

# How to create a performing worklist

When creating a dynamic case management application, you want to show a [AQ\\_WorkList](#) to the knowledge worker in which he/she can see the available tasks. As end users do not like to wait for relevant information to load on the screen, this worklist needs to be able to retrieve the relevant information fast. Blueriq makes the worklist as fast as possible from a technical point of view, but there are guidelines for you as Business Engineer to choose settings that result in a better performance.

Although this page concerns the worklist, all advice also holds for both the [AQ\\_CaseList](#) and the [AQ\\_TaskList](#). As these list automatically returns less results, it is less likely that you experience performance problems with it.

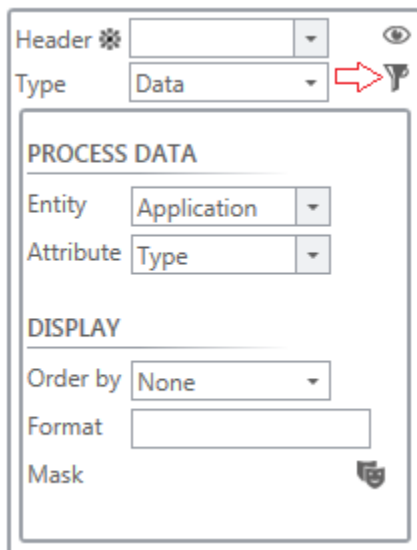
## Step-by-step guide

When a dynamic case management solution has been in production for a significant amount of time, the number of cases and tasks that are present in the database can also grow increasingly, if more cases are created than closed. Think of long-running cases that take months or years to finish. End users want to experience the same performance with such a full database as with an empty database. An important factor for a performing worklist is the number of results that match your search. If you model a worklist that always returns all tasks, then this worklist will not perform with an increasingly large database. We advise to create a worklist that filters as many results as possible, in order to create a good experience for the end user. This article describes some pointers that help you achieve a good performance.

1. **Avoid the show all tasks option.**

If this option is unchecked, the worklist only shows tasks that are routed to the current user. This should limit the result set significantly. If you check this option, you should apply filters to limit the result set, as shown in the next steps.

2.



**Set a filter on a column.** This filter should configure the list to only show certain tasks. For example, an insurance application can be for a car insurance and a health insurance. As a knowledge worker may have a specialization on car insurances, it would be good to only present tasks to him/her that belong to car insurance applications. It is possible to filter on a column that is not shown to the user by using the `Visible` setting (above the filter). A list for health insurance tasks can be modeled on a different page.

3. You can let the end user help to **refine the search**. It is possible to use an attribute in a filter expression that was placed on the page for user input. You can use this to drastically reduce your search result. A good design here is that the attribute is initially placed on a page where the worklist is not visible (either its precondition is false, or it is on a different page), and only after the user supplied filter information to show the list. After the filtered list is shown, you could let the end user refine the search, by using a refresh on the attribute. In this way there is no moment in which an unfiltered list is displayed to the end user.
4. **Distribute entries across multiple worklists.** If you for example have three categories for tasks with low, medium and high priority, you can model that as three different worklists. A page is only loaded when all content on it has been loaded. Having 3 worklists on the same page means that the page is not displayed until the entries for the last worklist are retrieved. However placing the three worklists in different widgets results in the behavior that the content of the first worklist is shown as soon as it is ready, after which it is proceeded to loading the second widget, etc. With this behavior you can influence what the user sees first, and load more important worklists faster. By splitting up a worklists into multiple worklists you will also not encounter the problem of reaching the maximum retrieved entries as easily.
5. You can **limit the number of the results coming from the database**. This setting was introduced in version 9.4. This is a configurable runtime setting that will speed up the worklist for large result sets.

In the properties file you can set the limit of the worklist and caselist by parameters.

Java

Setting the limit is done in `spring.config.additional-location`.

```
processengine.worklist.limit=x
processengine.caselist.limit=x
```

You can configure the message that is displayed to the end user in the *message.properties* (both Java and .Net)

```
worklist.paging.limit.applied.feedback=Results are limited to the maximum of {0}
caselist.paging.limit.applied.feedback=Results are limited to the maximum of {0}
```

There is one disadvantage however, for versions 9.4 up to 9.8. The sorting and paging of the results is done after the results are retrieved, which means that your sorting might not be reliable.

For example, you limit the retrieved results to 100.000 entries. However, 150.000 entries in the database match your search. You sort the 100.000 entries that were retrieved from the database on priority. The end user might think that he is performing the task with the highest priority, but this might not be true as the highest priority task could be among the 50.000 tasks that were not retrieved. The end user may also have the impression that there are fewer tasks in the system than actually present, as paging will show for example 10.000 pages of 10 tasks each.

From version 9.9 and later this mechanism has been improved. From that version on, only one page of data is read from the database, and this one page is correctly sorted. There are two reasons why this setting is still useful. (1) When creating a styling for the pagination of the worklist that uses a dropdown box, then this dropdown box can become very large. When having for example 10.000 pages the browser might need to work very hard to create all the entries and possibly freeze or crash. (2) Choosing a page with a high page number takes longer than opening a page with a low page number. With this setting you can reduce the number of pages so that the end user can not choose such a high page number. We believe that there is no use case to go to pages with a very high number, as you would rather refine your search using the filters or use sorting.

6. **Update to the latest version of Blueiriq.** Although this advice holds in general, we have made significant performance improvements in 9.9 and up. Updating to the latest version ensures that you can get the maximum out of your worklist, without the need to change how you model your application. We will be doing more performance improvements in the future as well.

## Advice for database maintenance

Blueiriq provides a set of general-purpose indexes, but for large databases we recommend dropping the indexes that aren't used in your application, as well as adding any necessary indexes that result in better execution plans for the most frequently used queries. The queries generated by AQ\_WorkList and AQ\_CaseList can be monitored by setting the log level of `org.hibernate.SQL` to `DEBUG`. A few examples of indexes you might create depending on the configuration of your worklist and the usage patterns of your users are:

- if your worklists have custom field columns that are often used for searching, you should create index (taskId, name) on the customFields table
- if your worklists have data columns which are often used for searching, you should create the following composite indexes:
  - (caseId, entityName) on the instances table
  - (instanceId, attributeName) on the instanceAttributes table
  - (instanceAttributeId, stringValue) on the attributeValues table (note: use the appropriate "value" column depending on the data types of the attributes shown in your worklists)
- if your worklists use the CaseID column and this is used for filtering (in Studio) or searching (by users on the frontend), you should create composite indexes which start with the caseID column. For example, if your worklist has columns Case ID, Task Name and Priority, there is a filter on the Case ID column in the Studio model and the users often search on the Task Name column, you should create the composite index (caseId, name) on the tasks table.

Keep in mind that query performance depends not only on the indexes but also on the data and the usage patterns of your users. Periodically monitor the execution plans of the worklist queries on real data and determine whether any changes to the execution plans occurred since the last check. Add indexes where appropriate in order to maintain good performance.

We also recommend that you monitor index fragmentation in production databases, as index fragmentation can lead to poor query performance. Your indexes become more fragmented the more data is inserted or deleted. Implement an index maintenance plan to reorganize or rebuild your indexes once the fragmentation level exceeds a set threshold.

---

## Related articles

- [How to use Decision Requirements Graphs to visualize ad-hoc tasks in business process modeling](#)
- [How to handle long-term monitoring and archiving](#)

