existing PBs and Flows etc you may well end up with something looking a little like the following initial sample.

There are some coding conventions I have not worried about, largely to avoid some awful line wrapping in a word document if we follow normal naming conventions. I have not actually checked the following samples throughout this document compile etc, there may be some typos.

## Starting point.

The starting code that is a performance and scalability issue is the following:

trigger Contact on Contact (after update)

{

    ContactHandler handler = new ContactHandler() ;

    handler.updateAccountFeatureOne(trigger.new, trigger.oldmap) ;

    handler.updateAccountFeatureTwo(trigger.new, trigger.oldmap) ;

}

public class ContactHandler

{

    public void updateAccountFeatureOne(List<Contact> newCts, Map<Id, Contact> oldCts)

    {

        Set<Id> actIds = new Set<Id>();

        for(Contact contact :newCts)

        {

            actIds.add(contact.AccountId);

        }

        List<Account> acts = [SELECT ID, featureField1\_\_c FROM Account WHERE Id IN: actIDs];

        for(Account act: acts)

        {

            act.featureField1\_\_c = calculateFeature1(act, newCts);

        }

        update acts;

    }

    public void updateAccountFeatureTwo(List<Contact> newCts, Map<Id, Contact> oldCts)

    {

        Set<Id> actIds = new Set<Id>();

        for(Contact contact :newCts)

        {

            actIds.add(contact.AccountId);

        }

        List<Account> acts = [SELECT ID, featureField2\_\_c FROM Account WHERE Id IN: actIDs];

        List<Account> accountsToUpdate = new List<Account>();

        for(Account act: acts)

        {

            Decimal newValue = calculateFeature2(act, newCts);

            if(newValue != act.featureField2\_\_c)

            {

                act.featureField2\_\_c = newValue;

                accountsToUpdate.add(act);

            }

        }

        if(accountsToUpdate.size() > 0)

            update accountsToUpdate;

    }

}

The problem with this code comes about from the way it is written as a series of distinct independent steps, with nothing in the way of collating either SOQL or DML operations to reduce the amount of either. Both 2 calls perform a query and both performs some DML on the same object type. The first feature also did the DML with no checks as to necessity, i.e., whether the value of the field was being changed. The second feature at least did check whether the value was going to alter the account, and only performed the DML if that was true.

If we were to grab a log of what is happening here and graph it as flame chart it would look something like:

A picture containing chart

Description automatically generated

This is somewhat stylised and ignores aspects not relevant to the discussion. The red blocks are the actual DML – the bottom contact update is whatever caused our Contact trigger to fire. Our contact trigger executes logic for both ‘features,’ which in turn each do a query (the orange blocks) and fire an account update. As DML tends to be expensive, doubly so on most standard objects, which have lots of other logic built into them, one can assume the Account Update will take a long time.

Whilst the log would be different in the detail, this is in essence what the graph would look like if the two features had been implemented as Process builders or Flows in isolation to each other. The length of the bars would just be longer as those technologies are slower than Apex.

As a side impact, If the account trigger also performs 10 queries, then we have caused 20 queries, 10 more than needed, due to triggering that logic twice. If the account update takes roughly 2 seconds, then we may have 4 seconds of account updates going on. It should also be borne in mind that the time taken, or extra queries performed during an account update will worsen over time as more logic is added into the account trigger, so the performance of our contact logic is extremely dependent on something else. Hence, we should do everything possible to avoid updating the same records more than once, if we must update at all.

## Goals

For the sample above our goals are to change the above code so that following happen:

1. Only perform updates if needed.
2. Only ever perform one update operation.
3. Perform only one query.

## Refactors

I am sure I will get comments on why some of this code or further discussion is different to what already exists. I have stressed a few times to several developers that what I have done so far will almost certainly change, some aspects will be moved from one place to another, or some classes will have further features added to etc. What already exists is not the destination, merely steps on the way to what we will eventually have; whatever that may look like.

An important principle in agile programming is YAGNI, “you ain’t gonna need it.” Put simply do not code what you foresee needing, only do stuff you need now. You may find you never needed what you expected, or that when you do need it you find there was some oddity to it that required something a bit different than originally expected. It gets a little complex, as if you foresee needing something then you want to be at least able to add it without too much work later, so it can be very much judgement calls as to what to do now vs later. There are also some principles that are so basic that there is never a reason to not do. However, I tend to move forward in steps, with an idea of where I want to go and full knowledge that I will change/enhance at later when I need to.

### Step 1

Move our code into our service pattern. The basic patterns have been discussed plenty of times before, and there are enough examples to go and look at, so I am only focusing on the relevant lines here and structuring it so that the code sample can focus on the important parts.

public class ContactService

{

    Public interface ISelector

    {

        List<Account> getAccountsForFeature1(List<Contact> cts);

        List<Account> getAccountsForFeature2(List<Contact> cts);

    }

    private ISelector selector;

    public ContactService(ISelector selector)

    {

        this.selector = selector;

    }

    /\*

    \*   See IDomainService

    \*/

    public override void processRelatedRecordsOnChange(List<SObject> newRecords, Map<Id, SObject> oldRecords)

    {

        List<Contact> newCts = (List<Contact>)newRecords;

        Map<Id, Contact> oldCts = (Map<Id, Contact>)oldRecords;

        updateAccountFeatureOne(newCts, oldCts);

        updateAccountFeatureTwo(newCts, oldCts);

    }

    private void updateAccountFeatureOne(List<Contact> newCts, Map<Id, Contact> oldCts)

    {

        List<Account> acts = selector.getAccountsForFeature1(newCts);

        for(Account act: acts)

        {

            act.featureField1\_\_c = calculateFeature1(act, newCts);

        }

        IUnitOfWork uow = UnitOfWork.createInstance(new SecurityCheck.AllowAllSecurity());

        uow.updateRecords(acts);

    }

    private void updateAccountFeatureTwo(List<Contact> newCts, Map<Id, Contact> oldCts)

    {

        List<Account> acts = selector.getAccountsForFeature2(newCts);

        List<Account> accountsToUpdate = new List<Account>();

        for(Account act: acts)

        {

            Decimal newValue = calculateFeature2(act, newCts);

            if(newValue != act.featureField2\_\_c)

            {

                act.featureField2\_\_c = newValue;

                accountsToUpdate.add(act);

            }

        }

        if(accountsToUpdate.size() > 0)

        {

            IUnitOfWork uow = UnitOfWork.createInstance(new SecurityCheck.AllowAllSecurity());

            uow.updateRecords(accountsToUpdate);

        }

    }

    public class DefaultSelector implements ISelector

    {

        public List<Account> getAccountsForFeature1(List<Contact> cts)

        {

            Set<Id> actIds = new Set<Id>();

            for(Contact contact :newCts)

            {

                actIds.add(contact.AccountId);

            }

            return [SELECT ID, featureField1\_\_c FROM Account WHERE Id IN: actIDs];

        }

        public List<Account> getAccountsForFeature2(List<Contact> cts)

        {

            Set<Id> actIds = new Set<Id>();

            for(Contact contact :newCts)

            {

                actIds.add(contact.AccountId);

            }

            return [SELECT ID, featureField2\_\_c FROM Account WHERE Id IN: actIDs];

        }

    }

}

It may look like there is more code here, there is. We have an interface to represent the querying and DML logic, which adds more lines but is an important aspect of keeping responsibilities clearly separated and becomes useful to help us optimise later.

### Step 2 – A single Query

Let us sort out the single query, so that we only ever perform a single query per execution of this set of logic.

Both queries are returning the Accounts, both request different fields, although the first feature had no code read it (yes, our code really does have lots of that). However, we are going to want both fields eventually so that we can check an update is required.

We therefore want the following:

1. Each feature to still call the selector to get the accounts it needs, as we do not really want to have to change the business logic flow if we can help it.
2. The selector will only have one method. There will be no real need to have distinct methods in this case, all we are doing for each is to return the accounts by contact.accountid. This may not be true in other scenarios, but where the purpose of the selector method is just to return a list of related records from a similar input then there is no need to have different methods.
3. No matter which feature is or is not called, or called first, the selector should work. We do not want the current specific code flow and whether the methods are called in a specific order or whether they are even both called to become an issue later.

There are many possibilities as to how this can be solved, there is seldom one solution that fits all problems. The better mechanism is likely based on what is going on from the service flow point of view and may well change over time.

#### Simple cache

The simplest initial solution for this specific scenario is to just use a simple cache of the answer within the selector.

    public class DefaultSelector implements ISelector

    {

        private Map<Id, Account> accountCache = new Map<Id, Account>();

        public List<Account> getAccounts(List<Contact> cts)

        {

            Set<Id> actIds = getAccountIds(cts);

            // get the currently cached Accounts for the account Ids

            Map<Id, Account> accountsForContacts = getAccountsFromCache(actIds);

            // remove the cached Accounts from the set of account Ids,

            // so we know what to load if anything.

            actIds.removeAll(accountsForContacts.keySet());

            // if now nothing to load then just return the cached accounts.

            if(actIds.isEmpty())

                return accountsForContacts;

            // ... otherwise load the remaining accounts,

            // which puts them in the cache for next call.

            // Then we add the new accounts to those we return.

            List<Account> loadedAccounts = loadAccounts(actIds);

            // put the newly loaded accounts in the collection we going to return.

            accountsForContacts.putAll(loadedAccounts);

            return accountsForContacts;

        }

        private Set<Id> getAccountIds(List<Contact> cts)

        {

            Set<Id> actIds = new Set<Id>();

            for(Contact contact :newCts)

            {

                actIds.add(contact.AccountId);

            }

            return actIds;

        }

        private Map<Id, Account> getAccountsFromCache(Set<Id> accountIds)

        {

            Map<Id, Account> accounts = accountCache.clone();

            accounts.keySet().retainAll(accountIds);

            return accounts;

        }

        private List<Account> loadAccounts(Set<Id> actIds)

        {

            List<Account> loadedAccounts = [SELECT ID, featureField1\_\_c, featureField2\_\_c

                                            FROM Account

                                            WHERE Id IN: actIDs];

            accountCache.putAll(loadedAccounts);

            return loadedAccounts;

        }

    }

The above code has used a member variable to cache the answer, which it uses every call to find any accounts already loaded, and if that means it does not need to do the SOQL then it does not. If a subsequent call to load accounts passes in different contacts, it will find any it can in the cache but still query for the rest. That will not save a SOQL query, but it may save on how many rows are returned across a transaction; which is another governor limit that can arise.

The above code is functionally safe, and the implementation has honoured the contract of the interface, in that it will manage calls for any lists of contacts and provide the correct response. That makes it useful to avoid future bugs even though it does a little more processing to ensure that it works. If we have achieved what we wanted, and saved a SOQL query then that will make up for the extra processing.

There is a downside with the above approach – if the calls to the selector pass in different Contacts, then there will likely still be 2 SOQL queries in addition to the extra processing we have added. The sample code we are dealing with does not do that, so for this scenario it is not a problem. If we want to monitor this type of issue, we can put in unit tests to flag up changes that result in more than the expected number of queries, and then adjust the logic when we see what caused that.

If the two service methods are going to pass in different sets of contacts this whole approach may be the wrong one given our primary goal is reducing the SOQL count; we should probably consider avoiding a simple cache and look at a pre-load option, as described later.

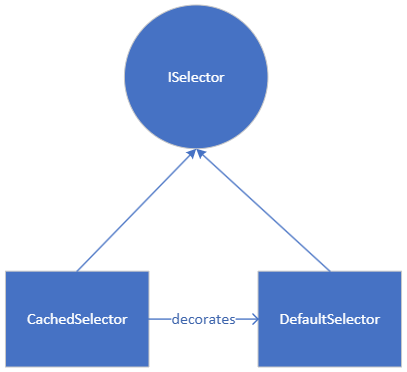
There is a second, possible, downside to this approach. The current service/selector pattern requires the caller to pass in the selector implementation, and we have only changed the DefaultSelector to do this caching. This could mean a service may be given a selector implementation that does not do the caching. This issue does not impact the functionality of the service, only how optimal it is. Overall, this potential issue is not one to worry about yet, we do not currently do this type of logic, and if we do start doing something like this it would be easy to fix by just adding some similar caching into the other new selector or change to using the Decorator approach below.

#### Cache Decorator

In software engineering terms the Decorator patterns adds functionality by wrapping an implementation of an interface in another implementation of the same interface. Whilst described in many places, Design Patterns by the Gang of Four is the seminal book to include it.

The decorating implementation adds a feature that the original version did not provide whilst delegating to the original for the core functionality. This allows for a flexible build-up of features whilst not having to change already written and tested code, and it allows for some easy flexibility around what features are or are not ran at any given time.

In this scenario the DefaultSelector would just perform the simple getAccounts SOQL query with no caching, as per the first refactor in step 1. The CachedSelector would replace the SQOL it was doing (in the sample above) by calling the selector it was wrapping.



public ContactService(ISelector selector)

    {

        this.selector = new CachedSelector(selector);

    }

public class CachedSelector implements ISelector

    {

        private Map<Id, Account> accountCache = new Map<Id, Account>();

        private ISelector baseSelector;

        CachedSelector(ISelector baseSelector)

        {

            this.baseSelector = baseSelector;

        }

[// other stuff like prior version]

        private List<Account> loadAccounts(List<Contact> cts)

        {

            List<Account> loadedAccounts = baseSelector.getAccounts(cts);

            accountCache.putAll(loadedAccounts);

            return loadedAccounts;

        }

There would have to be some other minor changes in the earlier caching code, but the essence is the same – the CachedSelector deals with the caching logic, and the DefaultSelector handles the actual selection (SOQL). The DefaultSelector therefore remains simple and focused on simply running whatever query it needs to, whilst the CachedSelector focuses on the complexity of caching.

This may seem overkill, however, what it provides is a way for the service to be provided any unknown selector and still handle the caching, rather than rely on the selector it had been provided to cache. Either area can be changed without impacting the other so the risk to overall service is reduced in future.

A decorator like this also opens some other useful possibilities, for example unit testing the primary selector is only called once, which is exactly what we are wanting to achieve. As the primary selector is only called if the cached selector needs to call it then it is easy to pass in a mock and track how often it is called due to cache misses, which can be due to some future change that would otherwise not be known about. This is a lot harder with the simple caching version above.

The decorator is also a useful approach to implementing preloading, see next.

#### Preloading

The other common way to manage query minimisation is for any calls into a service to first work out what needs to be loaded and then load up all that data before performing the primary logic. This still requires a cache of some description – for the preloaded data to be stored in. Whilst it is not the scenario for our example code, this is a reasonable solution where there will be multiple calls to a selector with different inputs, as that makes the simple cache above largely redundant in terms of what we need to achieve.

    public ContactService(ISelector selector)

    {

        this.selector = selector;

    }

    public override void processRelatedRecordsOnChange(List<SObject> newRecords, Map<Id, SObject> oldRecords)

    {

        List<Contact> newCts = (List<Contact>)newRecords;

        Map<Id, Contact> oldCts = (Map<Id, Contact>)oldRecords;

        preload(newCts, oldCts);

        updateAccountFeatureOne(newCts, oldCts);

        updateAccountFeatureTwo(newCts, oldCts);

    }

    private void preload(List<Contact> newCts, Map<Id, Contact> oldCts)

    {

        selector.getAccounts(newCts);

    }

In the above example we just loaded the accounts for all contacts in the preload method and do not try and work out which contacts are relevant. This is fine for our example code, as we know our logic was going to load all of them anyway. Even if you have logic that only loads accounts for a subset of contacts it may still be worth just loading all of them, we would be defending against some future logic bugs. If you have a complex preload that is not updated when a new feature is introduced, then the newer feature may not work properly as they had not pre-loaded the accounts. Worse, the preload may be updated for a new feature but in a way that breaks existing features. Just loading all contacts simply bypasses that risk.

There is a bit of a flaw here, in that it is assumed in the above sample that the selector caches when getAccounts is called. As the service works with an interface, that really should not be taken for granted and in fact the service may have just triggered an extra SOQL call rather than saved any. If the selector does not cache, then you would have to take the response and cache it directly in the service. The decorator version discussed above can be a particularly useful way of handling the caching in this scenario, as you know the caching happens no matter what selector was given to you in the constructor. Another approach still is to add a preload method to the selector interface, with what you think are the inputs needed for a preload. This allows you to make it explicit that the selector will have a preload called, and it can then choose to query and cache stuff without the service worrying about it. The selector may or may not do s preload, that would be up to it.

The downside of a pre-load is that it may cause memory issues, which is another key governor. As the data is held in a cache from the start of the service call for later use then it cannot readily be dropped when no longer needed, which would otherwise keep the memory usage lower. This is the sort of issue that needs dealing with on a case-by-case basis. Most of our services will not load up enough data to worry about – e.g.; if loading Accounts with only two fields is an issue, then it is likely that you have an even bigger issue elsewhere. For most services, this memory aspect is only likely to be problematic where large numbers of fields on large numbers of records are loaded per service call, or substantial number of heavy fields like long text.

#### Static Caches

It is common in Salesforce to see developers have some form of cache defined as a static variable at some point in the code structure.

The current design I have been pushing for in our triggers and services is that each service is short lived and is disposed of once the trigger call is handled. So long as the selector and any caches it contains are not static then they are also disposed of. This is a good design from the point of view of ensuring that memory is released and keeping things simple, however, it does have some downsides.

In a bulk scenario a trigger will fire many times, e.g., If an Account will update Contacts and there are hundreds of contacts. In such a scenario we would create a service and selector many times, and as each cache only lives for the duration of the 1 invocation there will be no caching between trigger invocations. This can be dealt with, if it is a problem, by making the cache static at some level.

For example, if 600 contacts are updated via whatever functionality, then our trigger fires 3 times. If we have followed one of the above caching patterns then each of those trigger invocations will now query for accounts only once, for a total of 3 queries over the 3 invocations (whereas it was 6 before our caching). If we put in a static cache, then we *might* reduce that to 1 query. The emphasis is on *might*, even with a static cache we cannot be sure that in each trigger invocation we will not have to load accounts we have not seen yet.

The major issue with static caches, from a functional point of view, is that we may cache the Account on the first trigger invocation and then some other logic might update the account before the second invocation. This may be due to another feature in our own service updating accounts after the query, or it may be something independent that this service knows nothing about, i.e. another trigger or Flow etc. We now have a cached version of the account that is ‘stale,’ i.e.; it no longer reflects the latest version of the account and later logic will be based on checking incorrect values. There are solutions to this, but again they significantly increase the complexity of the code and add yet further processing time to handle. If there is a high enough chance of having stale data in a cache then you will be engaging in that extra processing without saving on the SOQL queries, which was the goal here, so all you will achieve is making the system less performant and scalable.

Where static caches, and the processing for staleness may be useful is if the primary goal is reducing the number of records queried. As there is a governor on how many records are returned from the database you want to try and avoid querying for the same record repeatedly. For example, there was a bug months ago where the 50,000 query row limit was exceeded as we were loading All Users every trigger invocation of some record type. As we have thousands of users that quickly became a query row exceeded bug. Putting the response for that query into a static cache was a useful approach, not least as the data we were dealing with was not the sort of thing that is going to change mid execution which made Staleness a non-issue.

The mechanics of Salesforce make staleness a problem to start with, but in a business, like ours, that hands out poorly designed solutions to a range of people to simply ‘configure’ with no real thought as to wider technical system or data design then staleness is a major issue.

**I would strongly recommend against static caching unless you know you really need it**. They introduce extra complexities around staleness, and hence data corruption, as well as extra memory usage that can result in the opposite of improving performance and scalability. Focus on caches defined as standard member variables that are narrow in scope and safe from staleness concerns, and only contemplate static caches if there is real identified problem that they can solve.

#### Code Structure

The way we structure code can also have an impact on how to cache – the example code I am using is very simple, but it is worth bearing in mind that when you add to or refactor your code you may find that you need to look at introducing some caching that is awkward to achieve given the current code structure. Be prepared to change the structure if needed, whether it is by splitting out classes or interfaces, merging classes or changing a selector that now has a not so useful interface etc.

All code is subject to change as requirements change over time. What works today may not work tomorrow. This is especially true when it comes to introducing caching, as it is most likely that you are going from a scenario where you were not caching as there was no need to a scenario where there is now something else that wants the same information. You are doing something that impacts some older logic at the same time as adding something new, therefore you may well have to change how that old logic works or introduce a new structure for the old logic that makes it easier to share data.

The basic principles still apply though, keep the layering, and keep logic with different responsibilities separate, think about what makes sense as a single piece of logic – e.g., avoid a call to do X also do Y as mixing logical units of work in a single function can be a swine to deal with later. Keep interfaces that can be mocked out for unit testing at the layer boundaries etc.

#### Step 2 – Summary

There are different ways of managing the issue of minimising queries, but in one way or another they all rely on some form of caching. No one solution is perfect, each has advantages or disadvantages, and the best solution will depend on the specifics of the problem. The mechanisms discussed above should all be good enough, alone or in combination, for the logic Citation needs.

Caching is itself a complex topic at times, and you need to be careful of stale caches and how Salesforce makes that a likely scenario in trigger-based logic, even more so where bad business design results in a chain reaction of unexpected side effects. As far as possible try to keep any cache as narrowly scoped to the service call that needs it. Whilst adding ‘static’ to a cache looks simple and a quick win to even more optimisation it is often just a way to introduce several hard to debug/fix data corruption issues etc or to further degrade performance as you try to solve that.

There are scenarios that the above discussions will not be so good at handling in the most optimal way, e.g.; where multiple services need to call each other, but do not worry about such for now. Those scenarios are likely to be uncommon (we do not really do a service-based API for example) and the complexity of handling them may not be worth the benefit, at least until you have real life examples and can analyse what problems come up. Most of our querying issues are solved by the basic principles above.

### Step 3 – A single Update

Our next problem area to deal with is changing from performing two Account updates and only doing a single update, if needed.

#### Unit of Work

In software engineering terms the Unit of Work (UOW) is a pattern for controlling and optimising access to the database, primarily writing to the database. The most well-known description is in Martin Fowler’s Patterns of Enterprise Application Architecture.

There is currently a unit of work interface/class in our system. It was added recently, and in line with most of what I do it is currently very simple – the initial code was only really to allow mocking DML but was called ‘unit of work’ as that is what it will probably become. The concept of a unit of work will be a key aspect for this discussion, though the precise implementation may be up for further discussion.

The basic concept of using a UOW is that rather than perform DML at the point where some specific feature has finished updating a record instead it ‘registers’ the record for an update. The actual update will occur later, when the service has collected all changes from all features being executed. In essence our two features will now do the in-memory updates of accounts and signal which ones need to be persisted and leave the timing of persistence to something else. This provides the calling code much more control over how to interact with the database.

#### Only update when needed

As we mentioned earlier, we should only ever update a record that changes. This means that feature1 is currently bad, as it always updates accounts even if the field it updates will not have changed value.

This is a key concept. If we wish to have performance and minimise governor issues, then we should not engage in expensive operations like DML if it was not needed. A trigger may fire for a variety of reasons, and no feature should assume that it needs to execute or update a related record. In an ideal scenario there will be no updating of related records as no related record will change as a result of the edit and the trigger only needed to do some basic validation or setting some defaults in the before triggers etc. If even one feature fails to follow this rule, then a lot of the work on the other features is wasted.

Furthermore, this is a key concept to preventing recursive scenarios. Most recursive scenarios happen where developers never bothered to check whether an update was needed. If A updates B and B updates A and neither checks for the necessity, then there is an infinite loop and Salesforce will kill your process with a limit exceeded error. However, if both check for the necessity then it will almost never be the case that there will be a limit exceeded error – whatever value A updated B with will not need to occur when it is called a second time (after B updates A) as now B already has that value. The only likely way the recursion can occur at this point is if the business itself has defined a fundamentally flawed business rule.

Between the unit of work and fixing the feature1, to check for an update, the code would possibly look something like the following:

    public override void processRelatedRecordsOnChange(List<SObject> newRecords,

                                                        Map<Id, SObject> oldRecords)

    {

        List<Contact> newCts = (List<Contact>)newRecords;

        Map<Id, Contact> oldCts = (Map<Id, Contact>)oldRecords;

        IUnitOfWork uow = UnitOfWork.createInstance(new SecurityCheck.AllowAllSecurity());

        updateAccountFeatureOne(newCts, oldCts, uow);

        updateAccountFeatureTwo(newCts, oldCts, uow);

        uow.commitWork();

    }

     private void updateAccountFeatureOne(List<Contact> newCts,

                                            Map<Id, Contact> oldCts,

                                            IUnitOfWork uow)

    {

        List<Account> acts = selector.getAccounts(newCts);

for(Account act: acts)

        {

            Decimal newValue = calculateFeature1(act, newCts);

            if(newValue != act.featureField1\_\_c)

            {

act.featureField1\_\_c = newValue;

                uow.registerDirty(act);

            }

        }

    }

    private void updateAccountFeatureTwo(List<Contact> newCts,

                                            Map<Id, Contact> oldCts,

                                            IUnitOfWork uow)

    {

        List<Account> acts = selector.getAccounts(newCts);

for(Account act: acts)

        {

            Decimal newValue = calculateFeature2(act, newCts);

            if(newValue != act.featureField2\_\_c)

            {

act.featureField2\_\_c = newValue;

                uow.registerDirty(act);

            }

        }

    }

class UnitOfWork

{

    private Map<SObjectType, Map<Id, SObject>> recordsToUpdate

        = new Map<SObjectType, Map<Id, SObject>>();

    void registerDirty(SObject sobj)

    {

        Map<Id, SObject> sobjMap = recordsToUpdate.get(sobj.getSObjectType());

        if(sobjMap == null)

        {

            sobjMap = new Map<Id, SObject>();

            recordsToUpdate.put(sobj.getSObjectType(), sobjMap);

        }

        sobjMap.put(sobj.Id, sobj);

    }

    void commitWork()

    {

        for (Map<Id, SObject> objs: recordsToUpdate)

        {

            update objs.values();

        }

    }

}

We create a UOW instance before executing any of our features, and we pass that instance through to each method to use. Those methods no longer grab their own UOW, but instead register the updated Accounts with the UOW given to them. After the service has executed all features, it will call commitWork and the UOW will then perform a single DML statement (per object type).

The above unit of work is only part of what is likely to be a fully-fledged UOW class, but it demonstrates the basics. As objects are registered, we put them in a Map, keyed by SObjectType – this is for later so we do not perform mixed DML which can be problematic. Then, when commitWork is called we update the records in that collection.

A more complete UOW will have methods for registerNew and registerDelete, for which the commitWork will insert or delete, respectively. We would also likely have bulk register methods that take lists. It may allow some control over what order records, or types of record are processed first, or which order different types of DML are performed etc. Again, those are all features that can be added once they are needed, and the requirements can be better determined.

The above code has resulted in us having only a single Account update per trigger call. Assuming we have the cached selector we also have only 1 query. Our Flame chart now looks more like:

Graphical user interface, application

Description automatically generated

### Step 4 - Selectors and Unit of Work combined

There is, however, a final issue we need to deal with, or maybe more correctly, be aware of. In the simple sample we have this is not actually a problem. However, the sample works to a certain extent by accident over design.

In the section on Selectors I discussed stale data, as in cached data no longer reflecting the data on the database, as something else altered it mid-flow. In our sample this specific scenario cannot happen, as an update of the Accounts does not happen in between the 2 selector calls. However, the cached data is not of necessity a reflection of what is in the Unit of Work, and therefore there is still a ‘staleness’ issue to consider.

The sample code does not suffer this, so let us modify the code slightly to demonstrate this issue. Let us assume that a few months later there is an ‘enhancement’ to feature2. The enhancement adds an else clause that checks field ‘featureField1\_\_c’ being a certain value and updates something else if that is true.

    private void updateAccountFeatureTwo(List<Contact> newCts,

                                            Map<Id, Contact> oldCts,

                                            IUnitOfWork uow)

    {

        List<Account> acts = selector.getAccounts(newCts);

        List<Account> accountsToUpdate = new List<Account>();

for(Account act: acts)

        {

            Decimal newValue = calculateFeature2(act, newCts);

            if(newValue != act.featureField2\_\_c)

            {

act.featureField2\_\_c = newValue;

                uow.registerDirty(act);

            }

            else if(act.featureField1\_\_c == 'QMS')

            {

                act.someOtherField\_\_c = 'somevalue';

                uow.registerDirty(act);

            }

        }

    }

This is clearly contrived, but ignore why we might have this code here etc, the principle is important. Feature2 now has logic based on a record field the selector returned, but the field it is referencing is a field another feature may have updated in the same service call.

So, we have feature1 execute and potentially update the featureField1\_\_c value, let us say from ‘FRA’ to ‘QMS’ and then feature2 runs and, based on what the selector returns as the featureField1\_\_c value will execute some logic. But what value does the record from the selector show – is it reading ‘FRA’ or ‘QMS’? It obtained the record from the selector, which read from the database at the start of the call, so it could be ‘FRA’ or has something occurred, by accident or design, to give us ‘QMS’ as that is now the ‘current’ value (in the unit of work at least).

In our example it is probably ‘QMS,’ the new value that feature1 set. We would have to check the caching code to be sure, but if the example caching I have shown above was followed more or less verbatim then it will be. Whether we want to read the old or new value might need some business clarification on desired outcome, but I expect most scenarios will want the most up to date value.

The reason we expect the field to have the latest value here is probably accidental, in that we have not gone out of our way to specify the guaranteed behaviour. What has happened is that we have cached the initial query response, but the records are handed out by reference from the selector/cache and put into the unit of work by reference. At all times we are working with the same instance of Account in memory. It would, however, be extremely easy for that to ‘break.’ For example, the cache may clone the records to return, or some feature logic itself might instantiate another instance of account to put in the unit of work rather than the one the selector returns.

A classic example of this can occur where a feature is implemented that never read an account, but instead instantiates an account just based on the contacts accountId and pushes that into the unit of work. Then another feature reads accounts from a selector (which does not yet know about the unit of work version) and updates the account based on those selector values, which are effectively stale values; then to make matters worse it pushes its version of the account into the unit of work – overwriting the existing update. We now have two potential bugs; the first update never occurs and the second one was based on stale data. Once we move to these sorts of enterprise patterns such considerations as this become important, and any access to a related record should go through the well-defined selector route, and not bypassed.

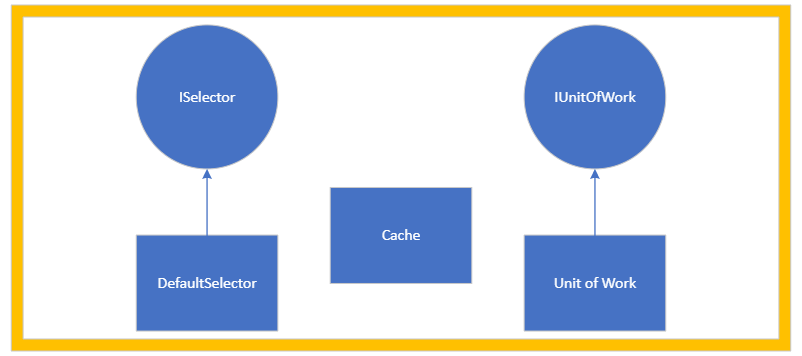
This may sound complicated and hard to fix. It is worth noting that FFLib does not really try and address this issue either, as it is hard to deal with generically on Salesforce. Fowler does cover some this when discussing the unit of work pattern, and another of Fowler’s patterns that often pairs with the UOW, called the Identity Map, can help but be aware that the Identity Map described by Fowler is intended for something a bit more complex still. Be wary of the Identity Map on Salesforce unless you really understand what you are doing, apart from being a tricky subject it also must be very record specific, which can run contrary to Salesforces bulk processing paradigm. I have seen an attempt to implement the Unit of Work and identity map cause some major performance issues. As with all patterns, Martin Fowler’s are good, if you understand what they are solving and do not try and apply them in the wrong place.

For the most part, if we have well designed business rules and data modelling then this should be an uncommon scenario, that is likely best dealt with case by case. It is harder to say how uncommon this will be given we do not have well designed business rules and very poor object schema. If it turns out that we do need it everywhere then we may need a more generic version – another case of YAGNI.

#### Database Façade.

In software engineering terms the Façade is a thin layer above a set of smaller components that form some (sub)system. The façade tries to reduce the complexity to a simpler, single interface that a caller needs rather than some caller having to handle the interactions and complexity of the smaller parts. There is another pattern, known as the Repository pattern that is really just a façade but specialised to persistence/storage. A Repository is a Façade, but a façade is not of necessity a Repository. What I am describing here could be viewed as a Repository, but I prefer the simpler Façade as a term.

If we look at our database components, we have something like:



The orange box represents the idea that we have a subsystem of database components, and the blue shapes within are the components that make up that subsystem. As it stands, our service interacts with all the components directly. But as this becomes more complex, we get to a point where the business logic code will become far too bloated and complicated by a lot of interactions with these sub components. Rather than have service level code deal with this we can put a Façade over the subsystem, that provides the methods our service needs, and that in turn understands how to handle any interactions between the unit of work, selector and/or caches.

Just as a standard interface defines some abstract behaviour that may be implemented in many ways that we do not care about; a Façade represents an interface to a complex subsystem that we really do not want to have to know how to implement.

A picture containing text, electronics

Description automatically generated

Our service feature code will now always interact with the Façade (ContactServiceRepo) rather than with either the selector or the unit of work. The Façade will be responsible for calling the selector, any caching, and calling the unit of work and keeping them all coordinated. This offers no great benefit in and of itself. However, what we can now do is have the updateAccount method on the Façade both register the account as dirty on the unit of work and, if needed, poke the cache so that both are coordinated. The façade is also in control of calling the selector, and depending on where you wish to put a cache, it may even manage the caching. We have a single place to handle any needed interactions between the subcomponents so we do not end up with that scattered around the service level code. It also encapsulates such logic, so we can change how it works without impacting the service level code.

I am not going to go too much into the details here. Hopefully, the concept is easy enough that the specifics for any given service are easy enough to handle.

## Final Sample Code

The following code is what we may have by the end. For sake of argument, I have assumed that we will use some form of Façade as above to handle the selector and UOW. The Caching is done via a Decorator to the base selector. Also for sake of this document I have not worried about whether this impacts how the service is constructed and just continued to pass in the Selector as now, Dependency Injection etc is another discussion and not overly related to this.

It is worth pointing out that this will look like a lot of code compared to the original. This is true. The original code was only interested in doing the simple logic for each feature and was not interested in controlling governor limits or doing so in a way that can handle certain issues that arise once you start to try and handle scalability problems on Salesforce.

Some of the code here is generic, and not specifically related to this sample, e.g. the Unit of work class is a reusable class. There may be room, once more scenarios are understood, to handle the caching or Facades in a more generic and reusable way as well.

public class ContactService

{

    public interface ISelector

    {

        List<Account> getAccounts(List<Contact> cts);

    }

    private ContactServiceRepo repo;

    public ContactService(ISelector selector)

    {

        this.repo = new ContactServiceRepo(selector);

    }

    public override void processRelatedRecordsOnChange(List<SObject> newRecords,

                                                        Map<Id, SObject> oldRecords)

    {

        List<Contact> newCts = (List<Contact>)newRecords;

        Map<Id, Contact> oldCts = (Map<Id, Contact>)oldRecords;

        updateAccountFeatureOne(newCts, oldCts);

        updateAccountFeatureTwo(newCts, oldCts);

        repo.commitWork(newCts, oldCts);

    }

    private void updateAccountFeatureOne(List<Contact> newCts,

                                            Map<Id, Contact> oldCts,

                                            IUnitOfWork uow)

    {

        List<Account> acts = repo.getAccounts(newCts);

        for(Account act: acts)

        {

            Decimal newValue = calculateFeature1(act, newCts);

            if(newValue != act.featureField1\_\_c)

            {

                act.featureField1\_\_c = newValue;

                repo.registerDirty(act);

            }

        }

    }

    private void updateAccountFeatureTwo(List<Contact> newCts,

                                            Map<Id, Contact> oldCts,

                                            IUnitOfWork uow)

    {

        List<Account> acts = repo.getAccounts(newCts);

        List<Account> accountsToUpdate = new List<Account>();

        for(Account act: acts)

        {

            Decimal newValue = calculateFeature2(act, newCts);

            if(newValue != act.featureField2\_\_c)

            {

                act.featureField2\_\_c = newValue;

                repo.registerDirty(act);

            }

        }

    }

}

public class UnitOfWork implements IUnitOfWork

{

    private Map<SObjectType, Map<Id, SObject>> recordsToUpdate

        = new Map<SObjectType, Map<Id, SObject>>();

    public void registerDirty(SObject sobj)

    {

        Map<Id, SObject> sobjMap = recordsToUpdate.get(sobj.getSObjectType());

        if(sobjMap == null)

        {

            sobjMap = new Map<Id, SObject>();

            recordsToUpdate.put(sobj.getSObjectType(), sobjMap);

        }

        sobjMap.put(sobj.Id, sobj);

    }

    public void commitWork()

    {

        for(Map<Id, SObject> objs: recordsToUpdate.values())

        {

            update objs.values();

        }

    }

}

class ContactServiceRepo

{

    private CacheSelector selector;

    private IUnitOfWork uow;

    ContactServiceRepo(ISelector selector)

    {

        this.selector = new CacheSelector(selector);

        this.uow = UnitOfWork.createInstance(new SecurityCheck.AllowAllSecurity());

    }

    List<Account> getAccounts(List<Contact> cts)

    {

        return selector.getAccounts(cts);

    }

    void registerDirty(Account act)

    {

        // register account for update

        // and update the selector with the latest version.

        uow.registerDirty(act);

        selector.push(act.Id, act);

    }

    void commitWork()

    {

        // commit the work

        // and clear the selector, as data is now stale.

        uow.commitWork();

        selector.clear();

    }

}

public class CachedSelector implements ISelector

{

    private Map<Id, Account> accountCache = new Map<Id, Account>();

    private ISelector baseSelector;

    CachedSelector(ISelector baseSelector)

    {

        this.baseSelector = baseSelector;

    }

    public void push(Account act)

    {

        accountCache.put(act.Id, act);

    }

    public void clear()

    {

        accountCache.clear();

    }

    public List<Account> getAccounts(List<Contact> cts)

    {

        Map<Id, Contact> contactsByAccountId = getAccountIds(cts);

        // get the currently cached Accounts for the account Ids

        Map<Id, Account> accountsForContacts = getAccountsFromCache(contactsByAccountId.keySet());

        // remove the cached Accounts from the set of account Ids,

        // so we know what to load if any.

        // If there are no accounts to load now we return.

        contactsByAccountId.keySet().removeAll(accountsForContacts.keySet());

        // if now nothing to load then just return the cached accounts.

        if(contactsByAccountId.isEmpty())

            return accountsForContacts;

        // ... otherwise load the remaining accounts,

        // which puts them in the cache for next call.

        // Then we add the new accounts to those we return.

        List<Account> loadedAccounts = loadAccounts(contactsByAccountId.values());

        // put the newly loaded accounts in the collection we going to return.

        accountsForContacts.putAll(loadedAccounts);

        return accountsForContacts;

    }

    private Map<Id, Contact> getAccountIds(List<Contact> cts)

    {

        Map<Id, Contact> actIds = new Map<Id, Contact>();

        for(Contact contact :newCts)

        {

            actIds.put(contact.AccountId, contact);

        }

        return actIds;

    }

    private Map<Id, Account> getAccountsFromCache(Set<Id> accountIds)

    {

        Map<Id, Account> accounts = accountCache.clone();

        accounts.keySet().retainAll(accountIds);

        return accounts;

    }

    private List<Account> loadAccounts(Set<Id> actIds)

    {

        List<Account> loadedAccounts = baseSelector.getAccounts(cts);

        accountCache.putAll(loadedAccounts);

        return loadedAccounts;

    }

}