1. Apex Triggers
2. Bulk Apex Triggers

Bulk Apex Triggers

Learning Objectives

After completing this unit, you'll be able to:

- Write triggers that operate on collections of sObjects.
- Write triggers that perform efficient SOQL and DML operations.

Bulk Trigger Design Patterns

Apex triggers are optimized to operate in bulk. We recommend using bulk design patterns for processing records in triggers. When you use bulk design patterns, your triggers have better performance, consume less server resources, and are less likely to exceed platform limits.

The benefit of bulkifying your code is that bulkified code can process large numbers of records efficiently and run within governor limits on the Lightning Platform. These governor limits are in place to ensure that runaway code doesn't monopolize resources on the multitenant platform.

The following sections demonstrate the main ways of bulkifying your Apex code in triggers: operating on all records in the trigger, and performing SOQL and DML on collections of sObjects instead of single sObjects at a time. The SOQL and DML bulk best practices apply to any Apex code, including SOQL and DML in classes. The examples given are based on triggers and use the `Trigger.New` context variable.

Operating on Record Sets

Let's first look at the most basic bulk design concept in triggers. Bulkified triggers operate on all sObjects in the trigger context. Typically, triggers operate on one record if the action that fired the trigger originates from the user interface. But if the origin of the action was bulk DML or the API, the trigger operates on a record set rather than one record. For example, when you import many records via the API, triggers operate on the full record set. Therefore, a good programming practice is to always assume that the trigger operates on a collection of records so that it works in all circumstances.

The following trigger assumes that only one record caused the trigger to fire. This trigger doesn't work on a full record set when multiple records are inserted in the same transaction. A bulkified version is shown in the next example.

```
trigger MyTriggerNotBulk on Account(before insert) {
    Account a = Trigger.New[0];
    a.Description = 'New description';
}
```

Copy

This example is a modified version of `MyTrigger`. It uses a for loop to iterate over all available sObjects. This loop works if `Trigger.New` contains one sObject or many sObjects.

```
trigger MyTriggerBulk on Account(before insert) {
    for(Account a : Trigger.New) {
        a.Description = 'New description';
    }
}
```

Copy

Performing Bulk SOQL

SOQL queries can be powerful. You can retrieve related records and check a combination of multiple conditions in one query. By using SOQL features, you can write less code and make fewer queries to the database. Making fewer database queries helps you avoid hitting query limits, which are 100 SOQL queries for synchronous Apex or 200 for asynchronous Apex.

The following trigger shows a SOQL query pattern to avoid. The example makes a SOQL query inside a for loop to get the related opportunities for each account, which runs once for each Account sObject in `Trigger.New`. If you have a large list of accounts, a SOQL query inside a for loop could result in too many SOQL queries. The next example shows the recommended approach.

```
trigger SoqlTriggerNotBulk on Account(after update) {
    for(Account a : Trigger.New) {
        // Get child records for each account
        // Inefficient SOQL query as it runs once for each account!
        Opportunity[] opps = [SELECT Id,Name,CloseDate
                             FROM Opportunity WHERE AccountId=:a.Id];

        // Do some other processing
    }
}
```

Copy

This example is a modified version of the previous one and shows a best practice for running SOQL queries. The SOQL query does the heavy lifting and is called once outside the main loop.

- The SOQL query uses an inner query—`(SELECT Id FROM Opportunities)`—to get related opportunities of accounts.
- The SOQL query is connected to the trigger context records by using the IN clause and binding the `Trigger.New` variable in the WHERE clause—`WHERE Id IN :Trigger.New`. This WHERE condition filters the accounts to only those records that fired this trigger.

Combining the two parts in the query results in the records we want in one call: the accounts in this trigger with the related opportunities of each account.

After the records and their related records are obtained, the for loop iterates over the records of interest by using the collection variable—in this case, `acctsWithOpps`. The collection variable holds the results of the SOQL query. That way, the for loop iterates only over the records we want to operate on. Because the related records are already obtained, no further queries are needed within the loop to get those records.

```
trigger SoqlTriggerBulk on Account(after update) {
    // Perform SOQL query once.
    // Get the accounts and their related opportunities.
    List<Account> acctsWithOpps =
        [SELECT Id,(SELECT Id,Name,CloseDate FROM Opportunities)
         FROM Account WHERE Id IN :Trigger.New];

    // Iterate over the returned accounts
    for(Account a : acctsWithOpps) {
        Opportunity[] relatedOpps = a.Opportunities;
        // Do some other processing
    }
}
```

Copy

Alternatively, if you don't need the account parent records, you can retrieve only the opportunities that are related to the accounts within this trigger context. This list is specified in the `WHERE` clause by matching the AccountId field of the opportunity to the ID of accounts in `Trigger.New`: `WHERE AccountId IN :Trigger.New`. The returned opportunities are for all accounts in this trigger context and not for a specific account. This next example shows the query used to get all related opportunities.

```
trigger SoqlTriggerBulk on Account(after update) {
    // Perform SOQL query once.
    // Get the related opportunities for the accounts in this trigger.
    List<Opportunity> relatedOpps = [SELECT Id,Name,CloseDate FROM Opportunity
        WHERE AccountId IN :Trigger.New];

    // Iterate over the related opportunities
    for(Opportunity opp : relatedOpps) {
        // Do some other processing
    }
}
```

Copy

You can reduce the previous example in size by combining the SOQL query with the for loop in one statement: the SOQL for loop. Here is another version of this bulk trigger using a SOQL for loop.

```
trigger SoqlTriggerBulk on Account(after update) {
    // Perform SOQL query once.
    // Get the related opportunities for the accounts in this trigger,
    // and iterate over those records.
    for(Opportunity opp : [SELECT Id,Name,CloseDate FROM Opportunity
        WHERE AccountId IN :Trigger.New]) {

        // Do some other processing
    }
}
```

Copy

Beyond the Basics

Triggers execute on batches of 200 records at a time. So if 400 records cause a trigger to fire, the trigger fires twice, once for each 200 records. For this reason, you don't get the benefit of SOQL for loop record batching in triggers, because triggers batch up records as well. The SOQL for loop is called twice in this example, but a standalone SOQL query would also be called twice. However, the SOQL for loop still looks more elegant than iterating over a collection variable!

Performing Bulk DML

When performing DML calls in a trigger or in a class, perform DML calls on a collection of sObjects when possible. Performing DML on each sObject individually uses resources inefficiently. The Apex runtime allows up to 150 DML calls in one transaction.

This trigger performs an update call inside a `for` loop that iterates over related opportunities. If certain conditions are met, the trigger updates the opportunity description. In this example, the update statement is inefficiently called once for each opportunity. If a bulk account update operation fired the trigger, there can be many accounts. If each account has one or two opportunities, we can easily end up with over 150 opportunities. The DML statement limit is 150 calls.

```
trigger DmlTriggerNotBulk on Account(after update) {
    // Get the related opportunities for the accounts in this trigger.
    List<Opportunity> relatedOpps = [SELECT Id,Name,Probability FROM Opportunity
        WHERE AccountId IN :Trigger.New];
    // Iterate over the related opportunities
    for(Opportunity opp : relatedOpps) {
        // Update the description when probability is greater
        // than 50% but less than 100%
        if ((opp.Probability >= 50) && (opp.Probability < 100)) {
            opp.Description = 'New description for opportunity.';
            // Update once for each opportunity -- not efficient!
            update opp;
        }
    }
}
```

Copy

This next example shows how to perform DML in bulk efficiently with only one DML call on a list of opportunities. The example adds the Opportunity sObject to update to a list of opportunities (`oppsToUpdate`) in the loop. Next, the trigger performs the DML call outside the loop on this list after all opportunities have been added to the list. This pattern uses only one DML call regardless of the number of sObjects being updated.

```
trigger DmlTriggerBulk on Account(after update) {
    // Get the related opportunities for the accounts in this trigger.
    List<Opportunity> relatedOpps = [SELECT Id,Name,Probability FROM Opportunity
        WHERE AccountId IN :Trigger.New];

    List<Opportunity> oppsToUpdate = new List<Opportunity>();
    // Iterate over the related opportunities
    for(Opportunity opp : relatedOpps) {
        // Update the description when probability is greater
        // than 50% but less than 100%
        if ((opp.Probability >= 50) && (opp.Probability < 100)) {
            opp.Description = 'New description for opportunity.';
            oppsToUpdate.add(opp);
        }
    }
    // Perform DML on a collection
    update oppsToUpdate;
}
```

Copy

Bulk Design Pattern in Action: Trigger Example for Getting Related Records

Let’s apply the design patterns you’ve learned by writing a trigger that accesses accounts’ related opportunities. Modify the trigger example from the previous unit for the `AddRelatedRecord` trigger. The `AddRelatedRecord` trigger operates in bulk, but is not as efficient as it could be because it iterates over all `Trigger.New` sObject records. This next example modifies the SOQL query to get only the records of interest and then iterate over those records. If you haven’t created this trigger, don’t worry—you can create it in this section.

Let’s start with the requirements for the `AddRelatedRecord` trigger. The trigger fires after accounts are inserted or updated. The trigger adds a default opportunity for every account that doesn’t already have an opportunity. The first problem to tackle is to figure out how to get the child opportunity records. Because this trigger is an *after* trigger, we can query the affected records from the database. They’ve already been committed by the time the after trigger is fired. Let’s write a SOQL query that returns all accounts in this trigger that don’t have related opportunities.

```
[SELECT Id,Name FROM Account WHERE Id IN :Trigger.New AND
                                             Id NOT IN (SELECT AccountId FROM Opportunity)]
```

Copy

Now that we got the subset of records we’re interested in, let’s iterate over those records by using a SOQL for loop, as follows.

```
for(Account a : [SELECT Id,Name FROM Account WHERE Id IN :Trigger.New AND
                                             Id NOT IN (SELECT AccountId FROM Opportunity)]){
}
```

Copy

You’ve now seen the basics of our trigger. The only missing piece is the creation of the default opportunity, which we’re going to do in bulk. Here’s the complete trigger.

1. If you’ve already created the `AddRelatedRecord` trigger in the previous unit, modify the trigger by replacing its contents with the following trigger. Otherwise, add the following trigger using the Developer Console and enter `AddRelatedRecord` for the trigger name.

```
trigger AddRelatedRecord on Account(after insert, after update) {
    List<Opportunity> oppList = new List<Opportunity>();
    
    // Add an opportunity for each account if it doesn't already have one.
    // Iterate over accounts that are in this trigger but that don't have opportunities.
    for (Account a : [SELECT Id,Name FROM Account
                     WHERE Id IN :Trigger.New AND
                     Id NOT IN (SELECT AccountId FROM Opportunity)]) {
        // Add a default opportunity for this account
        oppList.add(new Opportunity(Name=a.Name + ' Opportunity',
                                   StageName='Prospecting',
                                   CloseDate=System.today().addMonths(1),
                                   AccountId=a.Id));
    }

    if (oppList.size() > 0) {
        insert oppList;
    }
}
```

Copy

2. To test the trigger, create an account in the Salesforce user interface and name it `Lions & Cats`.
3. In the Opportunities related list on the account’s page, find the new opportunity Lions & Cats. The trigger added the opportunity automatically!

Resources

- [*Apex Developer Guide*: Triggers]()

Get Ready

You'll be completing this challenge in your own personal Salesforce environment. Choose from the dropdown menu, then click **Launch** to get started. If you use Trailhead in a language other than English, set the language of your Trailhead Playground to English before you attempt this challenge. Want to find out more about using hands-on orgs for Trailhead learning? Check out the [Trailhead Playground Management]() module.

Your Challenge

Create an Apex trigger for Opportunity that adds a task to any opportunity set to 'Closed Won'.
To complete this challenge, you need to add a trigger for Opportunity. The trigger will add a task to any opportunity inserted or updated with the stage of 'Closed Won'. The task's subject must be 'Follow Up Test Task'.

- The Apex trigger must be called 'ClosedOpportunityTrigger'
- With 'ClosedOpportunityTrigger' active, if an opportunity is inserted or updated with a stage of 'Closed Won', it will have a task created with the subject 'Follow Up Test Task'.
- To associate the task with the opportunity, fill the 'WhatId' field with the opportunity ID.
- This challenge specifically tests 200 records in one operation.

My Trailhead Playground 3

[Launch]()

Check challenge to earn 500 points