



**Using Map based query**

List<Account> lstacc=[Select Id from Account limit 10000];

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

for(Account a:lstacc){ //More CPU time for sure due to looping

setIds.add(a.id);

}

//Using Map query saves CPU time

Map<id,account> aMap = new Map<id,account>([Select Id,Name from Account limit 50000]);

//Creating list of accounts

List<account> accList = aMap.values() ;

//Creating set of ids

Set<id> accIds = aMap.keySet() ;

## **Using Collections, Streamlining Queries, and Efficient For Loops**

It is important to use [Apex Collections](http://www.salesforce.com/us/developer/docs/apexcode/index_Left.htm#StartTopic=Content/apex_methods_collections.htm) to efficiently query data and store the data in memory. A combination of using collections and streamlining SOQL queries can substantially help writing efficient Apex code and avoid governor limits.

Using map-based queries reduce the processing time considerably.

Likewise, do not write the logic to create a map with key as id of the record and value as the record itself, use inline maps.

//rather do this

Map<id,account> accountsMap = new Map<id,account>([Select Id,Name from Account limit 50000]);

//Creating list of accounts

List<account> accList = accountsMap.values() ;

//Creating set of ids

Set<id> accIds = accountsMap.keySet() ;

**Aggregate SOQL usage**   
Since the database time is not calculated in CPU time its always better to explore the usage of aggregate SOQL for your business use case .  
  
Say you want summation of field value  of some records ,if you use normal for loop to get these obvious you have spent CPU time there .Instead try to push your calculation using SUM,AVG aggregate functions at the database layer itself so that you have reduced CPU time and have pushed the process on database layer itself. Explore options to group by or create some sort of filtering at database layer and push your calculations at database layer to reduce chances of hitting CPU time out issue .

I.e., All the information that can be calculated using SOQL queries, it's better that we push it to SOQL by make proper use of notions like AVG(), SUM(), Count() and etc, rather than writing the business logic to calculate.

Because, when we write the logic using APEX to aggregate the results it will be counted against the 10 secs CPU time.

**Only take necessary data and run a loop**  
This is essential now to filter only specific data while doing a for on a list of records as too much looping will increase CPU time .Same held good when we had no of script statements limit .

**Avoid nested for loops**

When we have large data sets, quite often I have seen developers using nested for loops to process them.

It's better we avoid using nested for loops, it throttles the server resources.

**Enforcing the current user’s sharing rules can impact:**

SOQL and SOSL queries. A query may return fewer rows than it would operating in system context.

DML operations. An operation may fail because the current user doesn’t have the correct permissions. For example, if the user specifies a foreign key value that exists in the organization, but which the current user does not have access to.

Minimize the number of data manipulation language (DML) operations by adding records to collections and performing DML operations against these collections.

Minimize the number of SOQL statements by preprocessing records and generating sets, which can be placed in single SOQL statement used with the IN clause.

**Process Builder:** If you have a process builder on the Billing object, disable some of the process builder flows or move its logic to code which executes using an asynchronous approach.

Convert Process Builder processes to Apex Triggers if possible.

**Another Managed Package Involved:** If code from another managed package (i.e., Rollup Helper) is taking part in a transaction such as Posting a Billing you will need to contact the authors of the that managed package.

**Explore if your business allows you to do the operation asynchronously**   
In some cases business process may not be real time and hence if there is a chance to make code execute in @future ,this will break the context and also the CPU time out limit for asynchronous process is 60seconds(6X of synchronous process).So, consider this if you're in trouble of hitting the limit.

Recursion

1. Recursion trigger
   1. Self-Relationship. Husband contact record lookup relation mapped to Spouse Contact Record and then Spouse contact record lookup relation mapped to Husband Contact Record. Trigger call itsef to update the lookup fields
   2. Account Trigger call Contact Trigger and then Contact Trigger Calls Account Trigger to update the rollup summary fields etc
   3. Trigger updates the field value, then workflow rule updates. And if workflow field update rule executes, then again trigger executes
   4. Trigger calls callout through future method or outbound messages, if we update the same object with callout response, again future methods or outbound messages executes.
   5. Declarative side effect: save order life cycle

Workflow field update always leads the trigger to execute 2nd time.

|  |  |
| --- | --- |
| **Counts** | **Doesn't Count** |
| Any Apex Code | HTTP Callouts |
| Workflow Execution | Database operations, e.g., DML, SOQL |
| Library functions exposed through Apex | SOSL |
| Any time spent evaluating formulas for validation/workflows |  |