QUERY EFFICIENCY

It's important to be **efficient** when querying the database (which includes GlideRecord queries). To that end, this section contains some basic guidelines for making sure your queries are efficient.

Inefficient database operations can be the source of a multitude of performance woes, including client-side issues like fields taking a long time to update, or the browser locking up when you perform certain actions – and server-side issues, such as lists, forms, and dashboards simply taking a long time to load. Most commonly, the culprit is one (or many) inefficient or non-optimized queries.

We've already gone over how to improve client-side performance by using asynchronous queries or other database, server, and API calls. In this section, we're going to learn how to write **queries** in such a way that there is an "as-minimal-as-possible" performance hit.

QUERY SPECIFICITY

You should generally try to make your queries as **specific** as possible. For example, if you only do something with the records returned from your query *in the event* that a specific condition is true, then that condition should be **part of the query**!

Consider the following code:

```
   var grIncident = new GlideRecord('incident');
grIncident.addActiveQuery();
grIncident.query();
while (grIncident.next()) {
    if (grIncident.getValue('state') === '3') {
       gs.print('Incident ' + grIncident.getValue('number') + ' is in state: 3.');
    }
}
```

Fig. 9.01

In this example, our database query would be monumentally more efficient if we were to add the condition currently in the `if` block, to the query itself, like so:

```
   var grIncident = new GlideRecord('incident');
grIncident.addActiveQuery();
grIncident.addQuery('state', '3');
grIncident.query();
while (grIncident.next()) {
    gs.print('Incident ' + grIncident.getValue('number') + ' is in state 3.');
}
```

Fig. 9.02

There; that's **much** better!

One last note on the topic of query specificity: If you ever find yourself checking some field value inside of your query code block and only doing something if some condition is met, consider whether it would be possible to add that condition to the query itself.

For example, consider the following code:

```
   var grOpenInc = new GlideRecord('incident');
grOpenInc.addQuery('active', true);
grOpenInc.query();
while (grOpenInc.next()) {
    if (grOpenInc.getValue('priority', 1)) {
       // Do something with the incident
       gs.debug(
          'Processing incident: ' +
          grOpenInc.getValue('number')
       );
    }
}
```

In this code, we're only doing something if the priority of the Incident is 1. So, wouldn't it be significantly more efficient if we just added that condition to the query, as in the below example?

```
   var grOpenInc = new GlideRecord('incident');
grOpenInc.addQuery('active', true);
grOpenInc.addQuery('priority', 1);
grOpenInc.query();
while (grOpenInc.next()) {
    gs.debug(
       'Processing incident: ' +
       grOpenInc.getValue('number')
    );
}
```

SINGLE-RECORD QUERIES

Any time you use an `if` block rather than a loop (such as **`if`** `(grInc.next()) {}` rather than **`while`** `(gr.next()) {}`[1]), that means you're only looking for **one** record. The most efficient way to do this is to use the GlideRecord `.get()` API, and pass in a single argument: a **sys_id** or an **encoded query**. If it is not possible to specify a sys_id, there is a server-side version of the `.get()` API which can accept a query instead of a sys_id. However, due to the difference in client and server-side APIs, and for readability, it is best to use `.setLimit()` instead, whenever you can't specify a sys_id. Also, when a sys_id is *not* specified to the `.get()` API, it may return multiple records – which can be confusing if you expect only one record, and also has a deleterious impact on performance, since the query continues after finding one record, to see if it can find more. For these reasons, it's often a good idea to just stick with using `.setLimit(1)`.

You can use `grInc.setLimit(1)` to tell the database to stop searching after the first record is found. Otherwise, this would be like continuing to search for your keys, after you've found them. By the same token, if you are only expecting (or if you only *want*) a certain maximum number of records to be found, be sure to use `.setLimit()` to make the query easier on the database, and improve performance. Failing to set a limit on your query (or use the `.get()` API) will cause the database to continue searching the **entire table** that you've specified, even after it's found what it was looking for.

If you'd like to see how many "single-record query violations" you have in your instance, you can run a read-only background script I've written for just that purpose. You can find that script at https://setlimit-gist.snc.guru. That script will tell you not only how many violations you have in your instance but also give you a link to the script in question, tell you exactly which variable contains the violation, what function the violation is in, and even tell you on what **line** that variable is initialized. Just bear in

[1] *Note that you should never use the variable name "`gr`" in real-world code. It is not descriptive or useful, other than indicating that it hopefully contains (references) some kind of GlideRecord. See the Naming Conventions > Variables > GlideRecords section for more info.*

mind that this script will probably identify more out-of-box records with blatant performance violations than your own scripts... but I'll leave it to you to decide what you think about that[2].

NESTED QUERIES

It's a good idea to avoid using **nested queries** if at all possible. This is because nested queries usually require a separate "inner" query for every single loop of the "outer" query, and can almost always be written more efficiently.

Nested queries are often incorrectly used when one needs to perform one query based on the records found based on another query.

Consider the following code:

```
   var grUser;
var grIncident = new GlideRecord('incident');
grIncident.addEncodedQuery('some_query'); //include a query param for 'assigned_to not blank'
grIncident.query();
while (grIncident.next()) {
    grUser = new GlideRecord('sys_user');
    if (grUser.get(grIncident.getValue('assigned_to'))) {
       //do something with the user
    }
}
```

Fig. 9.03

As you can see in the preceding code block, this would result in a **separate query** for each record returned from the "outer" query on the Incident table. If you get 10,000 Incidents from the outer query, that'll result in **10,000 separate queries** on the *sys_user* table (10,001 total, including the Incident query)!

As you might imagine, 10,001 separate queries could take quite some time. Instead, it would be dramatically more efficient if we were to use the first query to build a list of *sys_user* records we want to get from the database, then do a single query to get all of them. Rather than **10,001** queries, we're just doing two! Even though we're getting just as many records as the previous queries, we're doing so in one fell swoop, rather than going back-and-forth to the database to get each record.

Consider the following code as an example of dramatically improved efficiency:

```
   var grUser, assignIDs = [];
var grInc = new GlideRecord('incident');
grInc.addEncodedQuery('some_query^asigned_to!=NULL');
grInc.query();
while (grInc.next()) {
    //This condition keeps the array values unique
    if (assignIDs.indexOf(grInc.getValue('assigned_to') < 0)) {
       assignIDs.push(grInc.getValue('assigned_to'));
    }
}

grUser = new GlideRecord('sys_user');
grUser.addQuery('sys_id', 'IN', assignIDs);
/*Include any other query params here, such as if you
only want to return active users.
Note that it is more efficient to add those query
params here, rather than via dot-walking on the
Incident table query, because dot-walking inside
of one query effectively results in a separate
query on the dot-walked table (which we're doing
here, anyway!)*/
```

[2] ʊ_ʊ

```
grUser.query();
while (grUser.next()) {
    //NOW do something with the assignee records
}
```

Fig. 9.04

***Note**: Although I didn't want to break the flow of the example code, it would actually be far more efficient to use* `GlideAggregate` *and the* `.groupBy()` *API for the first query in the preceding example, because we only need the unique* `assigned_to` *values!*

There is one down-side to the preceding example: When you do the second query, you lose the "context" of the Incident to which the user was assigned, because we didn't put that data into our array. This can make things difficult, if the action we want to take on the user record is *dependent* on some data in the Incident. Not to worry though, you can easily remedy this by storing this data in an **object** rather than an **array**!

In the example code below, we'll construct an object from each Incident returned from the first query, and use the data in that object to do the work in our second query:

```
   var inc, incAssignedToUser, grUser, assignID, incID;
var assignData = {};
var grIncident = new GlideRecord('incident');
grIncident.addEncodedQuery('some_query^assigned_to!=NULL');
grIncident.query();
while (grIncident.next()) {
    incID = grIncident.getValue('sys_id');
    assignID = grIncident.getValue('assigned_to');
    /*If the assignee has not been added to the
    assignData object, create it.*/
    if (!assignData[assignID]) {
       assignData[assignID] = {};
    }
    /*Create an object inside the assignee object in
    the assignData object, to hold the incident
    data (using the Incident() ES5 constructor)*/
    assignData[assignID][incID] = new Incident(grIncident);
}

grUser = new GlideRecord('sys_user');
grUser.addQuery('sys_id', 'IN', Object.keys(assignData));
grUser.query();
while (grUser.next()) {
    for (inc in assignData[grUser.getValue('sys_id')]) {
       //Check if property is enumerable
       if (assignData.hasOwnProperty(inc)) {
          incAssignedToUser = assignData[grUser.getValue('sys_id')][inc];
          //NOW do something with the user/incident pair
       }
    }
}

/**
 * @param grIncident {GlideRecord}
 * @constructor
 */
function Incident(grIncident) {
    this.sys_id = grIncident.getValue('sys_id');
    this.number = grIncident.getValue('number');
    //etc...
}
```

Fig. 9.05

This code uses a constructor function (`Incident()`) to generate a structured object from the `grIncident` GlideRecord object, to store all of the data about the Incident that we might need to use later. In this way, we're "building a to-do list" which we then leverage in our second query. In order to keep our second query efficient and limit it to just the records we know we'll need to interact with, we use `Object.keys(assignData)`, which returns an array of the keys in the `assignData` object (which you may have noticed, are the sys_ids of the assignees in question!)

Granted, the preceding code is... quite a lot longer than the first, or second examples. However, it is also a **lot** more efficient, and provides whatever additional context you need from the Incident records, when working with the user/assignee records.

***Pro-tip**: Should you need to, you can get an array of the sys_ids to which a specific user is assigned in the final `while` loop, using something like:*

```
Object.keys(assignData[grUser.getValue('sys_id')])
```