When making the transition from standard development to cloud development on a platform like Force.com, there are some critical differences that force you to change how you would approach the project from the architecture stage all the way through the final testing stage. One of those differences is governor limits. Since the Force.com platform is built on a multi-tenant architecture, it relies on governor limits to ensure applications from one tenant don't affect the performance of another tenant.

What is a Multi-Tenant Architecture?

Since the Force.com platform hosts around 77,300 customers (Salesforce.com, n.d.), traditional single-tenant architecture, where each customer has its own set of resources, wouldn't work very well. It wouldn't scale, updates would be difficult and administration would be a nightmare. That's why Salesforce.com built Force.com on a multi-tenant architecture. In a multi-tenant architecture, all the clients, or "tenants" share the same set of physical resources, and one instance of the software. They are separated by virtual environments, so to the end user, it appears as if they are running their own application.

According to Salesforce.com, there are some key advantages to multi-tenancy that are simply not possible in a traditional, single-tenant architecture. The first is instant provisioning. In a single tenant environment, when a new customer is brought on, a new stack must be provisioned and managed in order for them to use the application. In a multi-tenant environment, the same feat can be accomplished in a few clicks of the mouse because the physical environment is already in place. A virtual workspace for the new customer is all that needs to be created. Another advantage to multi-tenancy is scalability. Instead of managing thousands of software installs and servers, you only need to manage a few instances. This makes updates to the software and expansion of infrastructure much easier and quicker (Salesforce.com, n.d.).

These advantages do not come without their issues. Since multiple tenants share the same infrastructure, limits must be put in place to ensure one tenant's application does not cause disruptions or performance issues to another tenant's applications. This is where governor limits come into play.

What are Governor Limits, and Why Do We Need Them?

According to Salesforce.com, governor limits are "runtime limits enforced by the Apex runtime engine to ensure that code does not misbehave" (Albert, n.d.). Governor limits are Salesforce.com's way of making sure our code does not impact other tenants in the infrastructure. They affect almost every area of programming on the Force.com platform. From application logic such as how many script statements you can run, to memory management like heap size limits, to database query and DML limits.

Without governor limits, a multi-tenant architecture such as Force.com's, could never work. Imagine a developer creating a simple class that ended up trying to loop through millions of records at runtime. The impact could be catastrophic to the other users of the system, since they all share the same physical resources.

Work With Governor Limits, Not Against Them

From a developer's view, governor limits can be very painful. There is nothing worse than getting 75% of the way through a project, only to find out through testing that it can't support large numbers of records without hitting governor limits. You have to go all the way back to architecture and try to figure out a way of handling the large quantities without losing any functionality, and in a short time frame.

The best way to avoid this is to start your project with governor limits in mind. Before you start coding, figure out the size of the data set your application will be working with. Also, figure out how much other code will be in the environment. Your trigger may be well within the governor limits when it runs, but if there are four other triggers on the same object, you may hit the limits when they are executed together. Unfortunately, you may not easily be able to calculate the resources that will be needed by managed applications from the AppExchange, so always make sure your development org has all of the applications installed that you intend to install into the production org. Another thing to keep in mind is the number of people who will be using your application, as well as how often they will be using it. Some governor limits are on a timed basis or based off the amount of licenses in your organization, or both. Let's look at a couple of the most troubling governor limits and ways to get around them.

Database Query and DML Governor Limits

On the Force.com platform, you are limited in how often, and how many records you can retrieve using Salesforce.com Object Query Language (SOQL). You are also limited in the number of records you can perform DML operations like insert, update, or delete on in a single transaction. Because using Apex triggers to update large amounts of records in a single transaction is a very common practice, these SOQL and DML limits can be very troublesome to developers. Take for example, the following trigger:

```
trigger UpdateAccountManagerOnChildAccts on Account (after insert, after update) {
        for (Account a : trigger.new){
                    for (Account child : [SELECT Id, Account_Manager__c FROM Account WHERE
                    ParentId = :a.Id]){
                    child.Account_Manager__c = a.Account_Manager__c;
                    update child;
                    }
        }
}
```

This code will compile fine, and it will even give the intended result in some cases, but this code is far from being correct. It could potentially break in numerous ways. For example, a mass update of accounts would cause it to break numerous governor limits, including the 20 SOQL queries per transaction limit, and the 20 DML statements per transaction limit. To avoid these types of governor limits, always make sure to keep DML statements and SOQL queries out of loops. Load the data in to collections first, then issue your queries or DML statements. Here is an example of the previous trigger that has been fixed to avoid these limits:

```
trigger UpdateAccountManagerOnChildAccts on Account (after insert, after update) {
        //Define a map to hold parent accounts, and a list to hold child accounts to update:
        Map<id, Account> parentMap = new Map<id, Account>();
        List<Account> childAccounts = new List<Account>();
        //Add the parent accounts to the map:
        for (Account a : trigger.new){
                        parentMap.put(a.Id, a);
        }

       //Query for child accounts, set the Account Manager appropriately and add them to the
       list to update:
       for (Account child : [SELECT Id, Account_Manager__c, ParentId FROM Account WHERE ParentId
        IN :parentMap.keySet()]){
                child.Account_Manager__c = parentMap.get(child.ParentId).Account_Manager__c;

                childAccounts.add(child);
        }
       //Update the list of child accounts with a single DML statement:
       update childAccounts;

}
```

This trigger avoids looped queries and DML statements by adding the records to collections prior to executing the queries or DML statements. Using this trigger would allow accounts to be updated in batches, and allow accounts that have multiple child accounts to be updated.

Even the corrected trigger above is not completely immune from hitting governor limits. It is possible that an account may have multiple child accounts. If this happens, this trigger could break a governor limit that limits the amount of rows that can be returned by a query and a limit that regulates the amount of records that can be updated via DML in a trigger. There is also a potential for this trigger to call itself recursively. For example, if one of the accounts that where updated had child accounts of its own, this trigger would be called again. If this happens multiple times, governor limits could be hit because all those triggers would count cumulatively against the limits. To fix these issues, we would need to dive deeper than I am here. The fixes would most likely be based on certain criteria for the specific situation, like how many records in the organization could affect this and if there are other triggers running on the account object.

Runtime Execution Governor Limits

Another type of governor limit applies to runtime execution. These limits are in place to help control runaway processes or limit execution time that can be greatly increased by things like Web Service or HTTP callouts. Some of the more common limits of this type include total number of executed script statements per transaction and total number of future calls allowed per transaction.

Let's use the total number of future calls per transaction for an example. Future methods, which are designated using the "@future" annotation above the method declaration, are great because they take a job and move it out of the current transaction into a queue to be executed when resources are available. Because of this, the governor limits for future methods are much looser than on triggers. While they do help, they are not a silver bullet for working around governor limits. Take the following trigger for example:

```
trigger SetDomesticFlag on Lead (after insert, after update) {

        for (Lead l : trigger.new){
                FutureMethods.SetDomesticFlag(l.Id);
        }
}
```

The above trigger calls the following method:

```
public with sharing class FutureMethods {
        @future
        public static void SetDomesticFlag(id leadId){
                //Get the lead record passed in
                Lead l = [SELECT Country, Domestic__c FROM Lead WHERE Id = :leadId];

                //if the lead's address is in the USA then mark as domestic
                if (l.Country == 'USA'){
                        l.Domestic__c = true;
                }else{
                        l.Domestic__c = false;
                }

                //update the record
                update l;
          }
}
```

As with the previous example, there are no compilation errors, and this trigger will work, as long as there are no more than 10 records in the group that cause the trigger to execute. As soon as there are more than 10 leads loaded in a batch, this trigger will throw a governor limit exception for making more than 10 future calls in one transaction.

Similar to the first example, this issue can be resolved by using lists to pass batches of leads to a single future call as in the following example:

```
trigger SetDomesticFlag on Lead (after insert, after update) {
        List<id> leadIdList = new List<id>();

        //add all the Id's to a list before calling the future method
        for (Lead l : trigger.new){
                leadIdList.add(l.Id);
        }

        //Make a single future call to update all leads in the list
        FutureMethods.SetDomesticFlag(leadIdList);
}
```

The above trigger will call this updated future method:

```
public with sharing class FutureMethods {
        @future
        public static void SetDomesticFlag(List<id> leadIds){
                List<Lead> leadsToUpdate = new List<Lead>();

                //Loop over the lead records passed in
                for (Lead l : [SELECT Country, Domestic__c FROM Lead WHERE Id IN :leadIds]){
                        //if the lead's address is in the USA then mark as domestic
                        if (l.Country == 'USA'){
                                l.Domestic__c = true;
                        }else{
                                l.Domestic__c = false;
                        }

                        leadsToUpdate.add(l);
                }
                //update the records with a single DML statement
                update leadsToUpdate;
        }
}
```

By pulling the future method calls out of our loops and passing lists instead of individual records, we can greatly reduce our chances of running into governor limits, much in the same way we can pull queries and DML statements out of loops to reduce their risks for hitting governor limits.

Other Types of Governor Limits

There are other types of governor limits that don't fall into either of the above two categories. These are often the most dangerous as they often don't show up in testing.

Limits like the total amount of future method calls per 24 hours per license can cause a lot of headaches as they can't easily be reproduced in a testing environment, and can surprise you when you get to production. Using the future calls example above will work fine, but if you have one user in your organization and you load up more than 200 batches of leads in one day, you will not be able to make any further future calls after the first 200. So until your 24 hours restarts, your trigger will throw an error every time someone tries to create or update a lead. This is what makes these types of limits so dangerous.

Basic Governor Limit Chart

Governor Limit Type	*Example*	*Ways to Avoid*
Database Limits	Total number of SOQL queries issued	• Use collections like Lists, Sets, or Maps to avoid queries embedded in loops. • Set up the data model to allow for easy access of related records.
Execution Limits	Total number of executed script statements	• Use future methods and batch processing when possible. • Code efficiently and watch out for multiple triggers working on the same object.
Daily Limits	Organization-wide limit of 200 method calls with the future annotation per license per 24 hours	• Know the number of users and the frequency of use for the applications you are developing. • Use batch processing when possible.
Other Limits	Maximum size of WSDL if converted to Apex	• Most of these types of limits are avoided by thorough testing and proper architecture.
For more details on these and all the different types of governor limits go to http://tinyurl.com/25uxouz		

Conclusion

Governor limits can be quite a challenge for developers, especially coming from other platforms where they weren't bound by them. When architecting your project on the Force.com platform, governor limits should always be in your mind. When coding your project, take advantage of the tools available to you. Use collections when possible, and if you know your application will cause heavy processing, use future methods or batch processing. These are used for long running processes on large amounts records, and can even be scheduled to run at specific times, to help work around the governor limits.

There are also limits methods in Apex for checking what various types of limits are, and how close your application is breaking those limits. Some of these methods include: getAggregateQueries, getLimitAggregateQueries, getDMLStatements, getLimitDMLStatements, etc. These methods are available for all of the execution and database limits. Unfortunately, these limit's methods aren't available for other types of governor limits such as daily limits. For those, you still are left to plan and code as efficiently as possible. For details on all of the available limits methods, see http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_methods_system_limits.html.

The longer you work with the Force.com platform, the more familiar you will become with these limits, and the easier it will become to work around them. Going forward, I am hoping that Salesforce.com will continue to increase these limits as well as improve the tools available for working within them. When working in a multi-tenant architecture, governor limits are critical to the stability of the overall system. Likewise, when building a new platform such as Force.com, giving developers the tools and freedom to build creative and powerful applications is also critical. I'm excited to see what the future holds for the Force.com platform as well as other cloud-based development platforms.