Tivoli Netcool Support's Guide to Tuning the common Netcool/OMNIbus triggers

by

Jim Hutchinson

Document release: 2.0

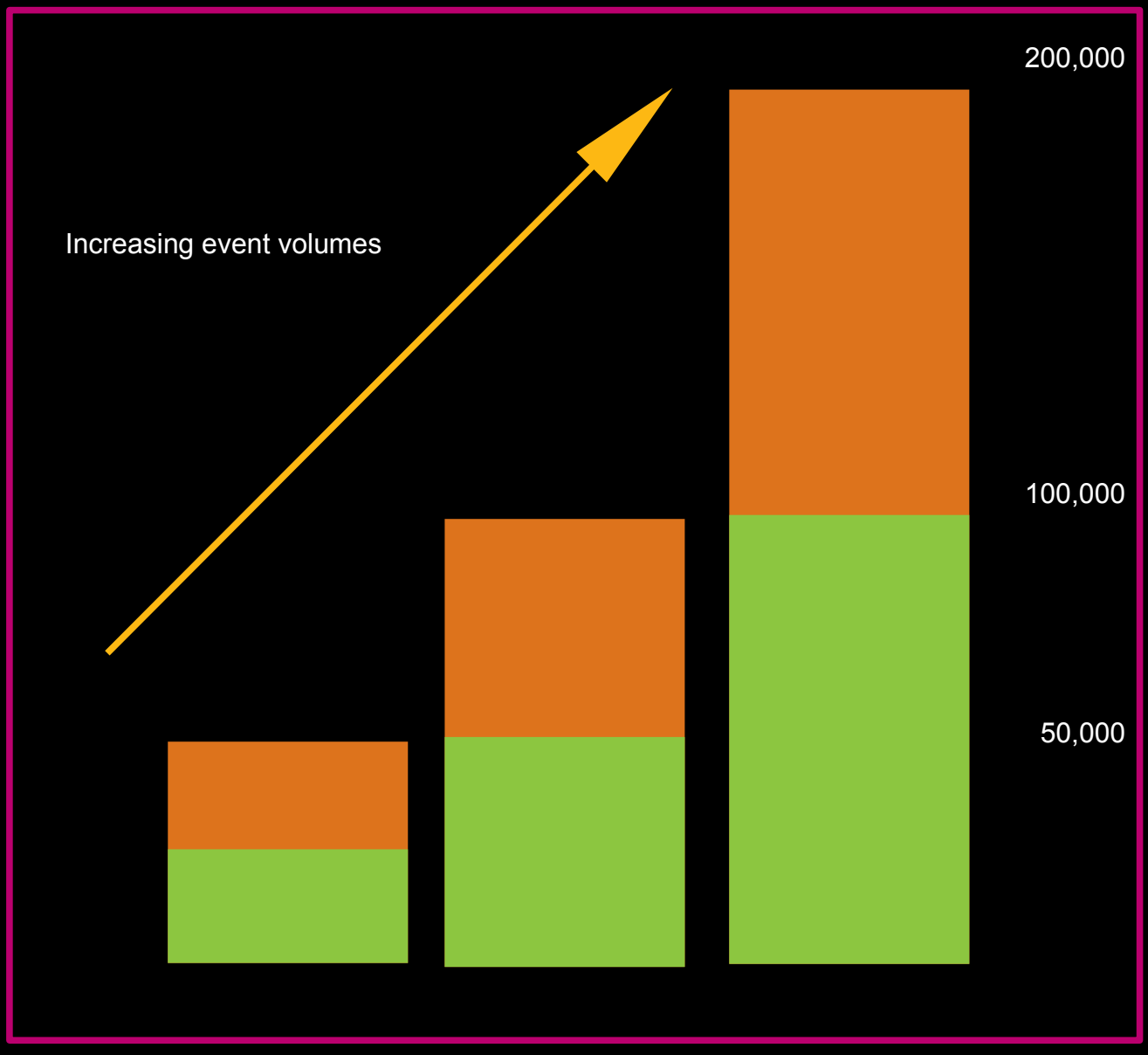
Table of Contents

1 Introduction

The Netcool/OMNIbus Object Server triggers were written when the Object Server handled a maximum of 50,000 active alarms, and these alarms were processed in realtime to clear events. With the expected standing event row count being around 20,000. With modern systems, there is an increase in the headroom available on the Object Server through the use of 64-bit technology and increased hardware, especially with respect to the available memory. The volume of events managed by the Netcool/OMNIbus object server has increased significantly, and maximum event volumes and standing events are no longer limited to the tens of thousands.

Because of these increased volumes, it will be necessary to ensure that the Object Server triggers and probe rules files are tuned to the specific event volume environment. The nature of the common triggers and how the Object Server processes events needs to be taken into account, so as to minimise loads and maximise event processing performance.

This document was written to highlight the need to understand the system, and tune common triggers, so as to prevent issues seen when event high volumes are normal. The examples customisations are provided as one possible solution, and it is expected that the administrator of the system will use them to customise Netcool/OMNIbus to their specific requirements.



2 Performance Considerations

All tests were performed on a stand-alone Object Server in the Solaris environment.

2.1 Object Server Table sizes

The main tables in the object server are alerts.status, alerts.journals and alerts.details. The number of columns and number of rows in a table affects performance. The alerts.journal and alerts.details tables have fixed columns, and so only the number of rows can affect performance. It is recommended that the alerts.details table is not populated in a production Object Server as in general, any usable data is stored in the alerts.status table, or in alerts.journal. The alerts.journal table is populate by triggers and user actions, and so care should be taken to ensure that only the required number of journal entries are made by triggers, and duplicate journal entries are avoided. For example, if one trigger adds just one unnecessary journal entry on each event, this will add significantly to the number of rows in alerts.journals. As a general rule, the number of rows in alerts.journal does not exceed five times the number of rows in alerts.status.

Typical dynamic table usage:

Table	Size
alerts.status	N
alerts.journal	5*N
alerts.details	0

2.2 Table Indexes

The manual discusses how indexes can be used to improve performance but does not recommend any indexes. The general guidance suggests that you can review your object server using nco_config and determine the best fields to have indexes on for a table:

In the Database Data View tab for a table, click the right menu and select Indexes->Column Selectivity.
High: Values in the table are at least 90% unique. These represent the ideal selectivity rating for indexing.

After reviewing performance for high volumes of events, the best performance was achieved when an index was added to the ServerSerial column in the alerts.status table.

```
-- Add indexes for performance
create index AlertsStatusServerSerialIdx on alerts.status (ServerSerial);
```

With this index in place, even the default triggers performed **exceptionally well**, together with the custom triggers the performance was improved significantly. No significant increase in memory usage was observed during testing.

Adding an index to the alerts.status ServerSerial column, which is always set to a unique or near unique value, allows the object server to find the events much quicker when running triggers, especially when the triggers reference events via ServerName/ServerSerial.

Notes:
You can create and drop indexes in an Object Server to confirm performance improvements in test environments.
The virtual table used in the generic_clear trigger has a hash index by default, and Primary Key fields cannot be indexed.
ServerSerial is set to the Serial value of the first Object Server the event is created in, with ServerName being set to the source Object Servers name. In general the fields uniqueness will be related to the number of collection Object Servers, plus the two Aggregation Object Servers.

2.3 Back-up frequency

The default backup frequency of the Object Server is every 5 minutes, which is very high, for a large Object Server. Given that the backup tab files are there to allow recovery of the Object Server data, unless the dumped files are copied to another location, in all probability the last backup tab files will be those of the current tab files, less five or ten minutes of data.

Trigger : automatic_backup
Default Frequency : 5 minutes
Backup command issued :
alter system backup '$OMNIHOME/backup/' + getservername() + '/BACKUP_' + to_char(backup_dir);

Consider that the back-up tab files need to be used to replace the current tab files, due to some catastrophic failure. In such a scenario it is best to dump the tab files hourly, to one of two locations, therefore allowing the system to be recovered to a point within an hour of failure. To achieve this, update the period of the automatic_backup trigger to every hour. By default the trigger works through two back-ups as seen in the trigger:

set **num_backups** = 2;

If more back-ups need to be maintained, then increase the **num_backups** to the required value in the trigger as required.

Ensure that the target directories exist before enabling the trigger:
e.g.
mkdir $OMNIHOME/backup/NCOMS/BACKUP_0
mkdir $OMNIHOME/backup/NCOMS/BACKUP_1

Note : For very large Object Servers the impact of performing a back-up will cause the Object Server to be unable to process events until the back-up is completed, due to table locks.

2.4 The Identifier String

The Object Server tables, alerts.status uses the Identifier field as a unique identifier in the table. Identifier is also used in the alerts.details table, whilst in the alerts.journal table the alerts.status Serial field is used. It is therefore important to ensure that the Identifier field is set to a string whose size is minimised, therefore reducing the memory footprint of the Object Server and its memory usage when performing selects containing the Identifier field.

Within the multitier system the following fields are used to identify an event:

- Identifier
- ServerName + ServerSerial

Note that the Object Server Serial is unique within each object server only, and ServerName+ServerSerial will generally be the ServerName and Serial from one of the collection Object Servers.

With this in mind, it is possible to reduce the overall Object Server size, by adopting ServerName+ServerSerial as the Identifier in the Aggregation and Display layers.

You can also ensure Identifier is minimised by using the Generic Clear fields to define the events, and ensure that these fields too are optimized.

The String fields used by the Generic clear trigger are:

- Node
- Manager
- AlertKey
- AlertGroup

2.5 The Generic clear trigger

The Generic clear trigger is used to perform problem/resolution.

It refers to the alerts.status Type=1 events and populates the virtual table alerts.problem_events with the resulting selected fields.

Because the trigger uses a for each loop to select Type=1 events, it is important to ensure only events that will have a Type=2 event have Type=1 set.
e.g.
for each row problem in alerts.status where problem.Type = 1
In addition the resolution events are selected, for comparison.
e.g.
(select Node + AlertKey + AlertGroup + Manager from alerts.status where Severity > 0 and Type = 2)
Therefore if these selects include events that are never used in problem/resolution they impact memory and CPU usage unnecessarily. Additionally the generic clear fields need to be minimised in size, so as to minimise memory usage, when performing selects.

The second for each loop works through the resolution events:
e.g.
for each row resolution in alerts.status where resolution.Type = 2
Which means that the generic clear trigger could potentially be looping through the entire alerts.status table, if only Type=1 and Type=2 events exist.

The third for each loop performs an update to the alerts.status table, setting all the problem events to Severity=0.

With the last action in the trigger being to remove the contents of the alerts.problem_events table. Deleting very large tables may cause noticeable Object Server locking, so it is important to keep the total number of resolved events to a manageable volume. For example, check behaviour of the object server in a test environment where 50% of the expected events are resolved in one IDUC period, so as to understand how best to tune the system.

2.6 The clean table triggers

The trigger that purges the alerts.details table performs a select on the alerts.status table, in order to determine which rows need to be removed. For alerts.details the Identifier is used, highlighting the need to prevent alerts.details being created, and for the Identifier field being minimised.

Trigger: clean_details_table

```
delete from alerts.details where Identifier not in (select Identifier from alerts.status);
```

The custom solution uses a database trigger to store the Serial and Identifier of the deleted event, so that the related alerts.details entries can be deleted later. This reduces the memory usage of the Object Server when there are high event volumes, as only the events that are to be deleted are held in memory, and not the select lists used by the default triggers.

3 Example Triggers

3.1 Generic Clear performance

The standard generic_clear trigger uses @Identifier to clear events:

update alerts.status via problem.Identifier set Severity = 0;

The custom_generic_clear triggers use ServerName + ServerSerial so that any character issues are avoided and less memory is required.

The generic_clear_snss_subsecond.sql generic_clear trigger includes the sub-second solution using ProbeSubsecondId value to clear problem events with a same second resolution.

IMPORTANT NOTE:
The generic_clear relies on the select :
(select Node + AlertKey + AlertGroup + Manager from alerts.status where Severity > 0 and Type = 2)
being small in comparison to the number of events.

When this select and the for-each problem_events returns more than 50,000 rows the object servers performance is severely impacted.

To prevent this only problem/resolution events should have Type=1|2 set.
Increase the generic_clear triggers period if more than 50,000 rows are expected.

In order to reduce loading when there are no resolutions [Type=2] additional logic was added to count the number of resolution events, so that the tables scans only happen when required.

Additionally the Production Triggers temporal trigger solution was added to prevent the custom_generic_clear from running on top of itself, which causes Object Server locking under high event loads.

Temporal trigger : custom_generic_clear
generic_clear_snss_subsecond.sql
Requires:
temporal_trigger_monitor.sql

3.2 Clean alerts.journal and alerts.details

Because clean_journal_table and clean_details_table use selects on the tables current context, large tables impact the object servers performance significantly.

The custom_clean_child_rows and clean_details_table triggers replace the two standard triggers by storing the related Serial and Identifier values for deleting the alerts.journal and alerts.details rows leter on. The reason why a temporal trigger is used, is because it was found that the database trigger did not function well under heavy problem/resolution loading. By performing a periodic delete of the data it was found that memory increase in storing the Serial and Identifier was not significant when compared to the performance improvement.

Datbase trigger : custom_clean_child_rows

clean_database_trigger.sql

3.3 Example Improved Performance

The Object Server was loaded with 100,000 rows of Problem [Type=1] events – no problem/resolution.

Because the object server is not performing any clearing it should not use any resources, either in generic_clear or for purging the tables. However, because the tables are scanned, a load is placed on the Object Server. With the custom triggers no purge selects are made unless events are set for deleting, and no table scans happen unless there are Resolution [Type=2] events.

In a production system both clearing and event deletion will be happening, however, for the periods where no actions are required, the Object Server is allowed to rest. This pause time maximises the efficiency of the Object Server, allowing other tasks to be performed.

With default triggers:

Trigger Profile Report
Trigger time for 'clean_details_table': 1.313872s
Trigger time for 'clean_journal_table': 1.012451s
Trigger time for 'generic_clear': 6.987523s
Time for all triggers in report period (60s): 9.720437s

With custom triggers:

Trigger Profile Report
Trigger time for 'custom_generic_clear': 1.201321s
Time for all triggers in report period (60s): 1.613772s

4 General guidance for very large Object Servers

4.1 Problem events

The generic_clear trigger does not handle large volumes of reoccurring Problem/Resolution (Type=1|2) events well, as it will select all of the new Resolution events with Severity greater than zero periodically. Therefore if there are large amounts of incoming Resolution events with no corresponding Problem event, the trigger performs unnecessary work. Therefore it is important to only set Type=1|2 for Problem/Resolution event pairs.

In addition for systems with large event volumes and significant volumes of Problem/Resolution events, it may be necessary to increase the period of the generic_clear trigger. To confirm this log the time when the generic_clear trigger is run, into a custom log file, and compare this time with the current period of the trigger.

You can improve the performance of table scans used in the generic_clear trigger with the use of indexes. You can review how to determine which columns in your object server may need indexes using the nco_config tool and the column views menu item Indexes->Column Selectivity.

4.2 Standing events

Where the number of standing events is high, it is important to ensure old unwanted events are purged. This can be achieved using the ExpireTime fields and triggers. Ensure that the ExpireTime is set for the Object Servers layer, for example in the Collection layer the field CollectionExpireTime is used rather than ExpireTime, with the purge trigger being called col_expire.

4.3 Buffering

In general, using probe and gateway Buffering, will improve performance. However, very large buffer sizes will affect the Object Servers performance detrimentally. This is because the insertion of events begins to interfere with the Object Servers triggers and general housekeeping. Therefore buffer sizes should be kept in step with the performance of the Object Server and its ability to process events. For example, a BufferSize of 100 or 500 may work well for a given Object Server environment. However, in another environment where the operating systems CPU is less powered, a BufferSize of 50-100 may provide the best performance. It is best to tune the BufferSize after configuring the Object Server triggers, and whilst the Object Server is being tested under expected peak loads.

If the BufferSize is oversized then the 'ObjectServer is busy. Waiting for locks to be released' messages will be observed in the client log files.

4.4 Impact on gateways

When there are a large number of events being processed by the Object Server gateways the following properties may need to be updated, depending upon the size of the source Object Server.

4.4.1 IpcTimeout

IpcTimeout should be set to 300 or 600 [5 or 10 minutes] in a multitier environment.

4.4.2 Ipc.StackSize

Ipc.StackSize affects the size of the selects allowed, but also affects the overall memory usage of the systems components. Ipc.StackSize must be set consistently throughout the Netcool/OMNIbus system; for Netcool/OMNIbus 8.1 the recommended Ipc.StackSize is 262144. Ipc.StackSize is set by default in the Object Server and gateways, but should be set explicitly if the system contains mixed Netcool/OMNIbus versions or if Ipc.StackSize is set to a value other than the default value. For non-Object Server gateways, like the JDBC Gateway, set Ipc.StackSize to the default value of the source Object Server, or the Ipc.StackSize set in the Object Servers property file.

4.4.3 Gate.CacheHashTblSize

The Gate.CacheHashTblSize property is used to set the size [number of rows] stored in the gateways cache. By default it is expected only 5,000 rows need to be stored in the gateways cache. When there are more rows being managed by the gateway, it will be advantageous to increase this setting to a value that best reflects the number of active rows in the source Object Server.

4.5 Integrations gateways synchronisation properties

The integrations gateways, such as the JDBC Gateway, have a number of properties that can be tuned to reduce the impact of having large volumes of source events. The best way to reduce loading, is to control the event flow using a filter and the After-IDUC features. In this way, only events not already processed will be forwarded.

Example for the JDBC Gateway:
Prevent older events being forwarded.

File : G_JDBC.props

```
Gate.Jdbc.ResyncFilter: 'LASTOCCURRENCE > (sysdate - 1)'
```

File : jdbc.rdrwtr.tblrep.def

```
REPLICATE ALL FROM TABLE 'alerts.status'
USING MAP 'StatusMap'
FILTER WITH 'LastOccurrence > (gatedate - 3600)'
;
```

You can use a custom flag to control the event flow, for example ReportGWFlag has to be set to forward the events, and afterwards is set to a value, so that the event can be deleted, if required.

File : jdbc.rdrwtr.tblrep.def

```
REPLICATE ALL FROM TABLE 'alerts.status'
USING MAP 'StatusMap'
FILTER WITH 'ReportGWFlag>1'
AFTER IDUC DO 'ReportGWFlag=3'
;
```