# Guidewire PolicyCenter™

## Database Upgrade Guide

Release 10.0.0

GUIDEWIRE
Adapt and succeed™

Product Name: Guidewire PolicyCenter
Product Release: 10.0.0
Document Name: Database Upgrade Guide
Document Revision: 26-September-2018

# Contents

# About PolicyCenter documentation

The following table lists the documents in PolicyCenter documentation:

| Document | Purpose |
|---|---|
| *InsuranceSuite Guide* | If you are new to Guidewire InsuranceSuite applications, read the *InsuranceSuite Guide* for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications. |
| *Application Guide* | If you are new to PolicyCenter or want to understand a feature, read the *Application Guide*. This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with PolicyCenter. |
| *Database Upgrade Guide* | Describes the overall PolicyCenter upgrade process, and describes how to upgrade your PolicyCenter database from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing PolicyCenter application extensions and integrations. |
| *Configuration Upgrade Guide* | Describes the overall PolicyCenter upgrade process, and describes how to upgrade your PolicyCenter configuration from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing PolicyCenter application extensions and integrations. The *Configuration Upgrade Guide* is published with the Upgrade Tools and is available from the Guidewire Community. |
| *New and Changed Guide* | Describes new features and changes from prior PolicyCenter versions. Intended readers are business users and system administrators who want an overview of new features and changes to features. Consult the “Release Notes Archive” part of this document for changes in prior maintenance releases. |
| *Installation Guide* | Describes how to install PolicyCenter. The intended readers are everyone who installs the application for development or for production. |
| *System Administration Guide* | Describes how to manage a PolicyCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring. |
| *Configuration Guide* | The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files for PolicyCenter. The intended readers are all IT staff and configuration engineers. |
| *PCF Reference Guide* | Describes PolicyCenter PCF widgets and attributes. The intended readers are configuration engineers. |
| *Data Dictionary* | Describes the PolicyCenter data model, including configuration extensions. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers. |
| *Security Dictionary* | Describes all security permissions, roles, and the relationships among them. The dictionary can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers. |
| *Globalization Guide* | Describes how to configure PolicyCenter for a global environment. Covers globalization topics such as global regions, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who localize PolicyCenter. |
| *Rules Guide* | Describes business rule methodology and the rule sets in Guidewire Studio for PolicyCenter. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu. |
| *Contact Management Guide* | Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are PolicyCenter implementation engineers and ContactManager administrators. |

| Document | Purpose |
|---|---|
| *Best Practices Guide* | A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers. |
| *Integration Guide* | Describes the integration architecture, concepts, and procedures for integrating PolicyCenter with external systems and extending application behavior with custom programming code. The intended readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java. |
| *Java API Reference* | Javadoc-style reference of PolicyCenter Java plugin interfaces, entity fields, and other utility classes. The intended readers are system architects and integration programmers. |
| *Gosu Reference Guide* | Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration. |
| *Gosu API Reference* | Javadoc-style reference of PolicyCenter Gosu classes and properties. The reference can be generated at any time to reflect the current PolicyCenter configuration. The intended readers are configuration engineers, system architects, and integration programmers. |
| *Glossary* | Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications. |
| *Product Model Guide* | Describes the PolicyCenter product model. The intended readers are business analysts and implementation engineers who use PolicyCenter or Product Designer. To customize the product model, see the *Product Designer Guide*. |
| *Product Designer Guide* | Describes how to use Product Designer to configure lines of business. The intended readers are business analysts and implementation engineers who customize the product model and design new lines of business. |

## Conventions in this document

| Text style | Meaning | Examples |
|---|---|---|
| *italic* | Indicates a term that is being defined, added emphasis, and book titles. In monospace text, italics indicate a variable to be replaced. | A *destination* sends messages to an external system.<br>Navigate to the PolicyCenter installation directory by running the following command:<br>`cd installDir` |
| **bold** | Highlights important sections of code in examples. | `for (i=0, i<someArray.length(), i++) {`<br>`  newArray[i] = someArray[i].getName()`<br>`}` |
| **narrow bold** | The name of a user interface element, such as a button name, a menu item name, or a tab name. | Click **Submit**. |
| `monospace` | Code examples, computer output, class and method names, URLs, parameter names, string literals, and other objects that might appear in programming code. | The `getName` method of the `IDoStuff` API returns the name of the object. |
| *`monospace italic`* | Variable placeholder text within code examples, command examples, file paths, and URLs. | Run the `startServer` *`server_name`* command.<br>Navigate to `http://`*`server_name`*`/index.html`. |

## Support

For assistance, visit the Guidewire Community.

### Guidewire customers

`https://community.guidewire.com`

### Guidewire partners

`https://partner.guidewire.com`

part 1

# Upgrade overview

This part describes strategies that you can consider for upgrade, provides a high-level overview of the upgrade process and lists versions of PolicyCenter that you can upgrade.

chapter 1

# Upgrade strategies

This topic explains the difference between a rolling upgrade and a full upgrade and presents the strategy of a mid-implementation upgrade.

## Versions supported by upgrade

You can upgrade to PolicyCenter 10.0.0 from the following versions:

- 9.0.x
- 8.0.1 or later

### Versions prior to 8.0

You cannot upgrade to PolicyCenter 10.0.0 directly from a PolicyCenter 7.0 or previous release. You must first upgrade your data model and database to the most recent PolicyCenter 9.0 release. Follow the process outlined in “Upgrade from 7.0 or previous versions” on page 111.

### Database

You can only upgrade an Oracle or SQL Server database. Guidewire does not support upgrade of the H2 Quickstart development database.

## Differences between a rolling upgrade and a full upgrade

This document describes how to do full upgrade. Rolling upgrades are addressed separately in the *System Administration Guide*. The following short overview explains the differences between these two types of upgrades.

Guidewire provides two different ways to modify the configuration of a PolicyCenter server in a cluster:

- Full upgrade
- Rolling upgrade

Each of these upgrade types has associated advantages and disadvantages. You use each upgrade type in different circumstances.

### Full application and database upgrade

A full application upgrade has the following functionality:

- Supports either a single or multiple server configuration and upgrade.
- Supports making significant changes to both the PolicyCenter application and database.
- Requires that you stop the entire PolicyCenter production installation for a period of time.
- Requires a significant amount of testing before and after complex changes.

It is possible to start a full upgrade from any server instance in the PolicyCenter cluster, from the Server Tools **Upgrade and Versions** screen.

### Rolling upgrade of a cluster member

A rolling upgrade of the individual server members in the cluster has the following functionality:

- Supports configuration deployment to multiple server instances.
- Successively targets a single cluster member for configuration deployment while leaving the remaining cluster members available for the processing of PolicyCenter operations.
- Supports a limited set of configuration changes.
- Requires that the new target configuration be compatible with the existing source configuration.
- Supports a command-line utility that checks the compatibility of the source and target configurations before deployment.
- Supports configuration deployment management and tracking from within PolicyCenter.

It is possible to start a rolling upgrade from any server instance in the PolicyCenter cluster, from the Server Tools **Upgrade and Versions** screen.

## Upgrades prior to initial production deployment

If a new major version is released during an initial implementation project, it is common to perform a mid-implementation upgrade. By enabling the implementation to go live on the more current version, this type of upgrade has many advantages, such as:

- Mid-implementation upgrades take less time and have less risk than production upgrades. There is no need for regression testing against a copy of a production database, and there is also no need for a production database upgrade. The project team can leverage the current implementation’s Stabilization Phase for regression testing of the upgrade work.
- If the initial development is still in progress, there is less code to upgrade.
- The deployment will benefit from a longer standard support period with the latest Guidewire product and required third party database and application server versions.
- The project can leverage new business and technical features from the latest version.

Guidewire strongly recommends that you consider including a major version upgrade as part of their initial implementation if the timing of the major release aligns with the implementation project. Your Guidewire project manager can provide an estimate for your mid-implementation upgrade based on how far along you are in your implementation.

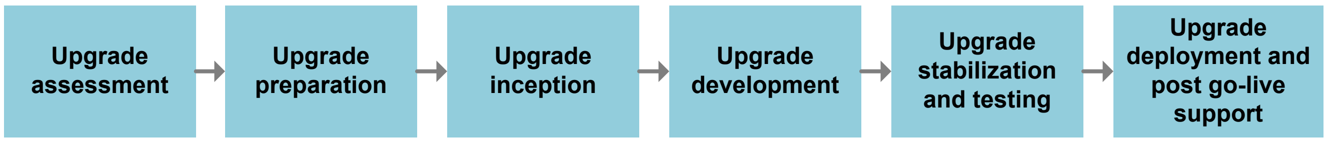
chapter 2

# Upgrade process

Before you begin an upgrade, plan and prepare for how an upgrade impacts both the business processes that rely on your Guidewire product and your organization as a whole. An upgrade requires commitment of time, personnel, and coordination among departments within your company.

**Note:** Guidewire recommends that you consult with Guidewire Services before beginning an upgrade. Guidewire Services has a number of Knowledge Base articles and Accelerators that might assist with your upgrade.

Include these phases while defining your upgrade project:


Upgrade assessment
Upgrade preparation
Upgrade inception
Upgrade development
Upgrade stabilization and testing
Upgrade deployment and post go-live support

- "Upgrade assessment" on page 13 – to plan the upgrade project.
- "Upgrade preparation" on page 15 – to prepare the environment and people for the upgrade project.
- "Upgrade inception" on page 16 – to define the upgrade project.
- – to implement the changes defined in the scope for the upgrade project.
- "Upgrade stabilization and testing" on page 17 – to perform system, performance and regression testing for the upgrade, as well as perform deployment dry runs.
- "Upgrade deployment and post go-live support" on page 17 – to migrate the upgrade to production and provide necessary post-implementation support during the first few weeks after deployment.

## Upgrade assessment

At the beginning, you need to determine the impact of the upgrade on your implementation. During this phase, you:

- Perform an opportunity assessment
- Analyze the impact on your installation
- Create project estimates

These steps provide your users with business benefits and provide a clear understanding of resource requirements and the schedule you need to complete the project.

This phase typically takes one to two weeks.

## Performing an opportunity assessment

The *New and Changed Guide* and the release notes describe new features available after upgrading PolicyCenter. As you review the new features, consider:

- How you might leverage these features into business benefits.
- How these features might help you improve your business processes.

Guidewire can also provide you with an evaluation copy of the new release, demonstrate new features for you, and help you understand opportunities these features can create for your business. Install the target version in a test environment. Take time to use the product to discover how to take advantage of the new features. Then, run some common scenarios for your company.

## Analyzing the impact on your installation

An upgrade might also impact your existing infrastructure, application configuration, and integration to other software.

### Infrastructure impact

You might find it desirable or necessary to upgrade your hardware or software. If a major infrastructure element changes, such as a database version, factor that into your upgrade planning.

Before starting the upgrade, make sure you have the supported server operating system, application server and database software, JDK, and client operating systems for the target version.

See the *Supported Software Components* knowledge article for current system and patch level requirements.

To access the knowledge article, search for *Supported Software Components* at the Guidewire Community:

**Guidewire Customers**

- `https://community.guidewire.com`

**Guidewire Partners**

- `https://partner.guidewire.com`

Complete any infrastructure updates for your upgrade development environment before you begin the upgrade.

### Configuration impact

Your company has uniquely configured PolicyCenter to meet its particular business needs. Configuration upgrade from the current version to the target version is your responsibility. Guidewire attempts to assist in this process by providing tools that locate and report back unique configurations in an installation. Additionally, The *New and Changed Guide* and the release notes provide a reference for changes between each version and how these changes impact an installation.

The time required to upgrade your configuration depends on the complexity of the installation and resources available to perform the upgrade. If you choose to deploy new features in a release, implementing them can also impact the upgrade duration.

Pay special attention to areas in which your installation has significant custom configurations. For example, if you have extensively configured a user interface page, review its related files, data objects, and upgrade documentation for specific changes. Guidewire documentation is searchable. Search for key words or attributes to see if they changed between releases. As part of your review, consider the following questions:

- Is there current functionality your company uses that is absent from the target version?
- Is there new functionality in the target version that your company will use?
- Do you need to customize the new functionality or is it adequate as provided?
- If you have existing integrations, how are they impacted?
- Are there data changes between the current version and the target version?

Guidewire recommends first running the automated upgrade procedure in a development environment. This can help identify areas requiring work and give an indication of how much work is involved. These areas include:

**Data Model** – The PolicyCenter data model includes all PolicyCenter data entities and their relationships. The base data model changes between versions. If you have added extensions to objects in the base data model, the upgrade procedure migrates your extensions. However, if you have rules or integration code that reference your extensions, you are responsible for ensuring that they are correct after the upgrade.

**User Interface** – The automated process updates user interface customizations if possible. The update tool reports back which files it could not change. You are responsible for identifying unique customizations and migrating them into the new interface.

**Other Configuration Files** – In addition to user interface configurations, your installation probably includes unique system settings, security settings, and other configuration file settings. Generally, the upgrade preserves customizations to these files. However, changes in these files could require you to migrate to new configuration files manually. In addition, all icons and `.gif` files must be reapplied and references to them in PCF files redone. This is not done by the upgrade tool.

**Rules** – Your installation includes rules that govern its behavior at critical decision points. Your rules must reference only objects in the upgraded data model. Manual changes to rules are the only way to ensure correct references. Review your rules to have a thorough understanding of their configuration and to identify and prepare for any technical debt that you will need to handle.

**Gosu** – Between versions, Guidewire deprecates or deletes some Gosu (formerly GScript) functions. You cannot use removed functions. Manually remove deleted functions. Remove references to deprecated functions.

#### Integration impact

Your PolicyCenter implementation supports many integrations with external systems. An upgrade requires that you manually update or rewrite these integrations. Guidewire provides tools and sample integrations for newer releases. Parts of this document that describe upgrading from a prior major version include a topic about upgrading integrations and Gosu from the prior major version. The *New and Changed Guide* and *Integration Guide* can also help you estimate the changes required for each integration.

### Creating project estimates

After completing the opportunity assessment and determining the impact of product changes on your implementation, you are able to define the scope of your upgrade project. Based on that scope you can identify some options for project staffing and schedule. If required, you can also produce a cost-benefit analysis for your upgrade project based on the scope and options defined.

## Upgrade preparation

Preparation work can be performed prior to or in parallel with the upgrade inception phase. Your main task is writing an upgrade specification. Other tasks in this phase include:

- Review results of the upgrade assessment and agree upon upgrade project scope.
- Review product documentation.
- Correct issues with database consistency checks.
- Ensure that your implementation documentation is up to date.
- Evaluate the skills and availability of internal resources.
- Identify and assign internal resources.
- Schedule and participate in training about changes and new features in the new release.
- Review and identify required changes to test scripts for the upgrade implementation.
- Acquire additional hardware and software to support infrastructure changes, if necessary.

Depending on your organization and implementation, this phase takes two to eight weeks.

### Writing an upgrade specification

Write an upgrade specification document that details the schedule, people, and equipment you expect to use during the upgrade. While writing the upgrade specification, answer the following questions.

### Schedule and people

- How does your company coordinate the production environment upgrade?
- Does your staff have the appropriate knowledge of PolicyCenter, or is additional training required?
- What external support, if any, is required to execute the upgrade?
- If you are using external support, how are responsibilities divided among the team members?
- Do you need to train your personnel on any aspects of the new system?
- Who supports the new configuration if users have questions?
- What type of documentation do you need for your new configuration?

### Equipment

- Does the upgrade to PolicyCenter 10.0.0 require software or hardware that your company must purchase? What is the expected lead time for a purchase at your company?
- Is the development and test infrastructure in place to ensure the new configuration meets company standards?
- Which old configurations are you upgrading?
- Which new features do you need to configure?
- Which integrations are impacted and to what extent?
- If something goes wrong during the upgrade of the production environment, how do you recover?
- How does the company support production fixes while executing the upgrade initiative?

To write an upgrade specification, take into account the testing and quality assurance your company requires. For each configuration upgrade and new feature in the configuration, define how that particular configuration will be tested and what criteria it must meet for acceptance.

## Upgrade inception

During this phase, you:

- Finalize the project scope, resources and schedule.
- Ensure any required training has been performed by Guidewire Education.
- Make any necessary adjustments to assigned resources.
- Prepare the upgrade project plan.
- Hold workshops to review product functionality.
- Review the documented Guidewire upgrade steps and adapt them to your project.
- Perform the upgrade project kick-off.

This phase typically takes one to two weeks.

## Upgrade stabilization and testing

Guidewire strongly recommends that you spend significant time testing your unique PolicyCenter configuration. Perform testing in a development environment, not a production environment. Follow the test plan you created in the planning phase. During this phase, you:

- Perform system testing, including performance and stress testing. Application hardware configurations such as clustering, load balancing, and email servers must work as expected.
- Test your entire configuration: user interface, data model and extensions, business model and the rules that implement it, and integrations to external systems.
- Perform several dry-runs of the database upgrade against a current copy of the production database. Test upgrading a copy of production data as early as possible when the upgrade includes LOB typelist changes. This provides time to address issues that might arise with production data that would not otherwise be detected.
- Perform user acceptance testing.
- Finalize training materials and deliver training.
- Document step-by-step tasks and projected schedule for deployment weekend.
- Prepare the production environment for deployment.
- Finalize plans for post go-live support.

Depending on the complexity of your installation, this phase takes two to three months.

## Upgrade deployment and post go-live support

The last part of the process is deployment of the upgraded implementation into the production environment. During this phase, you:

- Deploy the new configuration into production.
- Upgrade the production database.
- Verify the upgrade was successful.
- Provide heightened support.

During the deployment, coordinate within your company to ensure minimum interruption to business and sufficient time to qualify the new configuration in the production environment. Guidewire recommends that you perform the deployment over a weekend, with one to two weeks allocated for post go-live support.

### Sample deployment plan

This topic includes a sample deployment plan for the upgrade of ContactManager and PolicyCenter.

#### ContactManager prerequisite steps

##### Procedure

1. Complete the entire configuration upgrade (code merge) and all functional and non-functional testing against an upgraded copy of production.
2. During the configuration upgrade, preserve `MatchSetKey` column data if you want to keep it. This only applies to upgrades from 7.0 versions prior to 7.0.6.
3. Confirm that ContactManager 9.0 Studio shows no compilation errors, and ideally no warnings.
4. Confirm that all components of the upgraded infrastructure will be on a supported release, per the Platform Support Matrix.
5. In production, validate the database schema using the `system_tools -verifydbschema`. Ideally, there will be no issues reported.
6. In production, check the database consistency from the **Server Tools→Info Pages→Consistency Checks** page. Resolve any errors that are reported.
7. In production, generate a data distribution report. Confirm that the report can be generated properly.

8. In production, if database statistics are not already current, update database statistics shortly before the upgrade deployment.
9. For SQL Server: Decide whether to enable migration to 64-bit IDs during or after the upgrade.
10. For SQL Server: Confirm that the collation setting in the database matches the setting defined in `collations.xml` in the upgraded code base.
11. Prepare all infrastructure, including the database and application servers, load balancer, and so forth.
12. Preserve upgrade instrumentation from the most recent upgrade, if you want to keep it. This information can be downloaded from the **Upgrade Info** page.

## ContactManager deployment steps

### About this task

Perform steps 1-11 in the current production environment.

### Procedure

1. Confirm all batch processing has completed. Ensure no batch process or work queue has a **Status** of **Active** on the **Server Tools** > **Batch Process Info** page.
2. Suspend all message destinations from the **Administration** > **Event Messages** page.
3. Purge completed inactive messages. Note: You can also purge some in advance of the deployment window.
4. Purge completed workflows. Note: You can also purge some in advance of deployment window.
5. Purge completed workflow logs. Note: You can also purge some in advance of deployment window.
6. Validate the database schema using the `system_tools -verifydbschema` command. Ideally, there will be no issues reported. Drop unused columns if necessary, either before or after the upgrade.
7. Check the database consistency from the **Server Tools→Info Pages→Consistency Checks** page. Ensure no new errors are reported.
8. Generate a data distribution report.
9. Shut down production application servers.
10. Create a database backup.
11. Confirm adequate disk space for the database during upgrade. Allot at least 150% of the current production database size.
12. Disable database replication.
13. For Oracle: Assign default tablespace.
14. For Oracle: Optionally disable logging, statistics update, and statistics update for tables with locked statistics.
15. For SQL Server: Optionally disable SQL Server logging.
16. Move the database from old RDBMS release to new RDBMS release. This step applies to major version upgrades only.
17. For SQL Server: For major version upgrades, set the compatibility level to the database version supported by PolicyCenter 10.0.0.

    For example:

    ```
    ALTER DATABASE dbname SET COMPATIBILITY_LEVEL = 140
    ```
18. Upgrade application servers or update the production URL to point to the new application servers as appropriate. This step applies to major version upgrades only.
19. Disable the scheduler in `config.xml`.
20. Enable the database upgrade in `database-config.xml`.
21. Deploy WAR with upgraded configuration to application servers.
22. Start the server to begin database upgrade.
23. Review server log for unexpected errors. Search for the string `ERROR` in the log.
24. Perform initial high-level, view-only testing to ensure system availability.

25. Validate the database schema using the `system_tools -verifydbschema` command. Ideally, there will be no issues reported.
26. Check the database consistency from the **Server Tools→Info Pages→Consistency Checks** page.Compare the results with the results from the prerequisite steps.
27. Generate a data distribution report. Compare it against the data distribution report generated during“ContactManager deployment steps” on page 18.
28. Execute custom data validation scripts.
29. Run the Deferred Upgrade Tasks batch process if needed.
30. Run Phone Number Normalizer. Run Phone Number Normalizer on ContactManager first. This step applies to major version upgrades only.
31. Stop the server.
32. Create a database backup.
33. Enable the scheduler in `config.xml`.
34. Disable the database upgrade in `database-config.xml`.
35. Deploy new WAR.
36. Start the server. The upgrade is now completed.
37. Update database statistics by generating the statistics updating statements and executing them.
38. Perform initial testing of key application functionality.
39. Send announcement to user community.
40. For SQL Server: Migrate to 64-bit IDs if not done as part of initial upgrade.

## PolicyCenter prerequisite steps

### Procedure

1. Complete the entire configuration upgrade (code merge) and all functional and non-functional testing against an upgraded copy of production.
2. Confirm that PolicyCenter 9.0 Studio shows no compilation errors, and ideally no warnings.
3. Confirm that all components of the upgraded infrastructure will be on a supported release, per the Platform Support Matrix.
4. Confirm that user workstations will have a supported browser and, if appropriate, be able to work with documents.
5. In production, validate the database schema using the `system_tools -verifydbschema`. Ideally, there will be no issues reported.
6. In production, check the database consistency from the **Server Tools→Info Pages→Consistency Checks** page. Resolve any errors that are reported.
7. In production, generate a data distribution report. Confirm that the report can be generated properly.
8. In production, if database statistics are not already current, update database statistics shortly before the upgrade deployment.
9. For SQL Server: Decide whether to enable migration to 64-bit IDs during or after the upgrade.
10. For SQL Server: Confirm that the collation setting in the database matches the setting defined in `collations.xml` in the upgraded code base.
11. Prepare all infrastructure, including the database and application servers, load balancer, and so forth.
12. Preserve upgrade instrumentation from the most recent upgrade, if you want to keep it. This information can be downloaded from the **Upgrade Info** page.

## PolicyCenter deployment steps

### About this task

Perform steps 1-11 in the current production environment.

## Procedure

1. Confirm all batch processing has completed. Ensure no batch process or work queue has a **Status** of **Active** on the **Server Tools** > **Batch Process Info** page.
2. Suspend all message destinations from the **Administration** > **Event Messages** page.
3. Purge completed inactive messages. Note: You can also purge some in advance of the deployment window.
4. Purge completed workflows. Note: You can also purge some in advance of deployment window.
5. Purge completed workflow logs. Note: You can also purge some in advance of deployment window.
6. Set `pc_Coverage.CoverageSymbolGroup` to `null` for all records where it is not null. Preserve the existing value if you want.
7. If using Guidewire rating data, reload rating sample data.
8. Validate the database schema using the `system_tools -verifydbschema` command. Ideally, there will be no issues reported. Drop unused columns if necessary, either before or after the upgrade.
9. Check the database consistency from the **Server Tools→Info Pages→Consistency Checks** page. Ensure no new errors are reported.
10. Generate a data distribution report.
11. Shut down production application servers.
12. Create a database backup.
13. Confirm adequate disk space for the database during upgrade. Allot at least 150% of the current production database size.
14. Disable database replication.
15. For Oracle: Assign default tablespace.
16. For Oracle: Optionally disable logging, statistics update, and statistics update for tables with locked statistics.
17. For SQL Server: Optionally disable SQL Server logging.
18. Move the database from old RDBMS release to new RDBMS release. This step applies to major version upgrades only.
19. For SQL Server: For major version upgrades, set the compatibility level to the database version supported by PolicyCenter 10.0.0.

    For example:

    ```
    ALTER DATABASE dbname SET COMPATIBILITY_LEVEL = 140
    ```
20. Upgrade application servers or repoint the production URL to new application servers as appropriate. This step applies to major version upgrades only.
21. Disable the scheduler in `config.xml`.
22. Enable the database upgrade in `database-config.xml`.
23. Deploy WAR with upgraded configuration to application servers.
24. Run SQL script to clean up unreferenced address records.
25. Start the server to begin database upgrade.
26. Review server log for unexpected errors. Search for the string `ERROR` in the log.
27. Perform initial high-level, view-only testing to ensure system availability.
28. If using Guidewire rating data, reload rating sample data.
29. Validate the database schema using the `system_tools -verifydbschema` command. Ideally, there will be no issues reported.
30. Check the database consistency from the **Server Tools→Info Pages→Consistency Checks** page.Compare the results with the results from the prerequisite steps.
31. Generate a data distribution report. Compare it against the data distribution report generated in “PolicyCenter deployment steps” on page 19.
32. Execute custom data validation scripts.
33. Run the Deferred Upgrade Tasks batch process if needed. Launch this process from the `maintenance_tools`.

34. Run Phone Number Normalizer. Run Phone Number Normalizer on ContactManager first. This step applies to major version upgrades only.
35. If using Solr for free-text search, run job to create or refresh the Solr index.
36. Stop the server.
37. Create a database backup.
38. Enable the scheduler in `config.xml`.
39. Disable the database upgrade in `database-config.xml`.
40. Deploy new WAR.
41. Start the server. The upgrade is now completed.
42. Update database statistics by generating the statistics updating statements and executing them.
43. Perform initial testing of key application functionality.
44. Send announcement to user community.
45. For SQL Server: Migrate to 64-bit IDs if not done as part of initial upgrade.

part 2

# Performing an upgrade

This part describes how to do full upgrade of PolicyCenter.

For more information about this phase of the upgrade process, see .

For a high-level overview of the complete upgrade process, see “Upgrade process” on page 13.

chapter 3

# Upgrading from 9.0

This part describes how to perform an upgrade from PolicyCenter 9.0 to 10.0.0.

If you are upgrading from PolicyCenter 8.0, see “Upgrading from 8.0” on page 67 instead.

If you are upgrading from PolicyCenter 7.0 or a previous version, see “Upgrading from previous versions” on page 111 instead.

This part includes the following topics:

- “Upgrading the PolicyCenter 9.0 configuration” on page 25
- “Upgrading the PolicyCenter 9.0 database” on page 26
- “Upgrading PolicyCenter from 9.0 for ContactManager” on page 64
- “Upgrading ContactManager from 9.0” on page 65

## Upgrading the PolicyCenter 9.0 configuration

To upgrade your PolicyCenter configuration, refer to the *PolicyCenter Configuration Upgrade Guide*, which is available separately, from the Guidewire Community, as follows.

### Download the InsuranceSuite upgrade tools and documentation

#### About this task

Guidewire does not include the InsuranceSuite Upgrade Tools in the PolicyCenter installation.

You can access the latest version of the InsuranceSuite Upgrade Tools and the *PolicyCenter* Configuration Upgrade Guide using the following instructions.

#### Procedure

1. Visit the Guidewire Community:
   - Guidewire Customers - `https://community.guidewire.com`
   - Guidewire Partners - `https://partner.guidewire.com`
2. Select the **Resources** tab.
3. Under **Product Group**, select **InsuranceSuite**.
4. Under **Product**, select **InsuranceSuite Upgrade Tools**.
5. Under **Release**, select the most recent release.
6. Download the software, release notes, and documentation by clicking each corresponding download link.

# Upgrading the PolicyCenter 9.0 database

This topic provides instructions for upgrading the PolicyCenter 9.0 database to PolicyCenter 10.0.0.

You can only upgrade an Oracle or SQL Server database. Guidewire does not support upgrade of the H2 Quickstart development database.

## Upgrade administration data for testing

### About this task

You might want to create an upgraded administration data set for development and testing of rules and libraries with PolicyCenter 10.0.0. You can wait until the full database upgrade is complete and then export the administration data, as described in "Export administration data for testing" on page 62. Or, you can upgrade only the administration data to have this data available earlier in the upgrade process. Use the procedure in this section to create an upgraded administration data set before upgrading the full database.

### Procedure

1. Export administration data from your current (pre-upgrade) PolicyCenter production instance:
   a. Log on to PolicyCenter as a user with the `viewadmin` and `soapadmin` permissions.
   b. Click the **Administration** tab.
   c. Choose **Import/Export Data**.
   d. Select the **Export** tab.
   e. Select **Admin** from the **Data to Export** dropdown.
   f. Click **Export**. PolicyCenter exports an `admin.xml` file.
2. In a new base version development environment based on your production configuration, create empty files in the in the `modules/configuration/config/import/gen` directory:
   a. Create an empty version of `importfiles.txt`
   b. Create empty versions of the following CSV files:
   - `activity-patterns.csv`
   - `authority-limits.csv`
   - `reportgroups.csv`
   - `roleprivileges.csv`
   - `rolereportprivileges.csv`
     - Leave `roles.csv` as the original complete file.
3. Start the development environment server by opening a command prompt to `PolicyCenter` and entering the following command:

```
gwb runServer
```

4. Import this administration data into the development environment.
   a. Log on to PolicyCenter as a user with the `viewadmin` and `soapadmin` permissions.
   b. Click the **Administration** tab.
   c. Choose **Import/Export Data**.
   d. Select the **Import** tab.
   e. Click **Browse...**.
   f. Select the `admin.xml` file that you exported in "Upgrade administration data for testing" on page 26.
   g. Click **Open**.
5. Create a backup of the new development environment database.

6. Create a new database account for the development environment on a database management system supported by PolicyCenter 10.0.0.
   - See the *Supported Software Components* knowledge article for current system and patch level requirements. Visit the Guidewire Community and search for the *Supported Software Components* knowledge article.
   - See the *Installation Guide* for instructions to configure the database account.
7. Restore the backup of the database containing the imported administration data into the new database.
8. Connect your upgraded target PolicyCenter 10.0.0 configuration to the restored database.
9. Start the PolicyCenter 10.0.0 server to upgrade the database.
10. Export the upgraded administration data:
    a. Start the PolicyCenter 10.0.0 server by navigating to `PolicyCenter` and running the following command:
    ```
    gwb runServer
    ```
    b. Open a browser to PolicyCenter 10.0.0.
    c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
    d. Click the **Administration** tab.
    e. Choose **Import/Export Data**.
    f. Select the **Export** tab.
    g. For **Data to Export**, select **Admin**.
    h. Click **Export**.
    Your browser will note that you are opening a file and will prompt you to save or download the file.
    i. Download the `admin.xml` file.
    You can import this XML file into local development environments of PolicyCenter 10.0.0.

## Identify data model issues

### About this task

Before you upgrade a production database, identify issues with the data model by running the database upgrade on an empty database. This process does not identify all possible issues. The database upgrade does not detect issues caused by specific data in your production database. Instead, this procedure identifies issues with the data model.

Complete the following procedure to identify data model issues, and correct any issues on an empty schema. Then, follow the full list of procedures in this topic to upgrade a production database. This list begins with "Verify batch process and work queue completion" on page 28 and finishes with "Final steps after the database upgrade is complete" on page 63.

### To identify data model issues

### Procedure

1. Create an empty schema of your starting version database. You can do this in a development environment by pointing the development PolicyCenter installation at an empty schema and starting the PolicyCenter server. See the *Installation Guide*.
2. Complete the configuration upgrade for data model files in your starting version, according to the instructions in "Upgrading the PolicyCenter 8.0 configuration" on page 67. You do not need to complete the merge process for all files.
3. Configure your upgraded development environment to point to the database account containing the empty schema of your old version. See the *Installation Guide*.
4. Start the PolicyCenter server in your upgraded development environment. The server performs the database upgrade to PolicyCenter 10.0.0. See "Starting the server to begin automatic database upgrade" on page 56.
5. Check for errors reported during the upgrade process. Resolve any issues before upgrading your production database. You can use the `IDatabaseUpgrade` plugin to run custom SQL before and after the database upgrade. For more information, see "Creating custom upgrade version checks and version triggers" on page 35.

## Verify batch process and work queue completion

### About this task

All batch processes and work queues must complete before beginning the upgrade. Check the status of batch processes and work queues in your current production environment.

### Procedure

1. Log in to PolicyCenter as superuser.
2. Press `Alt` + `Shift` + `T`. PolicyCenter displays the **Server Tools** tab.
3. Click **Batch Process Info**.
4. Select **Any** from the **Processes** drop-down filter.
5. Click **Refresh**.
6. Check the **Status** column for each batch process listed. This list also includes batch processes that are writers for distributed work queues.

   If any of the batch processes have a **Status** of **Active**, wait for the batch process to complete before continuing with the upgrade.

## Purging data prior to upgrade

Purging certain types of data from the database prior to upgrade can improve the performance of the database upgrade and PolicyCenter.

## Purge old messages from the database

### About this task

Purge completed inactive messages before upgrading the database. Doing so improves performance and reduces the complexity of the database upgrade.

You cannot resend old messages after the upgrade. This is because integrations change and the message payload might be different. It is important that messages that have failed or not yet been consumed finish prior to upgrading.

Use one the following methods from the base customer configuration to purge completed messages from the `pc_MessageHistory` table:

### To purge old messages

### Procedure

1. The `messaging_tools` command from the `admin/bin` directory of base customer configuration. This tool deletes completed messages with a send time before the date *MM*/*DD*/YY*YY*: `messaging_tools -password` *`password`* `-server http://`*`server`*`:`*`port`*`/`*`instance`* `-purge` *`MM/DD/YYYY`*
2. The `purgeCompletedMessages` web service API:
   `IMessageToolsAPI.purgeCompletedMessages(java.util.Calendar cutoff)`

### Next steps

After you purge completed inactive messages, reorganize the `pc_MessageHistory` table. You might also want to rebuild any indexes on the table. Contact Guidewire Support if you need assistance.

## Purging completed workflows and workflow logs

Each time PolicyCenter creates an activity, the activity is added to the `pc_Workflow`, `pc_WorkflowLog` and `pc_WorkflowWorkItem` tables. Once a user completes the activity, PolicyCenter sets the workflow status to completed. The `pc_Workflow`, `pc_WorkflowLog` and `pc_WorkflowWorkItem` table entry for the activity are never used again. These tables grow in size over time and can adversely affect performance as well as waste disk space. Excessive records in these tables also negatively impacts the performance of the database upgrade.

Remove workflows, workflow log entries, and workflow items for completed activities to improve database upgrade and operational performance and to recover disk space.

Use the `purgeworkflows` batch process to purge older completed workflows and their logs. Guidewire recommends that you purge completed workflows and their logs periodically. This reduces performance issues caused by having a large number of unused workflow log records.

Alternatively, you can purge only the logs associated with completed workflows by running the `purgeworkflowlogs` batch process. This batch process leaves the workflow records and removes only the workflow log records.

### Purge completed workflows and associated logs

#### Procedure

1. Set the number of days after which the `purgeworkflows` process purges completed workflows and their logs, by setting the `WorkflowPurgeDaysOld` parameter in `config.xml`. By default, `WorkflowPurgeDaysOld` is set to `60`. This is the number of days since the last update to the workflow, which is the completed date.

   ```
   <param name="WorkflowPurgeDaysOld" value="60" />
   ```

2. Launch the **Purge Workflows** batch process from the `PolicyCenter/admin/bin` directory with the following command:

   ```
   maintenance_tools -password password -startprocess PurgeWorkflows
   ```

### Purge completed workflow logs only

#### Procedure

1. Set the number of days after which the `purgeworkflowlogs` process purges completed workflow logs, by setting the `WorkflowLogPurgeDaysOld` parameter in `config.xml`. This is the number of days since the last update to the workflow, which is the completed date.

   ```
   <param name="WorkflowLogPurgeDaysOld" value="60" />
   ```

2. Launch the **Purge Workflow Logs** batch process from the `PolicyCenter/admin/bin` directory with the following command:

   ```
   maintenance_tools -password password -startprocess PurgeWorkflowLogs
   ```

## Validate the database schema

#### About this task

This validation detects the unlikely event that the data model defined by your configuration files has become out of sync with the database schema.

#### Procedure

1. While the pre-upgrade server is running, use the `system_tools` command in `admin/bin` of the customer configuration to verify the database schema:

   ```
   system_tools -password password -verifydbschema -server servername:port/instance
   ```

2. Correct any validation problems in the database before proceeding. Contact Guidewire Support for assistance.

#### Next steps

Following the database upgrade, run this command again from the `admin/bin` directory of the target (upgraded) configuration.

## Checking database consistency

PolicyCenter has hundreds of internal database consistency checks. Before upgrading, run consistency checks to verify the integrity of your data.

Run database consistency checks early in the upgrade project. Fix any consistency errors. Continue to periodically run consistency checks and resolve issues so that your database is ready to upgrade when you begin the upgrade procedure. Consistency issues might take some time to resolve, so begin the process of running consistency checks and fixing issues early. Contact Guidewire Support for information on how to resolve any consistency issues.

After the database upgrade, run consistency checks again from the PolicyCenter **Consistency Checks** page.

### Run database consistency checks

#### Procedure

1. Start the PolicyCenter server if it is not already running.
2. Log in to PolicyCenter with an administrator account.
3. Press `Alt` + `Shift` + `T` to access the **Server Tools**.
4. Click **Info Pages**.
5. Select **Consistency Checks** from the drop-down list.
6. To increase the number of threads used to run consistency checks, increase the **Number of threads**. The number of threads to use depends on the capability of your database server. Increasing the number of threads can improve performance of consistency checks as long as your server can process the threads. Guidewire recommends starting with five threads. If too many threads are used, there is a greater chance that current users experience reduced performance if the database server is fully loaded.

   To set the number of threads in versions prior to 8.0, specify a value for the `checker.threads` parameter within the database block of `config.xml`.

   ```
   <database
     ...
     <param name="checker.threads" value="5" />
     ...
   </database>
   ```
7. Click **Run Consistency Checks**.

#### See also

*System Administration Guide*.

## Create a data distribution report

#### About this task

Generate a data distribution report for the database before an upgrade. Save the output of this report. Run the report again after the upgrade to ensure the distribution is still correct.

Guidewire is very interested in the data distribution of your databases. Guidewire uses these reports to better understand the nature of your database and to optimize PolicyCenter performance. Guidewire appreciate copies of your reports, both before and after upgrades.

You can also use this information to tune the application server cache. See the *System Administration Guide*.

#### To create a database distribution report

#### Procedure

1. In `config.xml`, set `<param name="EnableInternalDebugTools" value="true"/>`.
2. Start the PolicyCenter application server.
3. Log into PolicyCenter as an administrative user.
4. Type `ALT` + `SHIFT` + `T` while in any screen to reach the **Server Tools** page.
5. Choose **Info Pages** from the **Server Tools** tab.
6. Choose the **Data Distribution** page from the **Info Pages** dropdown.
7. Enter a reason for running the Data Distribution batch job in the **Description** field.
8. On this page, select the **Collect distributions for all tables** radio button and check all checkboxes to collect all distributions.
9. Push the **Submit Data Distribution Batch Job** button on this page to start the data collection.
10. Return to the **Data Distribution** page and push its **Refresh** button to see a list of all available reports. The batch job has completed when the **Available Data Distribution** list on the **Data Distribution** page includes your description.
11. Select the desired report and use the **Download** button to save it zipped to a text file. Unzip the file to view it.

## Generating database statistics

You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded PolicyCenter in a production environment.

To optimize the performance of the PolicyCenter database, it is a good idea to update database statistics on a regular basis. Both SQL Server and Oracle can use these statistics to optimize database queries.

If you update database statistics on a regular basis, you do not need to update statistics before an upgrade. If you do not update database statistics on a regular basis, Guidewire recommends that you update incremental statistics before running the upgrade.

### Generate incremental database statistics

#### Procedure

1. Get the proper SQL statements for updating the statistics in PolicyCenter tables by running the following command:

```
system_tools -password password -getincrementaldbstatisticsstatements -server
  http://server:port/instance > db_stats.sql
```

2. Run the resulting SQL statements against the PolicyCenter database.

#### Result

You can configure SQL Server to periodically update statistics using SQL. See your database documentation and the *System Administration Guide* for more information.

The database upgrade can take a long time, and has built-in statistics collection that help you see if any part of the upgrade is particularly slow. Collect these statistics, and compare them to the statistics you collected before the upgrade. The `config.xml` file has parameters that control this statistics collection.

### Generate full statistics after upgrade is complete

For Oracle, Guidewire recommends that you generate full statistics as soon as possible after the upgrade, during the next maintenance window. Do this even if you did not disable statistics collection during the upgrade by setting `updatestatistics` to `false`.

For SQL Server, this step is only necessary if you disabled statistics collection during the upgrade by setting `updatestatistics` to `false`. For this case, Guidewire recommends that you generate full statistics as soon as possible after the upgrade.

#### See also

*System Administration Guide*.

## Back up your database

### About this task

The first time you start the PolicyCenter server after running the upgrade tools, the server updates the database. During upgrade, PolicyCenter minimizes logging. For these reasons, your database might not be recoverable if the upgrade process is runs into problems.

Back up your database before starting an upgrade. Consult the documentation for your database management system for tools and techniques to back up the database.

## Update database infrastructure

### About this task

Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of PolicyCenter 10.0.0.

See the *Supported Software Components* knowledge article for current system and patch level requirements. Visit the Guidewire Community and search for the *Supported Software Components* knowledge article.

## Prepare the database for upgrade

### About this task

Do the following to prepare the database for the upgrade process:

### Procedure

1. **Ensure Adequate Free Space** – The database upgrade requires significant free space. Make sure the database has at least 50% of the current database size available as free space.
2. **Disable Replication** – Disable database replication during the database upgrade.
3. **Assign Default Tablespace (Oracle only)** – Set the default tablespace for the database user to the one mapped to the logical tablespace `OP` in `config.xml`.

   The database upgrade creates temporary tables during the upgrade without specifying the tablespace. If the Oracle database user was created without a default tablespace, Oracle by default creates the tables in the SYSTEM tablespace. The Guidewire database user is likely not to have the required quota permission on the SYSTEM tablespace. This results in an error of the type:

   java.sql.SQLException: ORA-01950: no privileges on tablespace 'SYSTEM'

   Even if the default tablespace is not SYSTEM, if the Guidewire database user does not have quota permission on the default tablespace, the temporary table creation during upgrade fails.
4. **Exclude tables from Change Data Capture (CDC) (SQL Server only)** – If Change Data Capture (CDC) is enabled, Guidewire recommends that you exclude some tables before upgrade.

   You can exclude the following tables from CDC:

   a. All entities with eti files that have `instrumentationtable="true"`
   b. `pc_batchprocesslease`
   c. `pc_batchprocessleasehistory`
   d. `pc_destinationlease`
   e. `pc_destinationleasehistory`
   f. `pc_messagerequestlease`
   g. `pc_messagerequestleasehistory`
   h. `pc_pluginlease`

   i. `pc_pluginleasehistory`

5. **Remove `adaptive-optimization` element from `database-config.xml` (Oracle only)**– If you applied Oracle patch 22652097, remove the `adaptive-optimization` element from `database-config.xml`.

   Keep the default `init.ora` adaptive parameters:
   - `optimizer_adaptive_statistics` (default `false`)
   - `optimizer_adaptive_plans` (default `true`)

## Set linguistic search collation

**WARNING** For SQL Server, compare the default collation of the database to the collation defined for your locale. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your SQL Server database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This attribute change results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, this adds time to the total database upgrade process.

**WARNING** Oracle Java Virtual Machine (JVM) must be installed on all Oracle databases hosting PolicyCenter. The only exception is when the PolicyCenter application locale is English and you only require case-insensitive searches. Ensure that Oracle initialization parameter `java_pool_size` is set to a value greater than 50 MB.

You can specify how you want PolicyCenter to collate search results. The `strength` attribute of the `LinguisticSearchCollation` element of `GWLanguage` in `language.xml` in the currently selected region specifies how PolicyCenter sorts search results. You can set the `strength` to `primary` or `secondary`.

With `LinguisticSearchCollation strength` set to `primary`, PolicyCenter searches results in a case-insensitive and accent-insensitive manner. PolicyCenter considers an accented character equal to the unaccented version of the character if the `LinguisticSearchStrength` for the default application locale is set to `primary`. For example, with `LinguisticSearchCollation strength` set to `primary`, PolicyCenter treats “Reneé”, “Renee”, “renee” and “reneé” the same.

With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter searches results in a case-insensitive, accent-sensitive manner. PolicyCenter does not consider an accented character equal to the unaccented version of the character if the `LinguisticSearchCollation strength` for the default application locale is set to `secondary`. For example, with `LinguisticSearchCollation strength` set to `secondary`, a PolicyCenter search treats “Renee” and “renee” the same but treats “Reneé” and “reneé” differently. By default, PolicyCenter uses a `LinguisticSearchCollation strength` of `secondary`, which specifies case-insensitive, accent-sensitive searching.

The `collations.xml` file defines the collations to use for different locales and different collation strengths. The `primary`, `secondary`, and `tertiary` attributes of the `Collation` element define the collation to use depending on the `LinguisticSearchCollation strength` attribute in `language.xml`.

PolicyCenter 7.0 introduced configurable linguistic searching for SQL Server databases. In releases prior to PolicyCenter 7.0, PolicyCenter used the collation setting of the database server. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This attribute change results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, dropping and recreating indexes adds time to the total database upgrade process.

For sorting search results, the following rules apply:

**Case**

All searches ignore the case of the letters, whether `LinguisticSearchCollation strength` is set to `primary` or `secondary`. "McGrath" equals "mcgrath".

**Punctuation**

Punctuation is always respected, and never ignored. “O'Reilly” does not equal “OReilly”.

**Spaces**

Spaces are respected. "Hui Ping" does not equal "HuiPing".

**Accents**

An accented character is considered equal to the unaccented version of the character if `LinguisticSearchCollation strength` is set to `primary`. An accented character is not equal to the unaccented version if `LinguisticSearchCollation strength` is set to `secondary`.

Japanese only:

**Half Width/Full Width**

Searches under a Japanese region always ignore this difference.

**Small/Large Kana**

Japanese small/large letter differences are ignored only when `LinguisticSearchCollation strength` is set to `primary`, meaning accent-insensitive.

**Katakana/Hiragana sensitivity**

Searches under a Japanese region always ignore this difference.

**Long dash character**

Always ignored.

**Soundmarks ( `` and ° )**

Only ignored if `LinguisticSearchCollation strength` is set to `primary`.

German only

**Vowels with an umlaut**

Compare equally to the same vowel followed by the letter e. Explicitly, “ä”, “ö”, “ü” are treated as equal to “ae”, “oe” and “ue”.

**The Eszett, or sharp-s, character "ß"**

Treated as equal to “ss”.

PolicyCenter populates denormalized values of searchable columns to support the search collation. For example, with `LinguisticSearchCollation strength` set to `primary`, PolicyCenter stores the value “Reneé”, “Renee”, “renee” and “reneé” in a denormalized column as “renee”. With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter stores a denormalized value of “renee” for “Renee” or “renee” and stores “reneé” for “Reneé” or “reneé”. Japanese and German locales make additional changes when storing values in denormalized columns in order to conform to the rules listed previously for those locales.

Any time you change the `LinguisticSearchCollation strength` and restart the server, PolicyCenter repopulates the denormalized columns. Previous versions of PolicyCenter populated the denormalized columns with lowercase values for case-insensitive search, equivalent to setting `LinguisticSearchCollation strength` to `secondary`. If you set `LinguisticSearchCollation strength` to `primary`, PolicyCenter repopulates the denormalized columns, substituting any accented characters for their base equivalents. This process can take a long time, depending on the amount of data. Therefore, if you want to change `LinguisticSearchCollation strength` to `primary`, you might want to do so after the database upgrade. If you are concerned about the duration of the database upgrade, you can change your search collation settings after the upgrade. During a maintenance period, change `LinguisticSearchCollation strength` to `primary` and restart the server to repopulate the denormalized columns.

For Japanese locales, the PolicyCenter database upgrade from a prior major version repopulates the denormalized columns regardless of the `LinguisticSearchCollation strength` value. PolicyCenter must repopulate the denormalized columns for Japanese locales to have search results obey the Japanese-only rules listed previously.

For more information, see in the *Globalization Guide*.

## Customizing the database upgrade

The `IDatamodelUpgrade` plugin interface provides hooks for custom code that you want to run during the database upgrade. You can implement the `IDatamodelUpgrade` plugin to:

- Execute custom version checks to test data or the data model itself before starting the upgrade.
- Make custom database changes before or after the database upgrade.
- Make data model changes to archived entities.

For example, you might fix a consistency check failure issue, correct issues reported by version checks, or delete a custom extension that you are no longer using.

**IMPORTANT** PolicyCenter 4.0 included a similar plugin interface, `IDatabaseUpgrade`. If you previously implemented `IDatabaseUpgrade` for an upgrade to PolicyCenter 4.0, you must now implement `IDatamodelUpgrade` if you want to execute custom upgrade code.

### Creating custom upgrade version checks and version triggers

Use the `IDatamodelUpgrade` plugin to run custom version checks and triggers before and after the database upgrade. The `IDatamodelUpgrade` plugin interface defines method signatures for two methods that you must define in your plugin. These signatures are:

- `property getBeforeUpgradeDatamodelChanges() : List<IDataModelChange<BeforeUpgradeVersionTrigger>>`
- `property getAfterUpgradeDatamodelChanges() : List<IDataModelChange<AfterUpgradeVersionTrigger>>`

Each method returns a list of `IDataModelChange` entities, each taking a `BeforeUpgradeVersionTrigger` or `AfterUpgradeVersionTrigger` type parameter. The `IDataModelChange` interface has two methods that you use to make data model changes. The `getDatabaseUpgradeVersionTrigger` method is for changes to the database. The `getArchivedDocumentUpgradeVersionTrigger` method is for changes to archived entities. If your organization has not implemented archiving or you do not want to make changes to archived entities, return null for `getArchivedDocumentUpgradeVersionTrigger`.

The `getAfterUpgradeDatamodelChanges` method runs after the Guidewire upgrade version triggers. You can use this method to move data into extension tables or columns that did not exist prior to upgrading.

You can return an empty list from either `getBeforeUpgradeDatamodelChanges` or `getAfterUpgradeDatamodelChanges`. For example, if you only have triggers to run before the upgrade, you can return an empty list from `getAfterUpgradeDatamodelChanges`.

### Modifying tables using a custom version trigger

Both `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` base classes provide a protected `getTable` method that accepts a `string` parameter. The `getTable` method returns an `IBeforeUpgradeTable` or `IAfterUpgradeTable` object that provides a number of methods for DDL and DML operations, such as:

- `create` – Create the table if it does not already exist. The table must be related to an entity defined in the data model. This method is available only for `IBeforeUpgradeTable`.
- `delete` – Deletes rows from a table. Returns a builder (`IBeforeUpgradeDeleteBuilder` for `IBeforeUpgradeTable`, `IDeleteBuilder` for `IAfterUpgradeTable`) that has methods for comparing data to restrict which rows are deleted.
- `drop` – Drops the table.
- `dropColumns` – Drops multiple columns from the table.
- `getColumn` – Returns an `IBeforeUpgradeColumn` or `IAfterUpgradeColumn` object that has methods to perform DDL operations on the column such as create, drop, rename, and more.
- `insert` – Returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable`) to perform an insert operation.
- `insertSelect` – Returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`) for SQL to perform an insert operation using data selected from a table.
- `rename` – Renames the table.
- `update` – Returns a builder (`IBeforeUpgradeUpdateBuilder` for `IBeforeUpgradeTable`, `IUpdateBuilder` for `IAfterUpgradeTable`) for SQL to perform an update operation.

For DML operations, call the `execute` method on the builder to actually perform the operation. The `execute` method runs in its own transaction. You do not need to handle transactions and `TransactionManager`.

There are more methods on the `IBeforeUpgradeTable` and `IAfterUpgradeTable` classes documented in the Guidewire Gosu API documentation. To generate the Guidewire Gosu API documentation, run the `gwb gosudo` command from the PolicyCenter installation directory. Then, open `PolicyCenter/build/gosudoc/index.html`.

The methods for `BeforeUpgradeVersionTrigger` intentionally take strings but not entities or properties. This is because the name of the column could change in the future. Consider `PropertyA` on `EntityE` which corresponds to column `A` in the database. Suppose you use `PROPERTYA_PROP` in a version trigger at minor version 200, but at minor version 250, you decide to rename the backing column from `A` to `B`. The version trigger you wrote in the past would break because it would execute before the rename operation and would try to use the new column name.

`AfterUpgradeVersionTrigger` is very similar to `BeforeUpgradeVersionTrigger`. A few differences include:

- The `AfterUpgradeVersionTrigger` DML builders use the query builder, `IQueryBuilder`.
- In an `AfterUpgradeVersionTrigger` you can use properties and types in addition to strings.
- Some DDL operations are not provided on the `IAfterUpgradeTable` object, including creating a table or adding a column.

  **Note:** Guidewire recommends you use `BeforeUpgradeVersionTrigger` for custom version triggers a unless you have a special case requiring one of the unique capabilities of `AfterUpgradeVersionTrigger`. `BeforeUpgradeVersionTrigger` has many more capabilities than `AfterUpgradeVersionTrigger`. One example of a limitation off `AfterUpgradeVersionTrigger` is that you cannot set `MonetaryAmount` fields from an `AfterUpgradeVersionTrigger`.

### Upgrading typelists using a custom version trigger

The `BeforeUpgradeVersionTrigger` class includes a `getTypeKeyID` method with the following signature:

```
protected final Integer getTypeKeyID(IEntityType subtype)
```

**Note:** Protected methods do not appear in the Gosu documentation. Use `CTRL` + `SPACE` in Studio to show available methods and properties.

The `getTypeKeyID` method returns the integer ID of the type code in the type list matching the given table name. This method checks both the existing typelist tables and the metadata files to determine what all typekey IDs will be after upgrade. Therefore, the `getTypeKeyID` method works as expected even before a new typekey or typelist table is created during the automatic schema upgrade phase.

This method also works for orphaned typecodes that have not yet been removed from the database. These are typecodes that still exist in the database table but not in the metadata file. You can use the `getTypeKeyID` method for remapping usages of orphaned typecodes.

### Changing the nullability of subtype columns after a database upgrade

For entity definitions, an automatic database upgrade converts nullable columns to non-nullable columns. For the converted columns, the upgrade process sets a default value successfully.

However, the automatic database upgrade process does not convert column nullability for columns in subtype definitions. Instead, PolicyCenter implements non-nullable columns on subtypes in the database as nullable columns. The product behaves this way because columns must have `null` values for rows that represent instances of other subtypes.

To make a subtype column non-nullable, write a custom version trigger. Program the version trigger to populate the column with an appropriate default value for existing subtype rows. After you perform this step, PolicyCenter makes new column values for the subtype non-nullable. The product does this by setting the default value for new subtype rows automatically.

See “Setting a column value for a specific subtype using a version trigger” on page 44 and “Create a custom version check or version trigger” on page 38 for more information on creating a version trigger in this manner.

### Version checks

You can check for a certain condition in the database before the upgrade proceeds. This is referred to as a version check. Only read operations are available in version checks. For example, you can implement a version check to query a table or check the existence of a table or column, but the check cannot insert new rows. The `BeforeUpgradeVersionTrigger` class includes a `hasVersionCheck` method that you must define to return true or false. If the trigger does include a version check, overwrite the `createVersionCheck` method to define your custom version check. For standalone version checks that are not associated with a version trigger, you can use `BeforeUpgradeVersionCheckWrapper`.

The upgrade executes all custom version checks before custom version triggers. The upgrade runs Guidewire version checks after all custom `BeforeUpgradeVersionTrigger` implementations, so you can create a `BeforeUpgradeVersionTrigger` to correct issues detected by the Guidewire version checks.

If a custom version check fails, the upgrade stops before running any upgrade triggers. Correct the issue and restart the upgrade.

### Order of execution

The following table describes failure cases that are caused by data issues. If the upgrade fails for other reasons, such as a disruption of the database server, fix the issue causing the disruption, restore the database, and restart the upgrade. During an upgrade, PolicyCenter performs database upgrade actions in the following order:

| Step | Action | In the event of failure due to a data issue... |
|---|---|---|
| 1 | Custom version checks | Correct the data issue. Restart the upgrade. You do not need to restore the database because the upgrade has not made any changes. |
| 2 | Custom `BeforeUpgradeVersionTrigger` implementations | Restore the database from a backup. Correct the data issue. Consider adding custom version checks to test for other instances of the data issue. |
| 3 | Guidewire version checks | If you do not have any custom `BeforeUpgradeVersionTrigger` implementations, correct the data issue and restart the upgrade.<br>If you do have custom `BeforeUpgradeVersionTrigger` implementations, restore the database from a backup. Then, correct the data issue.<br>In either case, consider creating a custom `BeforeUpgradeVersionTrigger` implementation to correct the data issue. |

| Step | Action | In the event of failure due to a data issue... |
|---|---|---|
| 4 | Guidewire version triggers | A failure due to data issues at this stage is unlikely. Contact Guidewire Support. |
| 5 | Automated data model upgrade to update the database to the defined data model. | A failure due to data issues at this stage is unlikely. Contact Guidewire Support. |
| 6 | Guidewire version triggers that require the updated data model in the database | A failure due to data issues at this stage is unlikely. Contact Guidewire Support. |
| 7 | Custom `AfterUpgradeVersionTrigger` implementations | Restore the database from a backup. Correct the data issue. Consider creating a custom `BeforeUpgradeVersionTrigger` implementation to correct the data issue if possible. |

## Create a custom version check or version trigger

### About this task

You can implement a `IDatamodelUpgrade` plugin with custom version triggers and version checks. When you start the server to perform the database upgrade from an earlier release, PolicyCenter calls the plugin and runs your custom methods.

Each `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` instance requires a minor version number, passed as an integer. If the data model version number is less than or equal to the number passed to the instance, then the trigger executes. Whenever you make a data model change, or you want to force an upgrade, increment the version number in `extensions.properties`.

### To create custom version checks and triggers

### Procedure

1. Create a new package, such as *`companyName`*`.upgrade`, to store your custom version triggers.
   a. Open Studio.
   b. In the Studio **Project** window, expand **configuration**.
   c. Right-click **gsrc** and click **New→Package**.
   d. Enter a package name for upgrade purposes, such as *`companyName`*`.upgrade`.
2. Right-click the upgrade package and click **New→Gosu Class**.
3. Enter a name for the class and click **OK**.
4. Create a new Gosu class that extends `CustomerDatamodelUpgrade` and implements `IDatamodelUpgrade`. The class you create must define the `getBeforeUpgradeDatamodelChanges` and `getAfterUpgradeDatamodelChanges` methods. This class is the container from which you call custom version trigger classes.

   For example:

```
package companyName.upgrade
uses gw.plugin.upgrade.IDatamodelUpgrade
uses java.lang.Iterable
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger
uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger
uses java.util.ArrayList
uses gw.api.datamodel.upgrade.CustomerDatamodelUpgrade
uses gw.api.datamodel.upgrade.IDatamodelChange
uses gw.api.database.upgrade.DatamodelChangeWithoutArchivedDocumentChange

 class TestDatamodelUpgradeImpl extends CustomerDatamodelUpgrade implements IDatamodelUpgrade {

   override property get BeforeUpgradeDatamodelChanges() :
  List<IDatamodelChange<BeforeUpgradeVersionTrigger>> {

     var list = new ArrayList<IDatamodelChange<BeforeUpgradeVersionTrigger>>()
    list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger1()))
    list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger2()))
    return list
  }

   override property get AfterUpgradeDatamodelChanges() :
  List<IDatamodelChange<AfterUpgradeVersionTrigger>> {
    var list = new ArrayList<IDatamodelChange<AfterUpgradeVersionTrigger>>()
    list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new AfterVersionTrigger1()))
    return list
  }
}
```

5. Create your custom `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` Gosu classes. See "IDatamodelUpgrade API examples" on page 39.
6. Implement the `IDatamodelUpgrade` plugin with the new class.
   a. Start Guidewire Studio 10.0.0 by entering `gwb studio` from the `PolicyCenter` directory.
   b. In Studio, expand **configuration→config→Plugins**.
   c. Right-click **registry** and click **New→Plugin**.
   d. In the **Plugin** dialog, enter the name `IDatamodelUpgrade`. For this plugin, the name must match the interface.
   e. In the **Plugin** dialog, click the ... button.
   f. In the **Select Plugin Class** dialog, type `IDatamodelUpgrade` and select the **IDatamodelUpgrade** interface.
   g. In the **Plugin** dialog, click OK. Studio creates a GWP file under **Plugins→registry** with the name you entered.
   h. Click the **Add Plugin** icon (a plus sign) and select **Add Gosu Plugin**.
   i. For **Gosu Class**, enter your class, including the package.
   j. Save your changes.

## IDatamodelUpgrade API examples

The `IDatamodelUpgrade` plugin interface provides hooks for custom code that you want to run during the database upgrade. You can implement the `IDatamodelUpgrade` plugin to:

- Execute custom version checks to test data or the data model itself before starting the upgrade.
- Make custom database changes before or after the database upgrade.
- Make data model changes to archived entities.

### See also

"Creating custom upgrade version checks and version triggers" on page 35

The following topics contain examples of how you can customize your database upgrade by implementing an `IDatamodelUpgrade` plugin:

- "BeforeUpgradeVersionTrigger Structure" on page 40
- "AfterUpgradeVersionTrigger Structure" on page 41
- "Altering columns to match the data model using a version trigger" on page 41
- "Alter a non-nullable column to nullable using a version trigger" on page 42
- "Creating columns using a version trigger" on page 42
- "Dropping columns using a version trigger" on page 43
- "Renaming columns using a version trigger" on page 44
- "Setting a column value for a specific subtype using a version trigger" on page 44
- "Creating tables using a version trigger" on page 44
- "Renaming tables using a version trigger" on page 45
- "Deleting rows using a version trigger" on page 45
- "Inserting rows using a version trigger" on page 45
- "Inserting data selected from another table using a version trigger" on page 46
- "Updating rows using a version trigger" on page 46

## BeforeUpgradeVersionTrigger Structure

A custom `BeforeUpgradeVersionTrigger` subclass has the following structure. Define the `execute` method to perform your custom trigger actions.

```
package companyName.upgrade.before

 uses gw.api.database.upgrade.before.BeforeUpgradeVersionCheck
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger

 class myBeforeUpgradeTrigger extends BeforeUpgradeVersionTrigger {

   construct() {
    super(dataModelVersionNumber)
  }

   override function execute() {
    // Perform actions here.
  }

   override function hasVersionCheck() : boolean {
    // return true if creating a version check to determine whether the trigger can run.
    // return false if you are not implementing a version check.
  }

   override property get Description() : String {
    return "Description of the version trigger."
  }

   // Override the createVersionCheck method if you are implementing a version check.
  override function createVersionCheck() : BeforeUpgradeVersionCheck {
    return new BeforeUpgradeVersionCheck(dataModelVersionNumber) {

       override function verifyUpgradability() {
        if (condition to detect) {
          addVersionCheckProblem("description of issue")
        }
      }

       override property get Description() : String {
        return "Description of the version check."
      }
    }
  }
}
```

### AfterUpgradeVersionTrigger Structure

A custom `AfterUpgradeVersionTrigger` subclass has the following structure.

```
package companyName.upgrade.after

 uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger

 class myAfterUpgradeTrigger extends AfterUpgradeVersionTrigger{

   construct() {
    super(dataModelVersionNumber)
  }

   override function execute() {
    // Perform actions here.
  }

   override property get Description() : String {
    return "Description of the version trigger."
  }

 }
```

### Altering columns to match the data model using a version trigger

In most cases, you do not need to alter a column to match a change to the column type in the logical data model. PolicyCenter automatically applies data model changes to the database during the upgrade. However, this occurs *after* all custom `BeforeUpgradeVersionTrigger` instances have run, so Guidewire provides methods to alter database columns to match the data model.

### Modify a single column using a version trigger

#### About this task

If you need to modify a single column for use in a `BeforeUpgradeVersionTrigger,` use the following procedure.

#### To modify a single column

#### Procedure

1. Modify the data model file.
2. Use the `alterColumnTypeToMatchDatamodel` method of `IBeforeUpgradeColumn` to make your changes.

#### Example

For example:

```
var table = getTable("TableName")
var column = table.getColumn("ColumnName")
column.alterColumnTypeToMatchDatamodel()
```

### Modify multiple columns using a version trigger

#### About this task

If you need to modify more than one column for use in a `BeforeUpgradeVersionTrigger,` use the following procedure.

### To modify multiple columns

#### Procedure

1. Modify the data model file.
2. Use the `alterMultipleColumnsToMatchDatamodel` method of `IBeforeUpgradeTable`.

   For example:

```
var table = getTable("TableName")
var columnsToChange = new IBeforeUpgradeColumn[2]

 columnsToChange[0] = table.getColumn("column1")
columnsToChange[1] = table.getColumn("column1")

 table.alterMultipleColumnsToMatchDatamodel(columnsToChange)
```

## Alter a non-nullable column to nullable using a version trigger

### About this task

To alter a column from non-nullable to nullable, use the `IBeforeUpgradeColumn` method `alterColumnToNullable`.

### Example

For example:

```
var table = getTable("TableName");

table.getColumn("ColumnName").alterColumnToNullable();
```

## Creating columns using a version trigger

The database upgrader automatically creates a column that is added to the data model if the column meets one of the following criteria:

- Nullable
- Non-nullable with a default value specified in the metadata
- Non-nullable without a default value if there are no rows in the table
- The column is an editable field

However, you might want to explicitly create the column in your upgrade trigger if you want the trigger to perform an action on the column such as populating it.

In the data model, the column must be defined as a property on an entity. The database upgrade will determine the correct datatype and nullability from the data model.

Creating a new column is moderately expensive in terms of performance of the upgrade.

### Creating a column

To create a column, invoke the `create` method on the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")

 // Create column with given name.
 // Column must be backed by a property on an entity.
 // Upgrader will figure out the correct datatype and nullability.

 table.getColumn("ColumnName").create()
```

### Creating a non-nullable column with an initial value

The upgrader throws an exception if you try to add a new non-nullable column without a default value and there are rows in the table. For non-nullable columns, either specify a default value, or create a version trigger that will populate the column.

To create a new column as non-nullable with an initial value, use the `createNonNullableWithInitialValue()` method. In the data model, the column must be defined as non-nullable.

For example:

```
IBeforeUpgradeTable table = getTable("TableName")
table.getColumn("ColumnName").createNonNullableWithInitialValue(Initial value)
```

The initial value must be of the appropriate type for the column's datatype. You can alter this value in later steps as needed.

### Creating a temporary column

Use the `createTempColumn` method of `IBeforeUpgradeTable` to add a temporary column to the table. The `createTempColumn` method takes two parameters, a `String` for the column name and an `IDataType` for the column data type. `createTempColumn` creates a new nullable column with the given name and datatype to hold temporary data. You must explicitly drop the temporary column during the upgrade. The schema verifier will report an error during server startup if the column has not been dropped. You can create the temporary column in a `BeforeUpgradeVersionTrigger` and drop it in an `AfterUpgradeVersionTrigger`. This approach is useful when you want to move data from a column that will be removed during the upgrade to a column that will be created during the upgrade.

In the following example, a `BeforeUpgradeVersionTrigger` adds a temporary `shorttext` column to an existing entity and populates it with data from another column on a different entity. An `AfterUpgradeVersionTrigger` moves the data to a new entity.

**BeforeUpgradeVersionTrigger Execute Method**

```
// Add a temporary column to TableA.
var tableA = getTable("TableA")
var tempColumn = tableA.createTempColumn("tmp_column", DataTypes.shorttext())

 // Get an IBeforeUpgradeUpdateBuilder for TableA.
var ub = tableA.update()

 // Set the value of the temporary column to the value of ColumnA.
ub.set(tempColumn, ub.getColumnRef("ColumnA"))

 ub.execute()
```

**AfterUpgradeVersionTrigger Execute Method**

```
// Get an IUpdateBuilder for TableA.
var ub = getTable("TableA").update().withLogSQL(true)

 var q = new Query(Account).withLogSQL(true)
q.compare("ID", Equals, ub.getQuery().getColumnRef("Account"))
var piDesc = PaymentInstrument.Type.TypeInfo.getProperty("Description") as IEntityPropertyInfo

 ub.set(piDesc, q, q.getColumnRef(DBFunction.Expr({"tmp_xyz"}))) // tmp_xyz is the DB table column name
ub.execute()

 var tempColumn = getTable("someTable").getColumn("tmp_xyz").drop()
```

## Dropping columns using a version trigger

The upgrader does not drop existing columns in order to prevent data loss. You can implement a version trigger to move the data (not shown in example) and then drop the column by using the `drop()` method of the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")
table.getColumn("ColumnName").drop()
```

There is a `dropColumns` method on `IBeforeUpgradeTable` to drop multiple columns in one statement. The `dropColumns` method takes an array of `IBeforeUpgradeColumn` objects.

For example:

```
var table = getTable("TableName")
table.dropColumns(table.getColumn("ColumnName2"), table.getColumn("ColumnName3"));
```

In Oracle, dropping a column usually has little effect on upgrade performance. Dropping a column actually marks the column as unused in the metadata. At a later point, the DBA is responsible for performing the necessary cleanup. You can override this functionality and force columns to be dropped right away.

In SQL Server, dropping a column is performance-intensive because the RDBMS has to do some clean up work.

### Renaming columns using a version trigger

To rename a column use the rename function on the column object.

```
override function execute() {
  getTable("TableName").getColumn("ColumnName").rename("NewColumnName")
}
```

### Setting a column value for a specific subtype using a version trigger

To set a column to a specific value for specific subtypes, use the `set` and `compare` methods of an `IBeforeUpgradeTable`. Get the typekey ID for comparison using the `BeforeUpgradeVersionTrigger` method `getTypekeyID`.

```
    final var myTable = getTable("tableName")
    final var myTypecode = getTypeKeyID("typelist name", "typelist code")

     final var updateBuilder = myTable.update()

     updateBuilder
      .set("myColumn", "some value")
      .compare("subtype", Equals, myTypecode)

     updateBuilder.execute()
```

### Creating tables using a version trigger

To add a new table to the database, define a new entity in the data model. The upgrade creates the table automatically. However, you might want to explicitly create the table in your upgrade trigger if you want the trigger to perform an action on the table such as populating it.

Creating a new table has negligible impact on upgrade performance.

You can create a regular table using the `create` method of `IBeforeUpgradeTable`. The table must first be defined in the data model.

For example:

```
var table = getTable("TableName").create()
```

#### Creating temporary tables

You can add a temporary table to the database based on either the current database schema for a table or the data model definition of a table. You can also create a temporary table with a custom definition.

To create a temporary table based on the current table schema in the database, use the `createNewTempTableBasedOnCurrentSchema` method of `IBeforeUpgradeTable`. The table must be associated with an entity and exist in the database. The returned temporary table will contain the columns that this table has in the database currently. The columns may not match those specified in the entity metadata. For example, the metadata might contain a new column that has not yet been created. The `createNewTempTableBasedOnCurrentSchema` method is usually more appropriate than `createNewTempTableBasedOnThis` if you want to copy data from this table into the new temporary table as the columns will match exactly.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnCurrentSchema()
```

To create a temporary table based on the entity definition of a table in the data model, use the `createNewTempTableBasedOnThis` method of `IBeforeUpgradeTable`. Columns that do not exist in the table are not created on the temporary table, even if the metadata defines such a column. This table may not contain columns that are going to be renamed. The metadata reflects the new name for the column but does not have an entry for the old name, so it would not be added to the temporary table.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnThis()
```

To create a temporary table with a custom definition, use the `createAsNewTempTable` method of `IBeforeUpgradeTable`. This method takes a Pair array in which the first object is a `String` defining the column name and the second object is an `IDataType` defining the column data type.

### Renaming tables using a version trigger

To rename a table use the rename function on the table object.

```
override function execute() {
  getTable("extTableName").rename("TableName_EXT")
}
```

### Deleting rows using a version trigger

To delete rows from a table, use the `delete` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `delete` method returns a delete builder (`IBeforeUpgradeDeleteBuilder` that provides methods for comparing column data to restrict the rows that are deleted.

In the following example, all rows that have a `columnA` value of `0` are deleted.

```
var table =  getTable("SomeTable")

 var deleteBuilder = table.delete()

 deleteBuilder.Query.compare("columnA", Equals, 0)

 deleteBuilder.execute()
```

### Inserting rows using a version trigger

To insert rows of data use the `insert` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insert` method returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable)` for SQL to perform an insert operation.

In the following example, an `IBeforeUpgradeInsertBuilder` is used to add two rows with three columns to table `myTable`. The `IBeforeUpgradeInsertBuilder` includes a description.

```
    var myTable = getTable("SomeTable")

     var insertBuilder = myTable.insert().withDescription("A custom insert
      trigger to add two rows.")
     insertBuilder
      .mapColumn("columnA", "value of column A for first row")
      .mapColumn("columnB", "value of column B for first row")
      .mapColumn("columnC", "value of column C for first row")

     insertBuilder.execute()

 // add a second row
    insertBuilder
      .mapColumn("columnA", "value of column A for second row")
      .mapColumn("columnB", "value of column B for second row")
      .mapColumn("columnC", "value of column C for second row")

     insertBuilder.execute()
```

### Inserting data selected from another table using a version trigger

To insert data selected from another table use the `insertSelect` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insertSelect` method returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`). The builder includes a `mapColumn` method that can be passed explicit values, columns, or a query.

In the following example, the trigger sets `targetTable.column1` to an explicit value. The trigger sets `targetTable.column2` to the value of `sourceTable.sourceColumn`. Because there is no comparison being performed, the trigger will insert a row in the target table for each row in the source table:

```
var sourceTable = getTable("sourceTable")
var targetTable =  getTable("targetTable")

 var insertSelectBuilder = targetTable.insertSelect(sourceTable)

 insertSelectBuilder.mapColumn("column1", "value")    // sets a hard-coded value
  .mapColumn("column2", sourceTable.getColumn("sourceColumn"))    // sets column2 on target table to
                                                                  // source table sourceColumn

 insertSelectBuilder.execute()
```

In the next example, an existing table, `sourceTable`, is split into two tables, `targetTable1` and `targetTable2`.

```
var sourceTable = getTable("sourceTable")
var targetTable1 =  getTable("targetTable1")
var targetTable2 =  getTable("targetTable2")

 var insertSelectBuilder1 = targetTable1.insertSelect(sourceTable)
var insertSelectBuilder2 = targetTable2.insertSelect(sourceTable)


 insertSelectBuilder1.mapColumn("column1", sourceTable.getColumn("sourceColumn1"))
  .mapColumn("column2", sourceTable.getColumn("sourceColumn2"))

 insertSelectBuilder1.execute()

 insertSelectBuilder2.mapColumn("column1", sourceTable.getColumn("sourceColumn3"))
  .mapColumn("column2", sourceTable.getColumn("sourceColumn4"))

 insertSelectBuilder2.execute()
```

### Updating rows using a version trigger

To update rows in a table, use the `update` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. This method returns a builder (`IBeforeUpgradeUpdateBuilder` or `IUpdateBuilder`). The builder includes methods to compare data to restrict which rows are updated.

In the following example, table `SomeTable` is updated to set `column1` to `SomeValue` for each row where the subtype matches a certain entity type:

```
var table =  getTable("SomeTable")

 // get IBeforeUpgradeUpdateBuilder
```

```
var ub = table.update()

 // set column 1 to SomeValue
ub.set("column1", "SomeValue")
  // where
  .compare("subType", Equals, getTypeKeyID(EntityType))

 ub.execute()
```

## Upgrading archived or quote store entities using a version trigger

If you implement archiving, and you make custom data model changes to entities stored in the archive, then you can upgrade the retrieved XML of your archived entity using the `IDatamodelUpgrade` plugin.

You can also use this plugin to upgrade entities in the quote store created by a quote-only instance of PolicyCenter that handles high volume quote requests.

Simple data model changes do not require a custom trigger. These include:

- Adding a new entity
- Updating denormalization columns
- Adding editable columns such as `updatetime`
- Adding new columns
- Changing the nullability of a column

More complex transformations or those that could result in loss of data require a version trigger. These include:

- changing a datatype (other than just length)
- migrating data from one table or column to another
- dropping a column
- dropping a table
- renaming a column
- renaming a table

PolicyCenter upgrades an archived entity as the entity is restored.

The `IDatamodelChange` interface includes a `getArchivedDocumentUpgradeVersionTrigger` method that returns an `ArchivedDocumentUpgradeVersionTrigger`.

You can define custom `ArchivedDocumentUpgradeVersionTrigger` entities to modify archived XML. The `ArchivedDocumentUpgradeVersionTrigger` is an abstract class that you can extend to create your custom triggers.

Define the constructor of your custom `ArchivedDocumentUpgradeVersionTrigger` to call the constructor of the superclass and pass it a numeric value. For example:

```
construct() {
  super(171)
}
```

This numeric value is the extension version to which your trigger applies. If you run the upgrade against a database with a lower extension version, then your custom trigger is called. The current extension version is defined in `modules/configuration/config/extensions/extensions.properties.`

Provide an override definition of the `get Description` property to return a `String` that describes the actions of your trigger.

Provide an override definition for the `execute` function to define the actions that you want your custom trigger to make on archived XML.

When the upgrade executes your custom trigger, it wraps each XML entity in an `IArchivedEntity` object. Each typekey is wrapped in an `IArchivedTypekey` object. The upgrade operates on a single XML document at a time.

`ArchivedDocumentUpgradeVersionTrigger` provides the following key operations:

- `getArchivedEntitySet(`*`entityName`*` : String)` – returns an `IArchivedEntitySet` object that contains all `IArchivedEntity` objects of the given type in the XML document.

`IArchivedEntitySet` provides the following key methods:

- `rename(`*`newEntityName`*` : String)` – renames all rows in the set to the new name.
- `delete()` – deletes all rows in the set.
- `search(`*`predicate`*` : Predicate<IArchivedEntity>)` – returns a `List` of `IArchivedEntity` objects that match the given predicate.
- `create(`*`referenceInfo`*` : String, `*`properties`*` : List<Pair<String, Object>>)` – returns a new `IArchivedEntity` with the given properties.

`IArchivedEntity` provides the following key methods:

- `delete()` – deletes just this row.
- `getPropertyValue(`*`propertyName`* : String`)` – returns the value of the property of the given name. If the property value is a reference to another entity, this method returns an `IArchivedEntity`.
- `move(`*`newEntityName`*` : String)` – moves this to a new entity type of the given name. The type is created if it did not exist. You must add any required properties to the type.
- `updatePropertyValue(`*`propertyName`*` : String, `*`newValue`*` : String)` – updates the property of the given name to the given value.
- `getArchivedTypekeySet(`*`typekeyName`*` : String)` – returns an `IArchivedTypekeySet` object that contains all `IArchivedTypekey` objects in the given typelist.
- `getArchivedTypekey(typelistName : String, code : String)` – Returns an `IArchivedTypekey` representing the typekey.

  `IArchivedTypekey` provides the following key method:

  - `delete()` – deletes the typekey from the XML.

More methods are available for `IArchivedEntitySet`, `IArchivedEntity` and `IArchivedTypekey`. See the Gosu documentation for a full listing. Generate the Gosu documentation by navigating to the PolicyCenter directory and entering the following command:

```
gwb genGosuApi
```

## Incremental upgrade

When PolicyCenter archives an entity, it records the current data model version on the entity. PolicyCenter upgrades an archived entity as the entity is restored. The upgrader executes the necessary archive upgrade triggers incrementally on each archived XML entity, according to the data model version of the archived entity.

An entity archived at one version might not be restored and upgraded until several intermediate data model upgrades have been performed. Therefore, do not delete your custom upgrade triggers.

Consider the following situation. Note that this example is for demonstration purposes only. The version numbers included do not represent actual PolicyCenter versions but are included to explain the incremental upgrade process. Each '+' after a version number indicates a custom data model change.

**Guidewire data model changes**

v6.0.0 – Entity does not have column X.

v6.0.1 – Guidewire adds column X to the entity with a default value of 100.

v6.0.2 – Guidewire updates the default value of column X to 200.

**Implementation #1 upgrade path**

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.0+++ – More custom data model changes.

v6.0.2+ – column X introduced with a default value of 200.

Result of upgrade of an entity archived at v6.0.0+: column X has value 200.

In this situation, version 6.0.1 was skipped in the upgrade path. Therefore, column X is added with the default value of 200 that it has in version 6.0.2.

**Implementation #2 upgrade path**

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.1+ – column X introduced with a default value of 100.

v6.0.2+ – column X already exists.

Result of upgrade of an entity archived at v6.0.0+: column X has value 100.

In this situation, column X is added in version 6.0.1 with the default value of 100. During the upgrade from version 6.0.1 to version 6.0.2, column X already exists. Therefore, the upgrader does not add the column. A custom trigger would be required to update the value of X from 100 to 200 during the upgrade from version 6.0.1 to 6.0.2.

## Configuring database upgrade

You can set parameters for the database upgrade in the PolicyCenter 10.0.0 `database-config.xml` file. The `<database>` block in `database-config.xml` contains parameters for database configuration, such as connection information. The `<database>` block contains an `<upgrade>` block that contains configuration information for the overall database upgrade. The `<upgrade>` block also contains a `<versiontriggers>` element for configuring general version trigger behavior and can contain `<versiontrigger>` elements to configure each version trigger.

#### See also

The `<versiontriggers>` database configuration element in the *Installation Guide*.

*Installation Guide*.

### Adjusting commit size for encryption

You can adjust the commit size for rows requiring encryption by setting the `encryptioncommitsize` attribute to an integer in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade encryptioncommitsize="10000">
    ...
  </upgrade>
</database>
```

If PolicyCenter encryption is applied on one or more attributes, the PolicyCenter database upgrade commits batches of encrypted values. The upgrade commits `encryptioncommitsize` rows at a time in each batch. The default value of `encryptioncommitsize` varies based on the database type. For Oracle, the default is `10000`. For SQL Server, the default is `100`.

Test the upgrade on a copy of your production database before attempting to upgrade the actual production database. If the encryption process is slow, and you cannot attribute the slowness to SQL statements in the database, try adjusting the `encryptioncommitsize` attribute. After you have optimized performance of the encryption process, use that `encryptioncommitsize` when you upgrade your production database.

### Configuring version trigger elements

The database upgrade executes a series of version triggers that make changes to the database to upgrade between versions. You can set some configuration options for version triggers in `database-config.xml`. Normally, the default settings are sufficient. Change these settings only while investigating a slow database upgrade.

The `<database>` element in `database-config.xml` contains an `<upgrade>` element to organize parameters related to database upgrades. Included in the `<upgrade>` element is a `<versiontriggers>` element, as shown below:

```
<database ...>
  <param ... />
  <upgrade>
    <versiontriggers dbmsperfinfothreshold="600" />
  </upgrade>
</database>
```

The `<versiontriggers>` element configures the instrumentation of version triggers. This element has one attribute: `dbmsperfinfothreshold`. The `dbmsperfinfothreshold` attribute specifies for all version triggers the threshold after which the database upgrader gathers performance information from the database. You specify `dbmsperfinfothreshold` in seconds, with a default of `600`. If a version trigger takes longer than `dbmsperfinfothreshold` to execute, PolicyCenter:

- Queries the underlying database management system (DBMS).
- Builds a set of html pages with performance information for the interval in which the version trigger was executing.
- Includes those html pages in the upgrade instrumentation for the version trigger.

You can completely turn off the collection of database snapshot instrumentation for version triggers by setting the `dbmsperfinfothreshold` to 0 in `config.xml`. If you do not have the license for the Oracle Diagnostics Pack, you must set `dbmsperfinfothreshold` to 0 before running the upgrade.

The `<versiontriggers>` element can contain optional `<versiontrigger>` elements for each version trigger. Each `<versiontrigger>` element can contain the following attributes.

| Attribute | Type | Description |
|---|---|---|
| `name` | String | The case-insensitive name of a version trigger. |
| `extendedquerytracingenabled` | Boo-lean | Oracle only. Controls whether or not to enable extended sql tracing (Oracle event 10046) for the SQL statements that are executed by the version trigger. Default is `false`. The output can be very useful when debugging certain types of perform-ance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation. |
| `parallel-dml` | Boo-lean | Oracle only. See "Configuring parallel dml and DDL statement execution for Ora-cle database upgrade" on page 52. |
| `parallel-query` | Boo-lean | Oracle only. Hints to the optimizer to use parallel queries when executing queries run by a version check associated with the upgrade trigger. See note below. |
| `queryoptimizertracingenabled` | Boo-lean | Oracle only. Controls whether or not to enable query optimizer tracing (Oracle event 10053) for the SQL statements that are executed by the version trigger. De-fault is `false`. The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation. |
| `recordcounters` | Boo-lean | Controls whether the DBMS-specific counters are retrieved at the beginning and end of the use of the version trigger. Default is `false`. If `true`, then PolicyCenter retrieves the current state of the counters from the underlying DBMS at the be-ginning of execution of the version trigger. If the execution of the version trigger exceeds the `dbmsperfinfothreshold`, then PolicyCenter retrieves the current state of the counters at the end of the execution of the version trigger. PolicyCenter writes differences to the DBMS-specific instrumentation pages of the upgrade instrumentation. |
| `updatejoinorderedhint` | Boo-lean | Oracle only. Whether to use the ORDERED hint for the UPDATE of a join. Default is false. |
| `updatejoinusemergehint` | Boo-lean | Oracle only. Whether to use the USE_MERGE hint for the UPDATE of a join. De-fault is false. |
| `updatejoinusenlhint` | Boo-lean | Oracle only. Whether to use the USE_NL hint for the UPDATE of a join. Default is false. |

**Note:** For Oracle, there is an important exception to the ability of the `<versiontriggers>` element to provide overrides for its `<versiontrigger>` subelements. Although the parent element, `<upgrade>`, might have defined the `ora-parallel-dml` and `ora-parallel-query` attributes, these values are not applied to version checks which are included as `<versiontrigger>` subelements. To get a version check to run with parallel query in Oracle, you must list it explicitly, add it as a subelement of the `<versiontriggers>` element, and indicate `parallel-query="true"` on that `<versiontrigger>` subelement.

For example: `<versiontrigger name="com.guidewire.cc.system.database.upgrade.check.RecoveryCategoryNullForPaymentsAndReservesVersionCheck" parallel-query="true"/>`. Other version triggers (but not version checks) that are not listed will inherit the parallel setting defined on the `<upgrade>` element. Note that even though `RecoveryCategoryNullForPaymentsAndReservesVersionCheck` is a version check, it must be included as `<versiontrigger>` element.)

In summary, version checks are exempted from Oracle parallelism unless you explicitly list them. Version triggers are not.

## Deferring creation of nonessential indexes

You can configure the upgrade to defer creation of nonessential indexes during the upgrade process until the upgrade completes and the application server is online. Nonessential indexes are performance-related indexes that do not enforce constraints and indexes on the `ArchivePartition` column on all entities that PolicyCenter can archive. Creation of nonessential indexes can add significant time to the upgrade duration, so it is possible to defer this process. By default, the upgrade does not defer creation of these indexes.

To configure the upgrade to defer creation of nonessential indexes set the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` to `true`.

```
<database ...>
  <upgrade defer-create-nonessential-indexes="true">
  ...
  </upgrade>
</database>
```

If you opt to defer creation of nonessential indexes, PolicyCenter runs the `DeferredUpgradeTasks` batch process as soon as the upgrade completes and the server is completely started. The `DeferredUpgradeTasks` batch process creates the nonessential performance indexes and indexes on archived entities. The database user must have permission to create indexes until after the `DeferredUpgradeTasks` batch process is complete.

Deferring nonessential index creation can shorten the duration of the upgrade process. The PolicyCenter database is then available sooner for tasks including upgrade verification and backing up the upgraded database before the database is opened up for production use. To take advantage of this earlier availability, perform upgrade testing and validation tasks while the `DeferredUpgradeTasks` batch process is running. Do not go into full production while the process is still running. The lack of so many performance-related indexes could likely make the system unusable.

Until the `DeferredUpgradeTasks` batch process has run to completion, PolicyCenter reports errors during schema validation when starting. These include errors for column-based indexes existing in the data model but not in the physical database and mismatches between the data model and system tables.

Do not use the archiving feature until the `DeferredUpgradeTasks` batch process has completed successfully.

Check the status of the `DeferredUpgradeTasks` batch process to determine when it has completed successfully. You can find the status of the deferred upgrade in the upgrade logs and on the PolicyCenter **Upgrade Info** page. If the `DeferredUpgradeTasks` batch process fails, manually run the batch process again during non-peak hours.

If you do not opt to defer creation of nonessential indexes, PolicyCenter creates these indexes as part of the upgrade process that must complete before the application server is online. If you do not want to defer creating nonessential indexes, the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` must be set to `false`. This is the default setting.

## Configuring the upgrade on Oracle

### Configuring column removal for Oracle database upgrade

The database upgrade removes some columns. For Oracle, you can configure whether the removed columns are dropped immediately or are marked as unused. Marking a column as unused is a faster operation than dropping the column immediately. However, because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments. You can drop the unused columns after the upgrade during off-peak hours to free the space. Or, you can configure the database upgrade to drop the columns immediately during the upgrade. By default, the PolicyCenter database upgrade marks columns as unused.

To configure the PolicyCenter upgrade to drop columns immediately during the upgrade, set the `deferDropColumns` attribute of the `<upgrade>` block in `database-config.xml` to `false`. For example:

```
<database ...>
  ...
  <upgrade deferDropColumns="false">
    ...
  </upgrade>
</database>
```

By default, `deferDropColumns` is `true`.

### Configuring parallel dml and DDL statement execution for Oracle database upgrade

You can configure whether the upgrade executes DML (Data Manipulation Language) and DDL (Data Definition Language) statements in parallel or not and the degree of parallelism to use.

The `<upgrade>` element includes an `ora-parallel-dml` attribute. This attribute can be set to `disable`, `enable`, or `enable_all`. The default value is `enable`. If `ora-parallel-dml` is set to `disable`, the upgrade does not conduct parallel execution of DML statements. If `ora-parallel-dml` is set to `enable`, the upgrade executes DML statements in parallel if configured or coded for a version trigger. If `ora-parallel-dml` is set to `enable-all`, the upgrade executes DML statements in parallel in all cases unless turned off in the code or configuration for a version trigger.

The Boolean attribute `parallel-dml` of a `<versiontrigger>` element controls parallel execution for that version trigger. If `parallel-dml` is not set, the upgrade executes parallel DML statements if coded or if `ora-paralled-dml` is set to `enable_all` on the `<upgrade>` element. If `parallel-dml` is set to `false`, the upgrade does not execute DML statements in parallel. If `parallel-dml` is set to `true`, the upgrade executes DML statements in parallel if `ora-parallel` is set to `enable` or `enable_all`.

To configure the degree of parallelism for insert, update and delete operations, set the `degree-of-parallelism` attribute on the `<upgrade>` element. To configure the degree of parallelism for commands such as creating an index and enabling constraints using the alter table command, set the `degree-parallel-ddl` attribute on the `<upgrade>` element.

You can specify a value from `2` to `1000` to force that degree of parallelism. Specify a value of `1` to disable the use of parallel execution.

Setting either parameter to `0` configures PolicyCenter to defer to Oracle to determine the degree of parallelism for the operations that attribute configures. The Oracle automatic parallel tuning feature determines the degree based on the number of CPUs and the value set for the Oracle parameter `PARALLEL_THREADS_PER_CPU`.

The default for both attributes is `4`.

You can configure parallel DML execution on the `InsertSelectBuilder`, `BeforeUpgradeUpdateBuilder` and `BeforeUpgradeInsertSelectBuilder` of a custom version trigger using the `withParallelDml(`*`boolean`*`)` method. If not explicitly set to `true` or `false`, the upgrade uses parallel execution if configured. If set to `false`, the upgrade does not use parallel execution unless set to `true` for that version trigger. If set to `true`, it will be done unless set to false for that version trigger or `ora-parallel-dml` is set to `disable`.

Some version triggers read large amounts of data when running their version check. To improve performance, you can configure a version trigger to hint to the optimizer to use parallel queries when executing SQL queries. To do so, set the `parallel-query` attribute on the `<versiontrigger>` element to `true`.

### Collecting tablespace usage and object size for Oracle database upgrade

To enable collection of tablespace usage and object size data on Oracle, set the `collectstorageinstrumentation` attribute of the `<upgrade>` block to `true`. For example:

```
<database ...>
  ...
  <upgrade collectstorageinstrumentation="true">
    ...
  </upgrade>
</database>
```

A value of `true` enables PolicyCenter to collect tablespace usage and size of segments such as tables, indexes and LOBs (large object binaries) before and after the upgrade. The values can then be compared to find the utilization change caused by the upgrade.

### Disabling Oracle logging for Oracle database upgrade

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="true">
    ...
  </upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run statements with the `NOLOGGING` option.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. Some examples include Reporting, Disaster Recovery through Standby databases and Oracle Dataguard. To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

### Disabling statistics update for the database for Oracle database upgrade

Generating table statistics during upgrade is optional for Oracle databases. The overall time required to upgrade the database is shorter if the database upgrade does not update statistics. To disable statistics generation during the upgrade, set the `updatestatistics` attribute of the `<upgrade>` element to `false`:

```
<upgrade updatestatistics="false">
```

Setting `updatestatistics` to `true` enables the upgrader to update statistics on changed objects. It also allows the upgrade to maintain column level statistics consistent with what is allowed in the code, data model and configuration.

If statistics are not updated during the upgrade, PolicyCenter reports a warning that recommends that you run the database statistics batch process in incremental mode. Additionally, the **Upgrade Info** page shows that statistics were not updated as part of the upgrade. If statistics generation was not disabled, the **Upgrade Info** page reports the runs of the statistics batch process, including incremental runs.

You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded PolicyCenter in a production environment. If you defer generating statistics during the upgrade, Guidewire recommends that you generate full statistics as soon as possible after the upgrade. For instructions, see the *System Administration Guide*.

The **Upgrade Info** page does not identify the following case: You ran an upgrade with `updatestatistics=true` after running a previous upgrade with `updatestatistics=false`, but you did not update statistics first.

When you click the **Download** button on the **Upgrade Info** page, you get a more detailed report. This report shows the value of the `updatestatistics` attribute at the time of upgrade. Additionally, the report shows the update statistics SQL statements that were skipped as part of the upgrade. These statements are provided for reference. You typically do not need to review these statements if you run the incremental database statistics process following the upgrade.

### Disabling statistics update for tables with locked statistics for Oracle database upgrade

If you have tables that have locked statistics, specify to keep statistics on these tables before starting the database upgrade. To specify to keep statistics on a table, set the `action` attribute of the `<tablestatistics>` element for that table to `keep`. The `<tablestatistics>` element is nested within the `<databasestatistics>` element, which is within the `<database>` element in `database-config.xml`.

For example, if statistics are locked on `pc_someTable_EXT`, specify a `<tablestatistics>` element for that table with the `action` attribute set to `keep`:

```
<database>
  ...
  <databasestatistics>
    <tablestatistics name="pc_someTable_EXT" action="keep" />
  </databasestatistics>
</database>
```

## Configuring the upgrade on SQL Server

### Disabling SQL Server logging for SQL Server database upgrade

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="true">
    ...
  </upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run with minimal logging. This can improve the performance of the upgrade. During the upgrade, set the SQL Server recovery model to Simple or Bulk logged. Once the upgrade and deferred upgrade tasks are complete, you can revert the recovery model setting and back up the full database.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. If you require full logging due to the presence of solutions such as Database Mirroring, continue to use the Full recovery model and set `allowUnloggedOperations` to `false`.

To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

### Storing temporary sort results in tempdb for SQL Server database upgrade

For SQL Server databases, you can specify to store temporary sort results in tempdb by setting the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` block to `true`. By using tempdb for sort runs, disk input and output is typically faster, and the created indexes tend to be more contiguous. By default, `sqlserverCreateIndexSortInTempDB` is `false` and sort runs are stored in the destination filegroup.

If you set `sqlserverCreateIndexSortInTempDB` to `true`, you must have enough disk space available to tempdb for the sort runs, which for the clustered index include the data pages. You must also have sufficient free space in the destination filegroup to store the final index structure, because the new index is created before the old index is deleted. Refer to `http://msdn.microsoft.com/en-us/library/ms188281.aspx` for details on the requirements to use tempdb for sort results.

### Specifying filegroup to store sort results for clustered indexes for SQL Server database upgrade

For SQL Server databases, a version trigger recreates non-clustered backing indexes for primary keys as clustered indexes.

Before recreating the indexes, the version trigger automatically drops (and later rebuilds) any referencing foreign keys and drops any clustered indexes on tables with a primary key.

If you are using filegroups, the upgrade recreates the clustered index in the OP filegroup. By default, the upgrade also stores the intermediate sort results that are used to build the index in the OP filegroup. You can configure the upgrade to instead use the tempdb filegroup for the intermediate sort results.

If you want the upgrade to stores the intermediate sort results in the tempbdb filegroup, set the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` element to `true`.

```
<database ...>
  ...
  <upgrade sqlserverCreateIndexSortInTempDB="true" />
    ...
  </upgrade>
</database>
```

This option increases the amount of temporary disk space that is used to create an index. However, it might reduce the time that is required to create or rebuild an index when tempdb is on a different set of disks from that of the user database.

By default, `sqlserverCreateIndexSortInTempDB` is `false`.

### Downloading database upgrade instrumentation details

The database upgrade deletes upgrade instrumentation information for prior database upgrades. If the database upgrade detects any prior upgrade instrumentation data, it reports a warning and deletes the data. If you have run previous database upgrades, and you want to preserve upgrade instrumentation details, follow the procedure in "Viewing detailed database upgrade information" on page 60.

## Checking the database before upgrade

The upgrade runs a series of version checks prior to making any changes to the database. These version checks ensure that the database is in a state that can be upgraded. Guidewire includes a number of version checks with PolicyCenter and you can also add custom version checks.

You can configure PolicyCenter to run the version checks only, including custom version checks. Before upgrading the production database, run version checks on a clone of your production database to identify any issues with your data.

### Run version checks without database upgrade

#### Procedure

1. Start Studio for PolicyCenter 10.0.0 by running the following command from the PolicyCenter directory:

   ```
   gwb studio
   ```

2. Expand **configuration→config** and open `database-config.xml`.
3. Add the attribute `versionchecksonly=true` to the `database` element.
4. Verify that the database connection is pointing to a clone of your production database.
5. Save your changes.
6. Start the server.

   PolicyCenter reports the number of version check errors. For any errors reported PolicyCenter reports which version check resulted in the error along with the error message.
7. If PolicyCenter reports version check errors, fix the data and rerun the version checks. Repeat this process until no errors are reported on the production clone. Apply the fixes to your production database prior to upgrade.

   With `versionchecksonly=true` set, PolicyCenter runs all version checks regardless of a failure in one of the checks. During a regular upgrade, PolicyCenter stops the upgrade if an error is detected.
8. After you have fixed all version check errors, set `versionchecksonly` to `false` to run the actual upgrade.

## Disable the scheduler before upgrade

### About this task

Before you start the server to upgrade the database, disable the scheduler for batch processes and work queues. Disabling the scheduler prevents batch processes and work queues from launching immediately after the database upgrade.

### Procedure

1. Open the PolicyCenter 10.0.0 `config.xml` file in a text editor.
2. Set the `SchedulerEnabled` parameter to `false`.

   ```
   <param name="SchedulerEnabled" value="false"/>
   ```

3. Save `config.xml`.

### Next steps

After you have verified that your database upgrade completed successfully, enable the scheduler again.

### See also

"Enable the scheduler after upgrade" on page 64

## Starting the server to begin automatic database upgrade

The database upgrade is an automatic process that occurs as you start the server with the upgraded configuration of a new PolicyCenter version. The database upgrade normally completes in a few hours or less.

If the database upgrade stops before completing, then restore your database from the backup, correct any issues reported, and repeat the database upgrade.

**IMPORTANT** Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of PolicyCenter 10.0.0. See the *Supported Software Components* knowledge article for current system and patch level requirements. Visit the Guidewire Community and search for the *Supported Software Components* knowledge article.

**WARNING** Except for your first database upgrade trials, do not start the server until you have upgraded all rules. Otherwise, default validation rules execute. This could strand objects at a high validation level and make it impossible to edit parts of the object.

**WARNING** The database upgrade runs a series of version checks prior to making any changes. If any of these checks fail, the upgrade aborts and reports an error message. You can fix the issue, create an updated backup of the database and attempt the upgrade again without restoring from a backup. However, if you experience a failure during the version triggers or upgrade steps portion of the upgrade, refresh the database from a backup before attempting the upgrade again.

Guidewire requires that you use a separate mirror database for reporting. If you did not do so, then you can experience problems during a database upgrade that are severe enough to prevent the upgrade.

In particular, the Premium Accounting extension package SQL scripts create special tables in the reporting database that reporting uses for storing premium accounting accrual data. The reporting scripts use these tables, but the PolicyCenter application does not. If you created these tables in a production database, then any attempt to upgrade that production database will fail.

If you encounter this situation, move data from all `pcrt_` prefixed tables to another schema. Then, drop all `pcrt_` prefixed tables from the database that you want to upgrade before starting the server to launch the upgrade.

## Testing the database upgrade

In a development environment the database upgrade process records checkpoints of upgrade triggers that complete successfully. You can restart a failed database upgrade, and it resumes with the upgrade trigger that failed. This restart feature helps you test the upgrade with realistically large data sets. You avoid time spent to restore the database and rerun upgrade triggers that worked successfully.

Guidewire provides this feature for convenience while testing. However, it does not work for all failure scenarios. Even in development mode, under certain scenarios, you will have to restore a backup of the database taken prior to the upgrade attempts and then run the upgrade.

The database upgrade writes SQL executed by the failed trigger to the console. To restart a test database upgrade from a checkpoint reached in an earlier upgrade, manually roll back any database changes that occurred during the upgrade trigger that failed. Resolve the problem that caused the trigger to fail. Then start the server again to restart the upgrade. The upgrade skips successful upgrade triggers and continues by rerunning the trigger that failed.

### Test the database upgrade

#### About this task

Prior to attempting the database upgrade on a full-production database clone, test the database upgrade.

In a development environment the database upgrade process records checkpoints of upgrade triggers that complete successfully. You can restart a failed database upgrade, and it resumes with the upgrade trigger that failed. This restart feature helps you test the upgrade with realistically large data sets. You avoid time spent to restore the database and rerun upgrade triggers that worked successfully.

Guidewire provides this feature for convenience while testing. However, it does not work for all failure scenarios. Even in development mode, under certain scenarios, you will have to restore a backup of the database taken prior to the upgrade attempts and then run the upgrade.

The database upgrade writes SQL executed by the failed trigger to the console. To restart a test database upgrade from a checkpoint reached in an earlier upgrade, manually roll back any database changes that occurred during the upgrade trigger that failed. Resolve the problem that caused the trigger to fail. Then start the server again to restart the upgrade. The upgrade skips successful upgrade triggers and continues by rerunning the trigger that failed.

**WARNING** Never use the restart feature of database upgrade in a production environment.

#### Procedure

1. Connected to the built-in Quickstart database, successfully start the built-in Quickstart application server with a merged configuration data model, including merged extensions, data types, field validators, and so forth.
2. Connected to an empty database on an Oracle or SQL Server database server, successfully start the Quickstart application server from the preceding step.
3. Connected to a restored backup of a production clone, start either the same Quickstart server from the preceding step or a supported third-party application server with your custom configuration.

#### Next steps

A test run of your upgrade is successful only when it runs from start to finish without a restart.

## Integrations and starting the server

Disable all integrations during the automatic database upgrade. Integration points might require updates due to changes in Guidewire APIs. See the *New and Changed Guide* for specifics.

It is not necessary to have completely migrated integrations before attempting to start the server for the first time. If you have integrations that rely on non-Guidewire applications, do not expect these integrations to work the first time you start the server.

## Understanding the automatic database upgrade

As the database upgrade proceeds, it logs messages to the console as well as the log file describing its progress. The database upgrade process requires thousands of steps, divided into three phases. Due to the relational nature of a database, these phases must execute in a specific order for the upgrade to succeed.

During the first phase, the upgrader first executes custom `BeforeUpgradeVersionTrigger` version checks and triggers defined in the `IDatamodelUpgrade` plugin. The upgrader next runs version checks defined by Guidewire. Then, the upgrader uses a set of version triggers defined by Guidewire to determine the actions that are required. The database upgrader requires version triggers in order to perform the following types of tasks:

- Changing a datatype (other than just length)
- Migrating data
- Dropping a column
- Dropping a table
- Renaming a column
- Renaming a table

### See also

"Version trigger descriptions" on page 58

Many version triggers have version checks associated with them. These checks ensure that the database is ready for the associated version trigger. The database upgrade runs all checks before running any version triggers. If a check detects a problem, it reports the issue, including a sample SQL query to find specific problematic records. If a version check discovers an issue, the database upgrade stops before any version triggers are run. Therefore, it is not necessary to restore the database from a backup if a version check reports an error. Correct the issue and then create a new backup of the database. Then, if you encounter errors after the version check stage, you can restore a version of your database with the issue reported by the version check resolved.

In the second phase, the upgrader compares the target data model and the current database to determine how they differ. The upgrader makes changes to the database that do not require a version trigger during this phase.

Following this process, the third phase runs a subsequent set of version triggers. These triggers create actions that must be run last due to a dependency on an earlier phase.

After the database upgrade concludes, it reports issues that the upgrader encountered and did not complete.

You are responsible for correcting these issues. This might involve modifying the data model or altering the table manually. If you do not correct them, the next time you start the server you do *not* see a message that the database and the data model are out of sync. You must then use the `system_tools` command to verify the database schema.

> **Note:** Given the complexity of database upgrade, Guidewire does not expose specific upgrade actions/steps to clients either in SQL or Java form. Any manual attempts to recreate or control the upgrade process can result in problems in the PolicyCenter database. Recovery from such attempts is not supported.

## Version trigger descriptions

PolicyCenter uses version triggers to update the database during an upgrade. Review the version trigger descriptions to familiarize yourself with the changes that will be applied to your database. If a version trigger has an associated version check, the version check is described with the trigger. If a version check reports an issue, review the error message and consult the description of the relevant version trigger for more information.

### Marking underwriting rule as externally managed

This trigger sets `UWRule.ExternallyManaged` to `true` if the `UWRule` has no `RuleCondition` and `AvailableToRun` is `false`. Before upgrade, ensure that Gosu rules are not marked `AvailableToRun`.

The result will be incorrect for rules for which the rule condition is not yet defined, but is not externally managed. For example, a rule under development may not yet have a rule condition. If this happens during your upgrade, delete the rule and redefine it.

### Drop tableregistry table

Drops the `tableregistry` table.

### Drop all staging tables

Drops all staging tables.

### Drop RuleHead and RuleVersion subtypes

Updates the `RuleHead` and `RuleVersion` tables to drop the subtype column.

### Create ConfigFPColumn for RollingUpgrade

Creates the `ConfigFP` configuration fingerprint column in the `RollingUpgrade` table.

### Alter Datetime columns to Datetime2

On SQL Server ONLY, alters all `datetime` columns to `datetime2`.

### Add rule type to RuleExportImportTask

Replaces the obsolete the obsolete `RuleVersionType` attribute with the new `RuleType` attribute for the `RuleExportImportTask` entity. This trigger creates the column and sets its value to the existing single use case rule subtype entity typekey.

### Add ActivityPattern

Adds the `async_quote_completed` activity pattern if it is absent.

### Add InvoicingMethod typekey

Adds the `InvoicingMethod` typekey to `PolicyPeriod`, and converts `CustomBilling` to `InvoicingMethod`. Removes `CustomBilling` from `PolicyPeriod`. For upgrade from a release prior to 8.0.2, converts the `BillingInvoiceStream.selected` to `InvoicingMethod`.

### Remove CreateUWIssuePermission

Removes all role privileges with the permission `createevalissue`.

### Rename AccountFrozen to LockedFromMerge

Renames the `Frozen` column on `Account` to `LockedFromMerge`.

### Drop LossRatioColumn from PolicyTermTable

Drops the `LossRatio` column from the `PolicyTerm` table.

### Drop PrimaryDriver column from VehicleDriver table

Drops the `PrimaryDriver` column from the `VehicleDriver` table.

### Drop ETLPurgedRoot Table

Drops the `ETLPurgedRoot` table if the table exists and is empty.

### No duplicate AddressBookUID

Verifies that there are no duplicate rows with matching AddressBookUID in the pc_contact, pc_address, pc_contactaddress and pc_contacttag tables.

#### Dropping deprecated MotorVehicleRecord table

Upgrade drops the `pcx_motorvehiclerecord` table. The MotorVehicleRecord entity has been deprecated since PolicyCenter 7.0.0 and has been removed in PolicyCenter 10.0.0. There is no data migration.

#### See also

- *Motor vehicle records in personal auto* in the *PolicyCenter Application Guide*

### Two-step quoting upgrade triggers

These related upgrade triggers convert the data model for changes introduced by two-step quoting.

#### Add quote maturity level field

This upgrade trigger adds the `QuoteMaturityLevel` column to the `pc_policyperiod` table and populates it based on the value of the existing `ValidQuote` column. If `ValidQuote` is `false`, the trigger sets `QuoteMaturityLevel` to `unrated`. If `ValidQuote` is `true`, the trigger sets `QuoteMaturityLevel` to `quoted`.

#### Drop valid quote column from policy period table

This upgrade trigger drops the `ValidQuote` column from the `pc_policyperiod` table.

## Viewing detailed database upgrade information

PolicyCenter includes a Server Tools **Upgrade and Versions** screen that provides detailed information about each database upgrade. The **Upgrade and Versions** screen includes information on the following:

- Version number of upgrade
- Status of upgrade
- Type of upgrade
- Start and time of upgrade
- Status of Deferred Upgrade Tasks batch processing

From this screen, you can drill down into more specific details about the upgrade, or download a report of the upgrade details.

#### See also

- *System Administration Guide*

## Dropping unused columns on Oracle

By default, the PolicyCenter database upgrade on Oracle marks columns that have been removed from the data model as unused. Marking a column unused is a faster operation than dropping a column. Because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments.

You can configure database upgrade to drop removed columns immediately by setting the `deferDropColumns` parameter to `false` before running the upgrade. This parameter is within the `<upgrade>` block of the `<database>` block of `database-config.xml`.

If you did not set `deferDropColumns` to `true` before the upgrade, perform the procedure to drop unused columns after the upgrade.

You can drop the unused columns after the upgrade during off-peak hours to free the space. PolicyCenter does not have to be shutdown to perform this maintenance task. You can drop all unused columns in one procedure, or you can drop unused columns for individual tables.

### Drop all unused columns on Oracle

### Procedure

1. Create the following Oracle procedure to purge all unused columns:

```
DECLARE
  dropstr VARCHAR2(100);
  CURSOR unusedcol IS
    SELECT table_name
    FROM user_unused_col_tabs;
BEGIN
  FOR tabs IN unusedcol LOOP
    dropstr := 'alter table '
               ||tabs.table_name
               ||' drop unused columns';
    EXECUTE IMMEDIATE dropstr;
  END LOOP;
END;
```

2. Run the procedure during a period of relatively low activity.

## Drop unused columns for a specific table on Oracle

### Procedure

1. Start the server to run the schema verifier. The schema verifier runs each time the server starts. If there are unused columns, the schema verifier reports a difference between the physical database and the data model. The schema verifier reports the name of each table and provides an SQL command to remove unused columns from each table.
2. Run the SQL command provided by the schema verifier. This command has the following format:

```
ALTER TABLE tableName DROP UNUSED COLUMNS
```

## Reload rating sample data

### About this task

If you are using any rating data provided by Guidewire, such as `calcRoutines`, `rateBooks`, `rateTableDefinition`, parameter sets, and so forth, remove all existing rating data, and reload new rating data.

### Procedure

1. Start the PolicyCenter server.
2. Remove the old rating sample data by running the following Gosu script against the database:

```
uses gw.transaction.Transaction
uses gw.api.database.Query

 function findEntity<T extends KeyableBean>() : List<T>{
  var q = Query.make(T)
  q.startsWith("PublicID", "pc:", false /*ignoreCase*/)
  return q.select().toList()
}

 Transaction.runWithNewBundle(\ bundle -> {
  findEntity<RateBook>().each(\ rb -> bundle.add(rb).remove())
  findEntity<RateTableDefinition>().each(\ rt -> bundle.add(rt).remove())
  findEntity<RateTableMatchOpDefinition>().each(\ rb -> bundle.add(rb).remove())
  findEntity<RateFactorRow>().each(\ rb -> bundle.add(rb).remove())
  findEntity<CoverageRateFactor>().each(\ rb -> bundle.add(rb).remove())
  findEntity<CalcRoutineDefinition>().each(\ rb -> bundle.add(rb).remove())
  findEntity<CalcRoutineParameterSet>().each(\ rb -> bundle.add(rb).remove())
}, "su")
```

3. Run the following Gosu script to reload the PolicyCenter 10.0.0 rating sample data:

```
uses gw.transaction.Transaction
uses gw.sampledata.small.SmallSampleRatingData
uses gw.sampledata.tiny.TinySampleRatingData

 Transaction.runWithNewBundle(\ bundle -> {
  var tinyData = new TinySampleRatingData()
  tinyData.load()
  var sampleData = new SmallSampleRatingData()
  sampleData.load()
}, "su")
```

## Export administration data for testing

### About this task

Guidewire recommends that you create a small set of administration data from an upgraded data set. Use this data for development and testing of rules and libraries with PolicyCenter 10.0.0. This procedure is optional.

You might have already created an upgraded administration data set by following the procedure "Upgrade administration data for testing" on page 26. If you followed that procedure, or you do not want an administration-only data set for testing purposes, you can skip this topic.

### Procedure

1. Export administration data from your upgraded production database.
    a. Start the PolicyCenter 10.0.0 server by navigating to `PolicyCenter` and running the following command:

       `gwb runServer`
    b. Open a browser to PolicyCenter 10.0.0.
    c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
    d. Click the **Administration** tab.
    e. Click **Utilities→Export Data**.
    f. Select the **Admin** data set to export.
    g. Click **Export** to download the `admin.xml` file.
2. Create a new database account for the development environment on a database management system supported by PolicyCenter 10.0.0. See the *Supported Software Components* knowledge article for current system and patch level requirements. Visit the Guidewire Community and search for the *Supported Software Components* knowledge article.

   See the *Installation Guide* for instructions to configure the database account.
3. Install a new PolicyCenter 10.0.0 development environment. Connect this development environment to the new database account that you created in "Export administration data for testing" on page 62. See the *PolicyCenter Installation Guide* for instructions.
4. Copy the `admin.xml` file that you exported to a location accessible from the new development environment.
5. Create an empty version of `importfiles.txt` in the `modules/configuration/config/import/gen` directory of the new development environment.
6. Create empty versions of the following CSV files:
    - `activity-patterns.csv`
    - `authority-limits.csv`
    - `reportgroups.csv`
    - `roleprivileges.csv`
    - `rolereportprivileges.csv`

   Leave `roles.csv` as the original complete file.
7. Import the administration data into the new database:
    a. Start the PolicyCenter 10.0.0 development server by navigating to `PolicyCenter` and running the following command:

       `gwb runServer`
    b. Open a browser to PolicyCenter 10.0.0.

c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
d. Click the **Administration** tab.
e. Click **Utilities→Import Data**.
f. Click **Browse...**.
g. Select the `admin.xml` file that you exported from the upgraded production database and modified.
h. Click **Open**.

## Final steps after the database upgrade is complete

After you have completed the upgrade procedure and migration of configurations and integrations, there are a final set of procedures to follow. These procedures provide you with a benchmark of the new system. Completing these steps is particularly important to going live in a production environment.

### Checking that contacts have unique addresses

An `Address` cannot be shared by more than one `Contact`. PolicyCenter 10.0.0 includes a commit-time check that does not allow a shared reference to an address instance even when one of the referring `Contact` or `ContactAddress` instances is retired. If you have multiple contacts at the same address, you can create separate address instances with the same field values.

A database consistency check on the `Contact` entity reports an error if it detects multiple `Contact` records using the same `PrimaryAddress`.

Before using PolicyCenter 10.0.0 in production, run database consistency checks to find any instances of shared references to address instances. If the consistency check reports shared addresses, contact Guidewire Support for assistance fixing your database.

### Changes to upgraded underwriting rules

#### About this task

UWRule no longer supports `util` (lowercase).

#### Procedure

Check your underwriting rules. Any rule that use `util` rather than `Util` will be marked as invalid. If you find such invalid rules (`util` is an unknown symbol), navigate to each rule and edit the rule to use `Util` rather than `util`.

### Completing deferred upgrade

If you have archiving enabled, and you did not set `deferCreateArchiveIndexes` to `false`, run the `Deferred Upgrade Tasks` batch process as soon as possible after the completion of the upgrade. To run the `Deferred Upgrade Tasks` batch process, use the `admin/bin/maintenance_tools` command:

```
maintenance_tools -password password -startprocess deferredupgradetasks
```

### Re-enabling database logging

You might have disabled logging of direct insert and create index operations during the database upgrade. After you complete the database upgrade successfully, you can re-enable logging by setting `allowUnloggedOperations` to `false` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="false">
    ...
  </upgrade>
</database>
```

For SQL Server, if you changed the recovery model from Full to Simple or Bulk logged during the upgrade, you can revert the recovery model. If you deferred migrating to 64-bit IDs, you might disable logging again when you perform the migration.

## Running key batch processes

- `Account Holder Count`

## Enable the scheduler after upgrade

### About this task

Before you perform these steps, verify that the database upgrade process succeeded.

Before you started the server to upgrade the database, you had to disable the scheduler to prevent batch processes and work queues from launching immediately after the database upgrade. After you have verified that the database upgrade process completed successfully, enable the scheduler again so that batch processes and work queues will resume normal a normal schedule.

### To enable the scheduler

### Procedure

1. Stop the server.
2. Create a new WAR or EAR file that differs from the former only by having `SchedulerEnabled` set to `true`.
   a. Open the PolicyCenter 10.0.0 `config.xml` file in a text editor.
   b. Set the `SchedulerEnabled` parameter to `true`, as follows:
      ```
      <param name="SchedulerEnabled" value="true"/>
      ```
   c. Save `config.xml`.
   d. Create a new WAR or EAR file as appropriate for your application server.
3. Deploy the new WAR or EAR file.
4. Restart the server to activate the scheduler.

### See also

*Installation Guide*

## Backing up the database after upgrade

Finally, before going live, back up the upgraded database. This provides you with a snapshot of the initial upgraded data set, if an unanticipated event occurs just after going live.

## Upgrading PolicyCenter from 9.0 for ContactManager

This topic lists the manual tasks required to upgrade from PolicyCenter 9.0 and ContactManager 9.0 to PolicyCenter 10.0.0 and ContactManager 10.0.0.

Guidewire recommends that you first upgrade ContactManager, integrate PolicyCenter with ContactManager, and refresh the ContactManager web APIs.

## Upgrade PolicyCenter to integrate with ContactManager

### About this task

Use the following steps to upgrade PolicyCenter 9.0 with ContactManager installed.

### Procedure

1. Run the Configuration Upgrade tools and complete the configuration upgrade for the PolicyCenter configuration. See the PolicyCenter *Configuration Upgrade Guide*. Do not make changes yet to the files listed in this topic for PolicyCenter. Make those changes later as described in this topic.
2. Run the database upgrade for the PolicyCenter database. See "Upgrading the PolicyCenter 9.0 database" on page 26.
3. You can perform any manual configuration upgrades except those related to files listed later in this topic. Before making those changes, configure ContactManager, regenerate the ContactManager SOAP API, and refresh that API in PolicyCenter Studio.
4. Manually configure ContactManager. See "Upgrading ContactManager from 9.0" on page 65.
5. Make any manual PolicyCenter changes indicated in the next topic.
6. Integrate PolicyCenter and ContactManager as described at the *Guidewire Contact Management Guide*.

## File changes in PolicyCenter related to ContactManager

This topic includes:

- "Changes to AddressBookUID column" on page 65
- "Web service version changes" on page 65
- "Changes to PolicyCenter classes for ContactManager integration" on page 65

### Changes to AddressBookUID column

Entities that use the `AddressBookLinkable` delegate and declare their own `AddressBookUID` columns will have that column removed during the upgrade process.

Further manual changes might be required after this upgrade process has run. The upgrader is not able to detect an `AddressBookLinkable` delegate and an `AddressBookUID` column that are declared in different metadata files. In these cases, you must remove the `AddressBookUID` column from the entity metadata file manually.

#### See also

- *Guidewire Contact Management Guide*

### Web service version changes

PolicyCenter now uses `wsi.remote.gw.webservice.ab.ab900.wsc` to access the ContactManager web service `${ab}/ws/gw/webservice/ab/ab900/abcontactapi/ABContactAPI?wsdl`. See the *Guidewire Contact Management Guide*.

### Changes to PolicyCenter classes for ContactManager integration

For changes to PolicyCenter files used with ContactManager, see the *Guidewire Contact Management Guide*.

## Upgrading ContactManager from 9.0

This topic covers the steps needed to perform an upgrade of ContactManager 9.0 to ContactManager 10.0.0.

## Upgrade the ContactManager configuration

### About this task

The general steps for upgrading ContactManager are as follows.

### Procedure

1. Use the configuration upgrade tools to perform an upgrade of the ContactManager configuration. See *"Upgrading your PolicyCenter configuration"* in the *Configuration Upgrade Guide*.

2. Run the database upgrade tool to upgrade the ContactManager database. See “Upgrading the PolicyCenter 9.0 database” on page 26.
3. Upgrade PolicyCenter as described at “Upgrading PolicyCenter from 9.0 for ContactManager” on page 64.

chapter 4

# Upgrading from 8.0

This part describes how to perform an upgrade from PolicyCenter 8.0 to 10.0.0.

If you are upgrading from PolicyCenter 9.0, see "Upgrading from 9.0" on page 25 instead.

If you are upgrading from PolicyCenter 7.0 or a previous version, see "Upgrading from previous versions" on page 111 instead.

This part includes the following topics:

- "Upgrading the PolicyCenter 8.0 configuration" on page 67
- "Upgrading the PolicyCenter 8.0 database" on page 67
- "Upgrading PolicyCenter from 8.0 for ContactManager" on page 109
- "Upgrading ContactManager from 8.0" on page 110

## Upgrading the PolicyCenter 8.0 configuration

### Download the InsuranceSuite upgrade tools and documentation

#### About this task

Guidewire does not include the InsuranceSuite Upgrade Tools in the PolicyCenter installation.

You can access the latest version of the InsuranceSuite Upgrade Tools and the *PolicyCenter* Configuration Upgrade Guide using the following instructions.

#### Procedure

1. Visit the Guidewire Community:
   - Guidewire Customers - `https://community.guidewire.com`
   - Guidewire Partners - `https://partner.guidewire.com`
2. Select the **Resources** tab.
3. Under **Product Group**, select **InsuranceSuite**.
4. Under **Product**, select **InsuranceSuite Upgrade Tools**.
5. Under **Release**, select the most recent release.
6. Download the software, release notes, and documentation by clicking each corresponding download link.

## Upgrading the PolicyCenter 8.0 database

This topic provides instructions for upgrading the PolicyCenter 8.0 database to PolicyCenter 10.0.0.

You can only upgrade an Oracle or SQL Server database. Guidewire does not support upgrade of the H2 Quickstart development database.

## Upgrade administration data for testing

### About this task

You might want to create an upgraded administration data set for development and testing of rules and libraries with PolicyCenter 10.0.0. You can wait until the full database upgrade is complete and then export the administration data, as described in “Export administration data for testing” on page 62. Or, you can upgrade only the administration data to have this data available earlier in the upgrade process. Use the procedure in this section to create an upgraded administration data set before upgrading the full database.

### Procedure

1. Export administration data from your current (pre-upgrade) PolicyCenter production instance:
   a. Log on to PolicyCenter as a user with the `viewadmin` and `soapadmin` permissions.
   b. Click the **Administration** tab.
   c. Choose **Import/Export Data**.
   d. Select the **Export** tab.
   e. Select **Admin** from the **Data to Export** dropdown.
   f. Click **Export**. PolicyCenter exports an `admin.xml` file.
2. In a new base version development environment based on your production configuration, create empty files in the in the `modules/configuration/config/import/gen` directory:
   a. Create an empty version of `importfiles.txt`
   b. Create empty versions of the following CSV files:
   - `activity-patterns.csv`
   - `authority-limits.csv`
   - `reportgroups.csv`
   - `roleprivileges.csv`
   - `rolereportprivileges.csv`
     - Leave `roles.csv` as the original complete file.
3. Start the development environment server by opening a command prompt to `PolicyCenter` and entering the following command:

```
gwb runServer
```

4. Import this administration data into the development environment.
   a. Log on to PolicyCenter as a user with the `viewadmin` and `soapadmin` permissions.
   b. Click the **Administration** tab.
   c. Choose **Import/Export Data**.
   d. Select the **Import** tab.
   e. Click **Browse...**.
   f. Select the `admin.xml` file that you exported in “Upgrade administration data for testing” on page 26.
   g. Click **Open**.
5. Create a backup of the new development environment database.
6. Create a new database account for the development environment on a database management system supported by PolicyCenter 10.0.0.
   - See the *Supported Software Components* knowledge article for current system and patch level requirements. Visit the Guidewire Community and search for the *Supported Software Components* knowledge article.
   - See the *Installation Guide* for instructions to configure the database account.

7. Restore the backup of the database containing the imported administration data into the new database.
8. Connect your upgraded target PolicyCenter 10.0.0 configuration to the restored database.
9. Start the PolicyCenter 10.0.0 server to upgrade the database.
10. Export the upgraded administration data:
    a. Start the PolicyCenter 10.0.0 server by navigating to `PolicyCenter` and running the following command:
    ```
    gwb runServer
    ```
    b. Open a browser to PolicyCenter 10.0.0.
    c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
    d. Click the **Administration** tab.
    e. Choose **Import/Export Data**.
    f. Select the **Export** tab.
    g. For **Data to Export**, select **Admin**.
    h. Click **Export**.
       Your browser will note that you are opening a file and will prompt you to save or download the file.
    i. Download the `admin.xml` file.
       You can import this XML file into local development environments of PolicyCenter 10.0.0.

## Identify data model issues

### About this task

Before you upgrade a production database, identify issues with the data model by running the database upgrade on an empty database. This process does not identify all possible issues. The database upgrade does not detect issues caused by specific data in your production database. Instead, this procedure identifies issues with the data model.

Complete the following procedure to identify data model issues, and correct any issues on an empty schema. Then, follow the full list of procedures in this topic to upgrade a production database. This list begins with “Verify batch process and work queue completion” on page 28 and finishes with “Final steps after the database upgrade is complete” on page 107.

### To identify data model issues

### Procedure

1. Create an empty schema of your starting version database. You can do this in a development environment by pointing the development PolicyCenter installation at an empty schema and starting the PolicyCenter server. See the *Installation Guide*.
2. Complete the configuration upgrade for data model files in your starting version, according to the instructions in “Upgrading the PolicyCenter 8.0 configuration” on page 67. You do not need to complete the merge process for all files.
3. Configure your upgraded development environment to point to the database account containing the empty schema of your old version. See the *Installation Guide*.
4. Start the PolicyCenter server in your upgraded development environment. The server performs the database upgrade to PolicyCenter 10.0.0. See “Starting the server to begin automatic database upgrade” on page 56.
5. Check for errors reported during the upgrade process. Resolve any issues before upgrading your production database. You can use the `IDatabaseUpgrade` plugin to run custom SQL before and after the database upgrade. For more information, see “Creating custom upgrade version checks and version triggers” on page 35.

## Verify batch process and work queue completion

### About this task

All batch processes and work queues must complete before beginning the upgrade. Check the status of batch processes and work queues in your current production environment.

### Procedure

1. Log in to PolicyCenter as superuser.
2. Press `Alt` + `Shift` + `T`. PolicyCenter displays the **Server Tools** tab.
3. Click **Batch Process Info**.
4. Select **Any** from the **Processes** drop-down filter.
5. Click **Refresh**.
6. Check the **Status** column for each batch process listed. This list also includes batch processes that are writers for distributed work queues.

   If any of the batch processes have a **Status** of **Active**, wait for the batch process to complete before continuing with the upgrade.

## Purging data prior to upgrade

Purging certain types of data from the database prior to upgrade can improve the performance of the database upgrade and PolicyCenter.

## Purge old messages from the database

### About this task

Purge completed inactive messages before upgrading the database. Doing so improves performance and reduces the complexity of the database upgrade.

You cannot resend old messages after the upgrade. This is because integrations change and the message payload might be different. It is important that messages that have failed or not yet been consumed finish prior to upgrading.

Use one the following methods from the base customer configuration to purge completed messages from the `pc_MessageHistory` table:

### To purge old messages

### Procedure

1. The `messaging_tools` command from the `admin/bin` directory of base customer configuration. This tool deletes completed messages with a send time before the date *MM*/*DD*/Y*YYY*: `messaging_tools -password` *`password`* `-server http://`*`server`*`:`*`port`*`/`*`instance`* `-purge` *`MM/DD/YYYY`*
2. The `purgeCompletedMessages` web service API: `IMessageToolsAPI.purgeCompletedMessages(java.util.Calendar cutoff)`

### Next steps

After you purge completed inactive messages, reorganize the `pc_MessageHistory` table. You might also want to rebuild any indexes on the table. Contact Guidewire Support if you need assistance.

## Purging completed workflows and workflow logs

Each time PolicyCenter creates an activity, the activity is added to the `pc_Workflow`, `pc_WorkflowLog` and `pc_WorkflowWorkItem` tables. Once a user completes the activity, PolicyCenter sets the workflow status to completed. The `pc_Workflow`, `pc_WorkflowLog` and `pc_WorkflowWorkItem` table entry for the activity are never used again. These tables grow in size over time and can adversely affect performance as well as waste disk space. Excessive records in these tables also negatively impacts the performance of the database upgrade.

Remove workflows, workflow log entries, and workflow items for completed activities to improve database upgrade and operational performance and to recover disk space.

Use the `purgeworkflows` batch process to purge older completed workflows and their logs. Guidewire recommends that you purge completed workflows and their logs periodically. This reduces performance issues caused by having a large number of unused workflow log records.

Alternatively, you can purge only the logs associated with completed workflows by running the `purgeworkflowlogs` batch process. This batch process leaves the workflow records and removes only the workflow log records.

### Purge completed workflows and associated logs

#### Procedure

1. Set the number of days after which the `purgeworkflows` process purges completed workflows and their logs, by setting the `WorkflowPurgeDaysOld` parameter in `config.xml`. By default, `WorkflowPurgeDaysOld` is set to `60`. This is the number of days since the last update to the workflow, which is the completed date.

   ```
   <param name="WorkflowPurgeDaysOld" value="60" />
   ```

2. Launch the **Purge Workflows** batch process from the `PolicyCenter/admin/bin` directory with the following command:

   ```
   maintenance_tools -password password -startprocess PurgeWorkflows
   ```

### Purge completed workflow logs only

#### Procedure

1. Set the number of days after which the `purgeworkflowlogs` process purges completed workflow logs, by setting the `WorkflowLogPurgeDaysOld` parameter in `config.xml`. This is the number of days since the last update to the workflow, which is the completed date.

   ```
   <param name="WorkflowLogPurgeDaysOld" value="60" />
   ```

2. Launch the **Purge Workflow Logs** batch process from the `PolicyCenter/admin/bin` directory with the following command:

   ```
   maintenance_tools -password password -startprocess PurgeWorkflowLogs
   ```

## Validate the database schema

#### About this task

This validation detects the unlikely event that the data model defined by your configuration files has become out of sync with the database schema.

#### Procedure

1. While the pre-upgrade server is running, use the `system_tools` command in `admin/bin` of the customer configuration to verify the database schema:

   ```
   system_tools -password password -verifydbschema -server servername:port/instance
   ```

2. Correct any validation problems in the database before proceeding. Contact Guidewire Support for assistance.

### Next steps

Following the database upgrade, run this command again from the `admin/bin` directory of the target (upgraded) configuration.

## Checking database consistency

PolicyCenter has hundreds of internal database consistency checks. Before upgrading, run consistency checks to verify the integrity of your data.

Run database consistency checks early in the upgrade project. Fix any consistency errors. Continue to periodically run consistency checks and resolve issues so that your database is ready to upgrade when you begin the upgrade procedure. Consistency issues might take some time to resolve, so begin the process of running consistency checks and fixing issues early. Contact Guidewire Support for information on how to resolve any consistency issues.

After the database upgrade, run consistency checks again from the PolicyCenter **Consistency Checks** page.

### Run database consistency checks

#### Procedure

1. Start the PolicyCenter server if it is not already running.
2. Log in to PolicyCenter with an administrator account.
3. Press `Alt` + `Shift` + `T` to access the **Server Tools**.
4. Click **Info Pages**.
5. Select **Consistency Checks** from the drop-down list.
6. To increase the number of threads used to run consistency checks, increase the **Number of threads**. The number of threads to use depends on the capability of your database server. Increasing the number of threads can improve performance of consistency checks as long as your server can process the threads. Guidewire recommends starting with five threads. If too many threads are used, there is a greater chance that current users experience reduced performance if the database server is fully loaded.

   To set the number of threads in versions prior to 8.0, specify a value for the `checker.threads` parameter within the database block of `config.xml`.

   ```
   <database
     ...
     <param name="checker.threads" value="5" />
     ...
   </database>
   ```
7. Click **Run Consistency Checks**.

#### See also

*System Administration Guide*.

## Create a data distribution report

#### About this task

Generate a data distribution report for the database before an upgrade. Save the output of this report. Run the report again after the upgrade to ensure the distribution is still correct.

Guidewire is very interested in the data distribution of your databases. Guidewire uses these reports to better understand the nature of your database and to optimize PolicyCenter performance. Guidewire appreciate copies of your reports, both before and after upgrades.

You can also use this information to tune the application server cache. See the *System Administration Guide*.

### To create a database distribution report

#### Procedure

1. In `config.xml`, set `<param name="EnableInternalDebugTools" value="true"/>`.
2. Start the PolicyCenter application server.
3. Log into PolicyCenter as an administrative user.
4. Type `ALT` + `SHIFT` + `T` while in any screen to reach the **Server Tools** page.
5. Choose **Info Pages** from the **Server Tools** tab.
6. Choose the **Data Distribution** page from the **Info Pages** dropdown.
7. Enter a reason for running the Data Distribution batch job in the **Description** field.
8. On this page, select the **Collect distributions for all tables** radio button and check all checkboxes to collect all distributions.
9. Push the **Submit Data Distribution Batch Job** button on this page to start the data collection.
10. Return to the **Data Distribution** page and push its **Refresh** button to see a list of all available reports. The batch job has completed when the **Available Data Distribution** list on the **Data Distribution** page includes your description.
11. Select the desired report and use the **Download** button to save it zipped to a text file. Unzip the file to view it.

## Generating database statistics

You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded PolicyCenter in a production environment.

To optimize the performance of the PolicyCenter database, it is a good idea to update database statistics on a regular basis. Both SQL Server and Oracle can use these statistics to optimize database queries.

If you update database statistics on a regular basis, you do not need to update statistics before an upgrade. If you do not update database statistics on a regular basis, Guidewire recommends that you update incremental statistics before running the upgrade.

### Generate incremental database statistics

#### Procedure

1. Get the proper SQL statements for updating the statistics in PolicyCenter tables by running the following command:

```
system_tools -password password -getincrementaldbstatisticsstatements -server
  http://server:port/instance > db_stats.sql
```

2. Run the resulting SQL statements against the PolicyCenter database.

#### Result

You can configure SQL Server to periodically update statistics using SQL. See your database documentation and the *System Administration Guide* for more information.

The database upgrade can take a long time, and has built-in statistics collection that help you see if any part of the upgrade is particularly slow. Collect these statistics, and compare them to the statistics you collected before the upgrade. The `config.xml` file has parameters that control this statistics collection.

### Generate full statistics after upgrade is complete

For Oracle, Guidewire recommends that you generate full statistics as soon as possible after the upgrade, during the next maintenance window. Do this even if you did not disable statistics collection during the upgrade by setting `updatestatistics` to `false`.

For SQL Server, this step is only necessary if you disabled statistics collection during the upgrade by setting `updatestatistics` to `false`. For this case, Guidewire recommends that you generate full statistics as soon as possible after the upgrade.

#### See also

*System Administration Guide*.

## Back up your database

### About this task

The first time you start the PolicyCenter server after running the upgrade tools, the server updates the database. During upgrade, PolicyCenter minimizes logging. For these reasons, your database might not be recoverable if the upgrade process is runs into problems.

Back up your database before starting an upgrade. Consult the documentation for your database management system for tools and techniques to back up the database.

## Update database infrastructure

### About this task

Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of PolicyCenter 10.0.0.

See the *Supported Software Components* knowledge article for current system and patch level requirements. Visit the Guidewire Community and search for the *Supported Software Components* knowledge article.

## Prepare the database for upgrade

### About this task

Do the following to prepare the database for the upgrade process:

### Procedure

1. **Ensure Adequate Free Space** – The database upgrade requires significant free space. Make sure the database has at least 50% of the current database size available as free space.
2. **Disable Replication** – Disable database replication during the database upgrade.
3. **Assign Default Tablespace (Oracle only)** – Set the default tablespace for the database user to the one mapped to the logical tablespace `OP` in `config.xml`.

   The database upgrade creates temporary tables during the upgrade without specifying the tablespace. If the Oracle database user was created without a default tablespace, Oracle by default creates the tables in the SYSTEM tablespace. The Guidewire database user is likely not to have the required quota permission on the SYSTEM tablespace. This results in an error of the type:

   java.sql.SQLException: ORA-01950: no privileges on tablespace 'SYSTEM'

   Even if the default tablespace is not SYSTEM, if the Guidewire database user does not have quota permission on the default tablespace, the temporary table creation during upgrade fails.
4. **Exclude tables from Change Data Capture (CDC) (SQL Server only)** – If Change Data Capture (CDC) is enabled, Guidewire recommends that you exclude some tables before upgrade.

   You can exclude the following tables from CDC:

   a. All entities with eti files that have `instrumentationtable="true"`
   b. `pc_batchprocesslease`
   c. `pc_batchprocessleasehistory`
   d. `pc_destinationlease`
   e. `pc_destinationleasehistory`
   f. `pc_messagerequestlease`
   g. `pc_messagerequestleasehistory`
   h. `pc_pluginlease`

   i. `pc_pluginleasehistory`

5. **Remove `adaptive-optimization` element from `database-config.xml` (Oracle only)**– If you applied Oracle patch 22652097, remove the `adaptive-optimization` element from `database-config.xml`.

   Keep the default `init.ora` adaptive parameters:

   - `optimizer_adaptive_statistics` (default `false`)
   - `optimizer_adaptive_plans` (default `true`)

## Set linguistic search collation

**WARNING** For SQL Server, compare the default collation of the database to the collation defined for your locale. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your SQL Server database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This attribute change results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, this adds time to the total database upgrade process.

**WARNING** Oracle Java Virtual Machine (JVM) must be installed on all Oracle databases hosting PolicyCenter. The only exception is when the PolicyCenter application locale is English and you only require case-insensitive searches. Ensure that Oracle initialization parameter `java_pool_size` is set to a value greater than 50 MB.

You can specify how you want PolicyCenter to collate search results. The `strength` attribute of the `LinguisticSearchCollation` element of `GWLanguage` in `language.xml` in the currently selected region specifies how PolicyCenter sorts search results. You can set the `strength` to `primary` or `secondary`.

With `LinguisticSearchCollation strength` set to `primary`, PolicyCenter searches results in a case-insensitive and accent-insensitive manner. PolicyCenter considers an accented character equal to the unaccented version of the character if the `LinguisticSearchStrength` for the default application locale is set to `primary`. For example, with `LinguisticSearchCollation strength` set to `primary`, PolicyCenter treats “Reneé”, “Renee”, “renee” and “reneé” the same.

With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter searches results in a case-insensitive, accent-sensitive manner. PolicyCenter does not consider an accented character equal to the unaccented version of the character if the `LinguisticSearchCollation strength` for the default application locale is set to `secondary`. For example, with `LinguisticSearchCollation strength` set to `secondary`, a PolicyCenter search treats “Renee” and “renee” the same but treats “Reneé” and “reneé” differently. By default, PolicyCenter uses a `LinguisticSearchCollation strength` of `secondary`, which specifies case-insensitive, accent-sensitive searching.

The `collations.xml` file defines the collations to use for different locales and different collation strengths. The `primary`, `secondary`, and `tertiary` attributes of the `Collation` element define the collation to use depending on the `LinguisticSearchCollation strength` attribute in `language.xml`.

PolicyCenter 7.0 introduced configurable linguistic searching for SQL Server databases. In releases prior to PolicyCenter 7.0, PolicyCenter used the collation setting of the database server. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This attribute change results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, dropping and recreating indexes adds time to the total database upgrade process.

For sorting search results, the following rules apply:

#### Case

All searches ignore the case of the letters, whether `LinguisticSearchCollation strength` is set to `primary` or `secondary`. "McGrath" equals "mcgrath".

**Punctuation**

Punctuation is always respected, and never ignored. “O'Reilly” does not equal “OReilly”.

**Spaces**

Spaces are respected. "Hui Ping" does not equal "HuiPing".

**Accents**

An accented character is considered equal to the unaccented version of the character if `LinguisticSearchCollation strength` is set to `primary`. An accented character is not equal to the unaccented version if `LinguisticSearchCollation strength` is set to `secondary`.

Japanese only:

**Half Width/Full Width**

Searches under a Japanese region always ignore this difference.

**Small/Large Kana**

Japanese small/large letter differences are ignored only when `LinguisticSearchCollation strength` is set to `primary`, meaning accent-insensitive.

**Katakana/Hiragana sensitivity**

Searches under a Japanese region always ignore this difference.

**Long dash character**

Always ignored.

**Soundmarks ( `` and ° )**

Only ignored if `LinguisticSearchCollation strength` is set to `primary`.

German only

**Vowels with an umlaut**

Compare equally to the same vowel followed by the letter e. Explicitly, “ä”, “ö”, “ü” are treated as equal to “ae”, “oe” and “ue”.

**The Eszett, or sharp-s, character "ß"**

Treated as equal to “ss”.

PolicyCenter populates denormalized values of searchable columns to support the search collation. For example, with `LinguisticSearchCollation strength` set to `primary`, PolicyCenter stores the value “Reneé”, “Renee”, “renee” and “reneé” in a denormalized column as “renee”. With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter stores a denormalized value of “renee” for “Renee” or “renee” and stores “reneé” for “Reneé” or “reneé”. Japanese and German locales make additional changes when storing values in denormalized columns in order to conform to the rules listed previously for those locales.

Any time you change the `LinguisticSearchCollation strength` and restart the server, PolicyCenter repopulates the denormalized columns. Previous versions of PolicyCenter populated the denormalized columns with lowercase values for case-insensitive search, equivalent to setting `LinguisticSearchCollation strength` to `secondary`. If you set `LinguisticSearchCollation strength` to `primary`, PolicyCenter repopulates the denormalized columns, substituting any accented characters for their base equivalents. This process can take a long time, depending on the amount of data. Therefore, if you want to change `LinguisticSearchCollation strength` to `primary`, you might want to do so after the database upgrade. If you are concerned about the duration of the database upgrade, you can change your search collation settings after the upgrade. During a maintenance period, change `LinguisticSearchCollation strength` to `primary` and restart the server to repopulate the denormalized columns.

For Japanese locales, the PolicyCenter database upgrade from a prior major version repopulates the denormalized columns regardless of the `LinguisticSearchCollation strength` value. PolicyCenter must repopulate the denormalized columns for Japanese locales to have search results obey the Japanese-only rules listed previously.

For more information, see in the *Globalization Guide*.

## Customizing the database upgrade

The `IDatamodelUpgrade` plugin interface provides hooks for custom code that you want to run during the database upgrade. You can implement the `IDatamodelUpgrade` plugin to:

- Execute custom version checks to test data or the data model itself before starting the upgrade.
- Make custom database changes before or after the database upgrade.
- Make data model changes to archived entities.

For example, you might fix a consistency check failure issue, correct issues reported by version checks, or delete a custom extension that you are no longer using.

**IMPORTANT** PolicyCenter 4.0 included a similar plugin interface, `IDatabaseUpgrade`. If you previously implemented `IDatabaseUpgrade` for an upgrade to PolicyCenter 4.0, you must now implement `IDatamodelUpgrade` if you want to execute custom upgrade code.

### Creating custom upgrade version checks and version triggers

Use the `IDatamodelUpgrade` plugin to run custom version checks and triggers before and after the database upgrade. The `IDatamodelUpgrade` plugin interface defines method signatures for two methods that you must define in your plugin. These signatures are:

- `property getBeforeUpgradeDatamodelChanges() : List<IDataModelChange<BeforeUpgradeVersionTrigger>>`
- `property getAfterUpgradeDatamodelChanges() : List<IDataModelChange<AfterUpgradeVersionTrigger>>`

Each method returns a list of `IDataModelChange` entities, each taking a `BeforeUpgradeVersionTrigger` or `AfterUpgradeVersionTrigger` type parameter. The `IDataModelChange` interface has two methods that you use to make data model changes. The `getDatabaseUpgradeVersionTrigger` method is for changes to the database. The `getArchivedDocumentUpgradeVersionTrigger` method is for changes to archived entities. If your organization has not implemented archiving or you do not want to make changes to archived entities, return null for `getArchivedDocumentUpgradeVersionTrigger`.

The `getAfterUpgradeDatamodelChanges` method runs after the Guidewire upgrade version triggers. You can use this method to move data into extension tables or columns that did not exist prior to upgrading.

You can return an empty list from either `getBeforeUpgradeDatamodelChanges` or `getAfterUpgradeDatamodelChanges`. For example, if you only have triggers to run before the upgrade, you can return an empty list from `getAfterUpgradeDatamodelChanges`.

## Modifying tables using a custom version trigger

Both `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` base classes provide a protected `getTable` method that accepts a `string` parameter. The `getTable` method returns an `IBeforeUpgradeTable` or `IAfterUpgradeTable` object that provides a number of methods for DDL and DML operations, such as:

- `create` – Create the table if it does not already exist. The table must be related to an entity defined in the data model. This method is available only for `IBeforeUpgradeTable`.
- `delete` – Deletes rows from a table. Returns a builder (`IBeforeUpgradeDeleteBuilder` for `IBeforeUpgradeTable`, `IDeleteBuilder` for `IAfterUpgradeTable`) that has methods for comparing data to restrict which rows are deleted.
- `drop` – Drops the table.
- `dropColumns` – Drops multiple columns from the table.
- `getColumn` – Returns an `IBeforeUpgradeColumn` or `IAfterUpgradeColumn` object that has methods to perform DDL operations on the column such as create, drop, rename, and more.
- `insert` – Returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable`) to perform an insert operation.
- `insertSelect` – Returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`) for SQL to perform an insert operation using data selected from a table.
- `rename` – Renames the table.
- `update` – Returns a builder (`IBeforeUpgradeUpdateBuilder` for `IBeforeUpgradeTable`, `IUpdateBuilder` for `IAfterUpgradeTable`) for SQL to perform an update operation.

For DML operations, call the `execute` method on the builder to actually perform the operation. The `execute` method runs in its own transaction. You do not need to handle transactions and `TransactionManager`.

There are more methods on the `IBeforeUpgradeTable` and `IAfterUpgradeTable` classes documented in the Guidewire Gosu API documentation. To generate the Guidewire Gosu API documentation, run the `gwb gosudo` command from the PolicyCenter installation directory. Then, open `PolicyCenter/build/gosudoc/index.html`.

The methods for `BeforeUpgradeVersionTrigger` intentionally take strings but not entities or properties. This is because the name of the column could change in the future. Consider `PropertyA` on `EntityE` which corresponds to column `A` in the database. Suppose you use `PROPERTYA_PROP` in a version trigger at minor version 200, but at minor version 250, you decide to rename the backing column from `A` to `B`. The version trigger you wrote in the past would break because it would execute before the rename operation and would try to use the new column name.

`AfterUpgradeVersionTrigger` is very similar to `BeforeUpgradeVersionTrigger`. A few differences include:

- The `AfterUpgradeVersionTrigger` DML builders use the query builder, `IQueryBuilder`.
- In an `AfterUpgradeVersionTrigger` you can use properties and types in addition to strings.
- Some DDL operations are not provided on the `IAfterUpgradeTable` object, including creating a table or adding a column.

**Note:** Guidewire recommends you use `BeforeUpgradeVersionTrigger` for custom version triggers a unless you have a special case requiring one of the unique capabilities of `AfterUpgradeVersionTrigger`. `BeforeUpgradeVersionTrigger` has many more capabilities than `AfterUpgradeVersionTrigger`. One example of a limitation off `AfterUpgradeVersionTrigger` is that you cannot set `MonetaryAmount` fields from an `AfterUpgradeVersionTrigger`.

## Upgrading typelists using a custom version trigger

The `BeforeUpgradeVersionTrigger` class includes a `getTypeKeyID` method with the following signature:

```
protected final Integer getTypeKeyID(IEntityType subtype)
```

**Note:** Protected methods do not appear in the Gosu documentation. Use `CTRL` + `SPACE` in Studio to show available methods and properties.

The `getTypeKeyID` method returns the integer ID of the type code in the type list matching the given table name. This method checks both the existing typelist tables and the metadata files to determine what all typekey IDs will be after upgrade. Therefore, the `getTypeKeyID` method works as expected even before a new typekey or typelist table is created during the automatic schema upgrade phase.

This method also works for orphaned typecodes that have not yet been removed from the database. These are typecodes that still exist in the database table but not in the metadata file. You can use the `getTypeKeyID` method for remapping usages of orphaned typecodes.

### Changing the nullability of subtype columns after a database upgrade

For entity definitions, an automatic database upgrade converts nullable columns to non-nullable columns. For the converted columns, the upgrade process sets a default value successfully.

However, the automatic database upgrade process does not convert column nullability for columns in subtype definitions. Instead, PolicyCenter implements non-nullable columns on subtypes in the database as nullable columns. The product behaves this way because columns must have `null` values for rows that represent instances of other subtypes.

To make a subtype column non-nullable, write a custom version trigger. Program the version trigger to populate the column with an appropriate default value for existing subtype rows. After you perform this step, PolicyCenter makes new column values for the subtype non-nullable. The product does this by setting the default value for new subtype rows automatically.

See “Setting a column value for a specific subtype using a version trigger” on page 44 and “Create a custom version check or version trigger” on page 38 for more information on creating a version trigger in this manner.

### Version checks

You can check for a certain condition in the database before the upgrade proceeds. This is referred to as a version check. Only read operations are available in version checks. For example, you can implement a version check to query a table or check the existence of a table or column, but the check cannot insert new rows. The `BeforeUpgradeVersionTrigger` class includes a `hasVersionCheck` method that you must define to return true or false. If the trigger does include a version check, overwrite the `createVersionCheck` method to define your custom version check. For standalone version checks that are not associated with a version trigger, you can use `BeforeUpgradeVersionCheckWrapper`.

The upgrade executes all custom version checks before custom version triggers. The upgrade runs Guidewire version checks after all custom `BeforeUpgradeVersionTrigger` implementations, so you can create a `BeforeUpgradeVersionTrigger` to correct issues detected by the Guidewire version checks.

If a custom version check fails, the upgrade stops before running any upgrade triggers. Correct the issue and restart the upgrade.

### Order of execution

The following table describes failure cases that are caused by data issues. If the upgrade fails for other reasons, such as a disruption of the database server, fix the issue causing the disruption, restore the database, and restart the upgrade. During an upgrade, PolicyCenter performs database upgrade actions in the following order:

| Step | Action | In the event of failure due to a data issue... |
|---|---|---|
| 1 | Custom version checks | Correct the data issue. Restart the upgrade. You do not need to restore the database because the upgrade has not made any changes. |
| 2 | Custom `BeforeUpgradeVersionTrigger` implementations | Restore the database from a backup. Correct the data issue. Consider adding custom version checks to test for other instances of the data issue. |
| 3 | Guidewire version checks | If you do not have any custom `BeforeUpgradeVersionTrigger` implementations, correct the data issue and restart the upgrade.<br>If you do have custom `BeforeUpgradeVersionTrigger` implementations, restore the database from a backup. Then, correct the data issue.<br>In either case, consider creating a custom `BeforeUpgradeVersionTrigger` implementation to correct the data issue. |

| Step | Action | In the event of failure due to a data issue... |
|---|---|---|
| 4 | Guidewire version triggers | A failure due to data issues at this stage is unlikely. Contact Guidewire Support. |
| 5 | Automated data model upgrade to update the database to the defined data model. | A failure due to data issues at this stage is unlikely. Contact Guidewire Support. |
| 6 | Guidewire version triggers that require the updated data model in the database | A failure due to data issues at this stage is unlikely. Contact Guidewire Support. |
| 7 | Custom `AfterUpgradeVersionTrigger` implementations | Restore the database from a backup. Correct the data issue. Consider creating a custom `BeforeUpgradeVersionTrigger` implementation to correct the data issue if possible. |

## Create a custom version check or version trigger

### About this task

You can implement a `IDatamodelUpgrade` plugin with custom version triggers and version checks. When you start the server to perform the database upgrade from an earlier release, PolicyCenter calls the plugin and runs your custom methods.

Each `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` instance requires a minor version number, passed as an integer. If the data model version number is less than or equal to the number passed to the instance, then the trigger executes. Whenever you make a data model change, or you want to force an upgrade, increment the version number in `extensions.properties`.

### To create custom version checks and triggers

### Procedure

1. Create a new package, such as *`companyName`*`.upgrade`, to store your custom version triggers.
   a. Open Studio.
   b. In the Studio **Project** window, expand **configuration**.
   c. Right-click **gsrc** and click **New→Package**.
   d. Enter a package name for upgrade purposes, such as *`companyName`*`.upgrade`.
2. Right-click the upgrade package and click **New→Gosu Class**.
3. Enter a name for the class and click **OK**.
4. Create a new Gosu class that extends `CustomerDatamodelUpgrade` and implements `IDatamodelUpgrade`. The class you create must define the `getBeforeUpgradeDatamodelChanges` and `getAfterUpgradeDatamodelChanges` methods. This class is the container from which you call custom version trigger classes.

   For example:

```
package companyName.upgrade
uses gw.plugin.upgrade.IDatamodelUpgrade
uses java.lang.Iterable
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger
uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger
uses java.util.ArrayList
uses gw.api.datamodel.upgrade.CustomerDatamodelUpgrade
uses gw.api.datamodel.upgrade.IDatamodelChange
uses gw.api.database.upgrade.DatamodelChangeWithoutArchivedDocumentChange

 class TestDatamodelUpgradeImpl extends CustomerDatamodelUpgrade implements IDatamodelUpgrade {

   override property get BeforeUpgradeDatamodelChanges() :
  List<IDatamodelChange<BeforeUpgradeVersionTrigger>> {

     var list = new ArrayList<IDatamodelChange<BeforeUpgradeVersionTrigger>>()
    list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger1()))
    list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger2()))
    return list
  }

   override property get AfterUpgradeDatamodelChanges() :
  List<IDatamodelChange<AfterUpgradeVersionTrigger>> {
    var list = new ArrayList<IDatamodelChange<AfterUpgradeVersionTrigger>>()
    list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new AfterVersionTrigger1()))
    return list
  }
}
```

5. Create your custom `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` Gosu classes. See "IDatamodelUpgrade API examples" on page 39.
6. Implement the `IDatamodelUpgrade` plugin with the new class.
   a. Start Guidewire Studio 10.0.0 by entering `gwb studio` from the `PolicyCenter` directory.
   b. In Studio, expand **configuration→config→Plugins**.
   c. Right-click **registry** and click **New→Plugin**.
   d. In the **Plugin** dialog, enter the name `IDatamodelUpgrade`. For this plugin, the name must match the interface.
   e. In the **Plugin** dialog, click the ... button.
   f. In the **Select Plugin Class** dialog, type `IDatamodelUpgrade` and select the **IDatamodelUpgrade** interface.
   g. In the **Plugin** dialog, click OK. Studio creates a GWP file under **Plugins→registry** with the name you entered.
   h. Click the **Add Plugin** icon (a plus sign) and select **Add Gosu Plugin**.
   i. For **Gosu Class**, enter your class, including the package.
   j. Save your changes.

## IDatamodelUpgrade API examples

The `IDatamodelUpgrade` plugin interface provides hooks for custom code that you want to run during the database upgrade. You can implement the `IDatamodelUpgrade` plugin to:

- Execute custom version checks to test data or the data model itself before starting the upgrade.
- Make custom database changes before or after the database upgrade.
- Make data model changes to archived entities.

### See also

"Creating custom upgrade version checks and version triggers" on page 35

The following topics contain examples of how you can customize your database upgrade by implementing an `IDatamodelUpgrade` plugin:

- "BeforeUpgradeVersionTrigger Structure" on page 40
- "AfterUpgradeVersionTrigger Structure" on page 41
- "Altering columns to match the data model using a version trigger" on page 41
- "Alter a non-nullable column to nullable using a version trigger" on page 42
- "Creating columns using a version trigger" on page 42
- "Dropping columns using a version trigger" on page 43
- "Renaming columns using a version trigger" on page 44
- "Setting a column value for a specific subtype using a version trigger" on page 44
- "Creating tables using a version trigger" on page 44
- "Renaming tables using a version trigger" on page 45
- "Deleting rows using a version trigger" on page 45
- "Inserting rows using a version trigger" on page 45
- "Inserting data selected from another table using a version trigger" on page 46
- "Updating rows using a version trigger" on page 46

## BeforeUpgradeVersionTrigger Structure

A custom `BeforeUpgradeVersionTrigger` subclass has the following structure. Define the `execute` method to perform your custom trigger actions.

```
package companyName.upgrade.before

 uses gw.api.database.upgrade.before.BeforeUpgradeVersionCheck
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger

 class myBeforeUpgradeTrigger extends BeforeUpgradeVersionTrigger {

   construct() {
    super(dataModelVersionNumber)
  }

   override function execute() {
    // Perform actions here.
  }

   override function hasVersionCheck() : boolean {
    // return true if creating a version check to determine whether the trigger can run.
    // return false if you are not implementing a version check.
  }

   override property get Description() : String {
    return "Description of the version trigger."
  }

   // Override the createVersionCheck method if you are implementing a version check.
  override function createVersionCheck() : BeforeUpgradeVersionCheck {
    return new BeforeUpgradeVersionCheck(dataModelVersionNumber) {

       override function verifyUpgradability() {
        if (condition to detect) {
          addVersionCheckProblem("description of issue")
        }
      }

       override property get Description() : String {
        return "Description of the version check."
      }
    }
  }
}
```

### AfterUpgradeVersionTrigger Structure

A custom `AfterUpgradeVersionTrigger` subclass has the following structure.

```
package companyName.upgrade.after

 uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger

 class myAfterUpgradeTrigger extends AfterUpgradeVersionTrigger{

   construct() {
    super(dataModelVersionNumber)
  }

   override function execute() {
    // Perform actions here.
  }

   override property get Description() : String {
    return "Description of the version trigger."
  }

 }
```

### Altering columns to match the data model using a version trigger

In most cases, you do not need to alter a column to match a change to the column type in the logical data model. PolicyCenter automatically applies data model changes to the database during the upgrade. However, this occurs *after* all custom `BeforeUpgradeVersionTrigger` instances have run, so Guidewire provides methods to alter database columns to match the data model.

### Modify a single column using a version trigger

#### About this task

If you need to modify a single column for use in a `BeforeUpgradeVersionTrigger,` use the following procedure.

#### To modify a single column

#### Procedure

1. Modify the data model file.
2. Use the `alterColumnTypeToMatchDatamodel` method of `IBeforeUpgradeColumn` to make your changes.

#### Example

For example:

```
var table = getTable("TableName")
var column = table.getColumn("ColumnName")
column.alterColumnTypeToMatchDatamodel()
```

### Modify multiple columns using a version trigger

#### About this task

If you need to modify more than one column for use in a `BeforeUpgradeVersionTrigger,` use the following procedure.

### To modify multiple columns

#### Procedure

1. Modify the data model file.
2. Use the `alterMultipleColumnsToMatchDatamodel` method of `IBeforeUpgradeTable`.

   For example:

```
                        var table = getTable("TableName")
                        var columnsToChange = new IBeforeUpgradeColumn[2]

                         columnsToChange[0] = table.getColumn("column1")
                        columnsToChange[1] = table.getColumn("column1")

                         table.alterMultipleColumnsToMatchDatamodel(columnsToChange)
```

## Alter a non-nullable column to nullable using a version trigger

### About this task

To alter a column from non-nullable to nullable, use the `IBeforeUpgradeColumn` method `alterColumnToNullable`.

### Example

For example:

```
var table = getTable("TableName");

table.getColumn("ColumnName").alterColumnToNullable();
```

## Creating columns using a version trigger

The database upgrader automatically creates a column that is added to the data model if the column meets one of the following criteria:

- Nullable
- Non-nullable with a default value specified in the metadata
- Non-nullable without a default value if there are no rows in the table
- The column is an editable field

However, you might want to explicitly create the column in your upgrade trigger if you want the trigger to perform an action on the column such as populating it.

In the data model, the column must be defined as a property on an entity. The database upgrade will determine the correct datatype and nullability from the data model.

Creating a new column is moderately expensive in terms of performance of the upgrade.

### Creating a column

To create a column, invoke the `create` method on the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")

 // Create column with given name.
 // Column must be backed by a property on an entity.
 // Upgrader will figure out the correct datatype and nullability.

 table.getColumn("ColumnName").create()
```

### Creating a non-nullable column with an initial value

The upgrader throws an exception if you try to add a new non-nullable column without a default value and there are rows in the table. For non-nullable columns, either specify a default value, or create a version trigger that will populate the column.

To create a new column as non-nullable with an initial value, use the `createNonNullableWithInitialValue()` method. In the data model, the column must be defined as non-nullable.

For example:

```
IBeforeUpgradeTable table = getTable("TableName")
table.getColumn("ColumnName").createNonNullableWithInitialValue(Initial value)
```

The initial value must be of the appropriate type for the column's datatype. You can alter this value in later steps as needed.

### Creating a temporary column

Use the `createTempColumn` method of `IBeforeUpgradeTable` to add a temporary column to the table. The `createTempColumn` method takes two parameters, a `String` for the column name and an `IDataType` for the column data type. `createTempColumn` creates a new nullable column with the given name and datatype to hold temporary data. You must explicitly drop the temporary column during the upgrade. The schema verifier will report an error during server startup if the column has not been dropped. You can create the temporary column in a `BeforeUpgradeVersionTrigger` and drop it in an `AfterUpgradeVersionTrigger`. This approach is useful when you want to move data from a column that will be removed during the upgrade to a column that will be created during the upgrade.

In the following example, a `BeforeUpgradeVersionTrigger` adds a temporary `shorttext` column to an existing entity and populates it with data from another column on a different entity. An `AfterUpgradeVersionTrigger` moves the data to a new entity.

**BeforeUpgradeVersionTrigger Execute Method**

```
// Add a temporary column to TableA.
var tableA = getTable("TableA")
var tempColumn = tableA.createTempColumn("tmp_column", DataTypes.shorttext())

 // Get an IBeforeUpgradeUpdateBuilder for TableA.
var ub = tableA.update()

 // Set the value of the temporary column to the value of ColumnA.
ub.set(tempColumn, ub.getColumnRef("ColumnA"))

 ub.execute()
```

**AfterUpgradeVersionTrigger Execute Method**

```
// Get an IUpdateBuilder for TableA.
var ub = getTable("TableA").update().withLogSQL(true)

 var q = new Query(Account).withLogSQL(true)
q.compare("ID", Equals, ub.getQuery().getColumnRef("Account"))
var piDesc = PaymentInstrument.Type.TypeInfo.getProperty("Description") as IEntityPropertyInfo

 ub.set(piDesc, q, q.getColumnRef(DBFunction.Expr({"tmp_xyz"}))) // tmp_xyz is the DB table column name
ub.execute()

 var tempColumn = getTable("someTable").getColumn("tmp_xyz").drop()
```

## Dropping columns using a version trigger

The upgrader does not drop existing columns in order to prevent data loss. You can implement a version trigger to move the data (not shown in example) and then drop the column by using the `drop()` method of the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")
table.getColumn("ColumnName").drop()
```

There is a `dropColumns` method on `IBeforeUpgradeTable` to drop multiple columns in one statement. The `dropColumns` method takes an array of `IBeforeUpgradeColumn` objects.

For example:

```
var table = getTable("TableName")
table.dropColumns(table.getColumn("ColumnName2"), table.getColumn("ColumnName3"));
```

In Oracle, dropping a column usually has little effect on upgrade performance. Dropping a column actually marks the column as unused in the metadata. At a later point, the DBA is responsible for performing the necessary cleanup. You can override this functionality and force columns to be dropped right away.

In SQL Server, dropping a column is performance-intensive because the RDBMS has to do some clean up work.

## Renaming columns using a version trigger

To rename a column use the rename function on the column object.

```
override function execute() {
  getTable("TableName").getColumn("ColumnName").rename("NewColumnName")
}
```

## Setting a column value for a specific subtype using a version trigger

To set a column to a specific value for specific subtypes, use the `set` and `compare` methods of an `IBeforeUpgradeTable`. Get the typekey ID for comparison using the `BeforeUpgradeVersionTrigger` method `getTypekeyID`.

```
    final var myTable = getTable("tableName")
    final var myTypecode = getTypeKeyID("typelist name", "typelist code")

     final var updateBuilder = myTable.update()

     updateBuilder
      .set("myColumn", "some value")
      .compare("subtype", Equals, myTypecode)

     updateBuilder.execute()
```

## Creating tables using a version trigger

To add a new table to the database, define a new entity in the data model. The upgrade creates the table automatically. However, you might want to explicitly create the table in your upgrade trigger if you want the trigger to perform an action on the table such as populating it.

Creating a new table has negligible impact on upgrade performance.

You can create a regular table using the `create` method of `IBeforeUpgradeTable`. The table must first be defined in the data model.

For example:

```
var table = getTable("TableName").create()
```

### Creating temporary tables

You can add a temporary table to the database based on either the current database schema for a table or the data model definition of a table. You can also create a temporary table with a custom definition.

To create a temporary table based on the current table schema in the database, use the `createNewTempTableBasedOnCurrentSchema` method of `IBeforeUpgradeTable`. The table must be associated with an entity and exist in the database. The returned temporary table will contain the columns that this table has in the database currently. The columns may not match those specified in the entity metadata. For example, the metadata might contain a new column that has not yet been created. The `createNewTempTableBasedOnCurrentSchema` method is usually more appropriate than `createNewTempTableBasedOnThis` if you want to copy data from this table into the new temporary table as the columns will match exactly.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnCurrentSchema()
```

To create a temporary table based on the entity definition of a table in the data model, use the `createNewTempTableBasedOnThis` method of `IBeforeUpgradeTable`. Columns that do not exist in the table are not created on the temporary table, even if the metadata defines such a column. This table may not contain columns that are going to be renamed. The metadata reflects the new name for the column but does not have an entry for the old name, so it would not be added to the temporary table.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnThis()
```

To create a temporary table with a custom definition, use the `createAsNewTempTable` method of `IBeforeUpgradeTable`. This method takes a Pair array in which the first object is a `String` defining the column name and the second object is an `IDataType` defining the column data type.

### Renaming tables using a version trigger

To rename a table use the rename function on the table object.

```
override function execute() {
  getTable("extTableName").rename("TableName_EXT")
}
```

### Deleting rows using a version trigger

To delete rows from a table, use the `delete` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `delete` method returns a delete builder (`IBeforeUpgradeDeleteBuilder` that provides methods for comparing column data to restrict the rows that are deleted.

In the following example, all rows that have a `columnA` value of `0` are deleted.

```
var table =  getTable("SomeTable")

 var deleteBuilder = table.delete()

 deleteBuilder.Query.compare("columnA", Equals, 0)

 deleteBuilder.execute()
```

### Inserting rows using a version trigger

To insert rows of data use the `insert` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insert` method returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable)` for SQL to perform an insert operation.

In the following example, an `IBeforeUpgradeInsertBuilder` is used to add two rows with three columns to table `myTable`. The `IBeforeUpgradeInsertBuilder` includes a description.

```
    var myTable = getTable("SomeTable")

     var insertBuilder = myTable.insert().withDescription("A custom insert
      trigger to add two rows.")
     insertBuilder
      .mapColumn("columnA", "value of column A for first row")
      .mapColumn("columnB", "value of column B for first row")
      .mapColumn("columnC", "value of column C for first row")

     insertBuilder.execute()

 // add a second row
    insertBuilder
      .mapColumn("columnA", "value of column A for second row")
      .mapColumn("columnB", "value of column B for second row")
      .mapColumn("columnC", "value of column C for second row")

     insertBuilder.execute()
```

### Inserting data selected from another table using a version trigger

To insert data selected from another table use the `insertSelect` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insertSelect` method returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`). The builder includes a `mapColumn` method that can be passed explicit values, columns, or a query.

In the following example, the trigger sets `targetTable.column1` to an explicit value. The trigger sets `targetTable.column2` to the value of `sourceTable.sourceColumn`. Because there is no comparison being performed, the trigger will insert a row in the target table for each row in the source table:

```
var sourceTable = getTable("sourceTable")
var targetTable =  getTable("targetTable")

 var insertSelectBuilder = targetTable.insertSelect(sourceTable)

 insertSelectBuilder.mapColumn("column1", "value")    // sets a hard-coded value
  .mapColumn("column2", sourceTable.getColumn("sourceColumn"))    // sets column2 on target table to
                                                                  // source table sourceColumn

 insertSelectBuilder.execute()
```

In the next example, an existing table, `sourceTable`, is split into two tables, `targetTable1` and `targetTable2`.

```
var sourceTable = getTable("sourceTable")
var targetTable1 =  getTable("targetTable1")
var targetTable2 =  getTable("targetTable2")

 var insertSelectBuilder1 = targetTable1.insertSelect(sourceTable)
var insertSelectBuilder2 = targetTable2.insertSelect(sourceTable)


 insertSelectBuilder1.mapColumn("column1", sourceTable.getColumn("sourceColumn1"))
  .mapColumn("column2", sourceTable.getColumn("sourceColumn2"))

 insertSelectBuilder1.execute()

 insertSelectBuilder2.mapColumn("column1", sourceTable.getColumn("sourceColumn3"))
  .mapColumn("column2", sourceTable.getColumn("sourceColumn4"))

 insertSelectBuilder2.execute()
```

### Updating rows using a version trigger

To update rows in a table, use the `update` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. This method returns a builder (`IBeforeUpgradeUpdateBuilder` or `IUpdateBuilder`). The builder includes methods to compare data to restrict which rows are updated.

In the following example, table `SomeTable` is updated to set `column1` to `SomeValue` for each row where the subtype matches a certain entity type:

```
var table =  getTable("SomeTable")

 // get IBeforeUpgradeUpdateBuilder
```

```
var ub = table.update()

 // set column 1 to SomeValue
ub.set("column1", "SomeValue")
  // where
  .compare("subType", Equals, getTypeKeyID(EntityType))

 ub.execute()
```

## Upgrading archived or quote store entities using a version trigger

If you implement archiving, and you make custom data model changes to entities stored in the archive, then you can upgrade the retrieved XML of your archived entity using the `IDatamodelUpgrade` plugin.

You can also use this plugin to upgrade entities in the quote store created by a quote-only instance of PolicyCenter that handles high volume quote requests.

Simple data model changes do not require a custom trigger. These include:

- Adding a new entity
- Updating denormalization columns
- Adding editable columns such as `updatetime`
- Adding new columns
- Changing the nullability of a column

More complex transformations or those that could result in loss of data require a version trigger. These include:

- changing a datatype (other than just length)
- migrating data from one table or column to another
- dropping a column
- dropping a table
- renaming a column
- renaming a table

PolicyCenter upgrades an archived entity as the entity is restored.

The `IDatamodelChange` interface includes a `getArchivedDocumentUpgradeVersionTrigger` method that returns an `ArchivedDocumentUpgradeVersionTrigger`.

You can define custom `ArchivedDocumentUpgradeVersionTrigger` entities to modify archived XML. The `ArchivedDocumentUpgradeVersionTrigger` is an abstract class that you can extend to create your custom triggers.

Define the constructor of your custom `ArchivedDocumentUpgradeVersionTrigger` to call the constructor of the superclass and pass it a numeric value. For example:

```
construct() {
  super(171)
}
```

This numeric value is the extension version to which your trigger applies. If you run the upgrade against a database with a lower extension version, then your custom trigger is called. The current extension version is defined in `modules/configuration/config/extensions/extensions.properties.`

Provide an override definition of the `get Description` property to return a `String` that describes the actions of your trigger.

Provide an override definition for the `execute` function to define the actions that you want your custom trigger to make on archived XML.

When the upgrade executes your custom trigger, it wraps each XML entity in an `IArchivedEntity` object. Each typekey is wrapped in an `IArchivedTypekey` object. The upgrade operates on a single XML document at a time.

`ArchivedDocumentUpgradeVersionTrigger` provides the following key operations:

- `getArchivedEntitySet(`*`entityName`*` : String)` – returns an `IArchivedEntitySet` object that contains all `IArchivedEntity` objects of the given type in the XML document.

`IArchivedEntitySet` provides the following key methods:

- `rename(`*`newEntityName`*` : String)` – renames all rows in the set to the new name.
- `delete()` – deletes all rows in the set.
- `search(`*`predicate`*` : Predicate<IArchivedEntity>)` – returns a `List` of `IArchivedEntity` objects that match the given predicate.
- `create(`*`referenceInfo`*` : String, `*`properties`*` : List<Pair<String, Object>>)` – returns a new `IArchivedEntity` with the given properties.

`IArchivedEntity` provides the following key methods:

- `delete()` – deletes just this row.
- `getPropertyValue(`*`propertyName`* : String`)` – returns the value of the property of the given name. If the property value is a reference to another entity, this method returns an `IArchivedEntity`.
- `move(`*`newEntityName`*` : String)` – moves this to a new entity type of the given name. The type is created if it did not exist. You must add any required properties to the type.
- `updatePropertyValue(`*`propertyName`*` : String, `*`newValue`*` : String)` – updates the property of the given name to the given value.
- `getArchivedTypekeySet(`*`typekeyName`*` : String)` – returns an `IArchivedTypekeySet` object that contains all `IArchivedTypekey` objects in the given typelist.
- `getArchivedTypekey(typelistName : String, code : String)` – Returns an `IArchivedTypekey` representing the typekey.

  `IArchivedTypekey` provides the following key method:

  - `delete()` – deletes the typekey from the XML.

More methods are available for `IArchivedEntitySet`, `IArchivedEntity` and `IArchivedTypekey`. See the Gosu documentation for a full listing. Generate the Gosu documentation by navigating to the PolicyCenter directory and entering the following command:

```
gwb genGosuApi
```

## Incremental upgrade

When PolicyCenter archives an entity, it records the current data model version on the entity. PolicyCenter upgrades an archived entity as the entity is restored. The upgrader executes the necessary archive upgrade triggers incrementally on each archived XML entity, according to the data model version of the archived entity.

An entity archived at one version might not be restored and upgraded until several intermediate data model upgrades have been performed. Therefore, do not delete your custom upgrade triggers.

Consider the following situation. Note that this example is for demonstration purposes only. The version numbers included do not represent actual PolicyCenter versions but are included to explain the incremental upgrade process. Each '+' after a version number indicates a custom data model change.

**Guidewire data model changes**

v6.0.0 – Entity does not have column X.

v6.0.1 – Guidewire adds column X to the entity with a default value of 100.

v6.0.2 – Guidewire updates the default value of column X to 200.

**Implementation #1 upgrade path**

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.0+++ – More custom data model changes.

v6.0.2+ – column X introduced with a default value of 200.

Result of upgrade of an entity archived at v6.0.0+: column X has value 200.

In this situation, version 6.0.1 was skipped in the upgrade path. Therefore, column X is added with the default value of 200 that it has in version 6.0.2.

**Implementation #2 upgrade path**

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.1+ – column X introduced with a default value of 100.

v6.0.2+ – column X already exists.

Result of upgrade of an entity archived at v6.0.0+: column X has value 100.

In this situation, column X is added in version 6.0.1 with the default value of 100. During the upgrade from version 6.0.1 to version 6.0.2, column X already exists. Therefore, the upgrader does not add the column. A custom trigger would be required to update the value of X from 100 to 200 during the upgrade from version 6.0.1 to 6.0.2.

## Configuring database upgrade

You can set parameters for the database upgrade in the PolicyCenter 10.0.0 `database-config.xml` file. The `<database>` block in `database-config.xml` contains parameters for database configuration, such as connection information. The `<database>` block contains an `<upgrade>` block that contains configuration information for the overall database upgrade. The `<upgrade>` block also contains a `<versiontriggers>` element for configuring general version trigger behavior and can contain `<versiontrigger>` elements to configure each version trigger.

#### See also

The `<versiontriggers>` database configuration element in the *Installation Guide*.

*Installation Guide*.

### Adjusting commit size for encryption

You can adjust the commit size for rows requiring encryption by setting the `encryptioncommitsize` attribute to an integer in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade encryptioncommitsize="10000">
    ...
  </upgrade>
</database>
```

If PolicyCenter encryption is applied on one or more attributes, the PolicyCenter database upgrade commits batches of encrypted values. The upgrade commits `encryptioncommitsize` rows at a time in each batch. The default value of `encryptioncommitsize` varies based on the database type. For Oracle, the default is `10000`. For SQL Server, the default is `100`.

Test the upgrade on a copy of your production database before attempting to upgrade the actual production database. If the encryption process is slow, and you cannot attribute the slowness to SQL statements in the database, try adjusting the `encryptioncommitsize` attribute. After you have optimized performance of the encryption process, use that `encryptioncommitsize` when you upgrade your production database.

### Configuring version trigger elements

The database upgrade executes a series of version triggers that make changes to the database to upgrade between versions. You can set some configuration options for version triggers in `database-config.xml`. Normally, the default settings are sufficient. Change these settings only while investigating a slow database upgrade.

The `<database>` element in `database-config.xml` contains an `<upgrade>` element to organize parameters related to database upgrades. Included in the `<upgrade>` element is a `<versiontriggers>` element, as shown below:

```
<database ...>
  <param ... />
  <upgrade>
    <versiontriggers dbmsperfinfothreshold="600" />
  </upgrade>
</database>
```

The `<versiontriggers>` element configures the instrumentation of version triggers. This element has one attribute: `dbmsperfinfothreshold`. The `dbmsperfinfothreshold` attribute specifies for all version triggers the threshold after which the database upgrader gathers performance information from the database. You specify `dbmsperfinfothreshold` in seconds, with a default of `600`. If a version trigger takes longer than `dbmsperfinfothreshold` to execute, PolicyCenter:

- Queries the underlying database management system (DBMS).
- Builds a set of html pages with performance information for the interval in which the version trigger was executing.
- Includes those html pages in the upgrade instrumentation for the version trigger.

You can completely turn off the collection of database snapshot instrumentation for version triggers by setting the `dbmsperfinfothreshold` to 0 in `config.xml`. If you do not have the license for the Oracle Diagnostics Pack, you must set `dbmsperfinfothreshold` to 0 before running the upgrade.

The `<versiontriggers>` element can contain optional `<versiontrigger>` elements for each version trigger. Each `<versiontrigger>` element can contain the following attributes.

| Attribute | Type | Description |
|---|---|---|
| `name` | String | The case-insensitive name of a version trigger. |
| `extendedquerytracingenabled` | Boolean | Oracle only. Controls whether or not to enable extended sql tracing (Oracle event 10046) for the SQL statements that are executed by the version trigger. Default is `false`. The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation. |
| `parallel-dml` | Boolean | Oracle only. See “Configuring parallel dml and DDL statement execution for Oracle database upgrade” on page 52. |
| `parallel-query` | Boolean | Oracle only. Hints to the optimizer to use parallel queries when executing queries run by a version check associated with the upgrade trigger. See note below. |
| `queryoptimizertracingenabled` | Boolean | Oracle only. Controls whether or not to enable query optimizer tracing (Oracle event 10053) for the SQL statements that are executed by the version trigger. Default is `false`. The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation. |
| `recordcounters` | Boolean | Controls whether the DBMS-specific counters are retrieved at the beginning and end of the use of the version trigger. Default is `false`. If `true`, then PolicyCenter retrieves the current state of the counters from the underlying DBMS at the beginning of execution of the version trigger. If the execution of the version trigger exceeds the `dbmsperfinfothreshold`, then PolicyCenter retrieves the current state of the counters at the end of the execution of the version trigger. PolicyCenter writes differences to the DBMS-specific instrumentation pages of the upgrade instrumentation. |
| `updatejoinorderedhint` | Boolean | Oracle only. Whether to use the ORDERED hint for the UPDATE of a join. Default is false. |
| `updatejoinusemergehint` | Boolean | Oracle only. Whether to use the USE_MERGE hint for the UPDATE of a join. Default is false. |
| `updatejoinusenlhint` | Boolean | Oracle only. Whether to use the USE_NL hint for the UPDATE of a join. Default is false. |

**Note:** For Oracle, there is an important exception to the ability of the `<versiontriggers>` element to provide overrides for its `<versiontrigger>` subelements. Although the parent element, `<upgrade>`, might have defined the `ora-parallel-dml` and `ora-parallel-query` attributes, these values are not applied to version checks which are included as `<versiontrigger>` subelements. To get a version check to run with parallel query in Oracle, you must list it explicitly, add it as a subelement of the `<versiontriggers>` element, and indicate `parallel-query="true"` on that `<versiontrigger>` subelement.

For example: `<versiontrigger name="com.guidewire.cc.system.database.upgrade.check.RecoveryCategoryNullForPaymentsAndReservesVersionCheck" parallel-query="true"/>`. Other version triggers (but not version checks) that are not listed will inherit the parallel setting defined on the `<upgrade>` element. Note that even though `RecoveryCategoryNullForPaymentsAndReservesVersionCheck` is a version check, it must be included as `<versiontrigger>` element.)

In summary, version checks are exempted from Oracle parallelism unless you explicitly list them. Version triggers are not.

## Deferring creation of nonessential indexes

You can configure the upgrade to defer creation of nonessential indexes during the upgrade process until the upgrade completes and the application server is online. Nonessential indexes are performance-related indexes that do not enforce constraints and indexes on the `ArchivePartition` column on all entities that PolicyCenter can archive. Creation of nonessential indexes can add significant time to the upgrade duration, so it is possible to defer this process. By default, the upgrade does not defer creation of these indexes.

To configure the upgrade to defer creation of nonessential indexes set the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` to `true`.

```
<database ...>
  <upgrade defer-create-nonessential-indexes="true">
  ...
  </upgrade>
</database>
```

If you opt to defer creation of nonessential indexes, PolicyCenter runs the `DeferredUpgradeTasks` batch process as soon as the upgrade completes and the server is completely started. The `DeferredUpgradeTasks` batch process creates the nonessential performance indexes and indexes on archived entities. The database user must have permission to create indexes until after the `DeferredUpgradeTasks` batch process is complete.

Deferring nonessential index creation can shorten the duration of the upgrade process. The PolicyCenter database is then available sooner for tasks including upgrade verification and backing up the upgraded database before the database is opened up for production use. To take advantage of this earlier availability, perform upgrade testing and validation tasks while the `DeferredUpgradeTasks` batch process is running. Do not go into full production while the process is still running. The lack of so many performance-related indexes could likely make the system unusable.

Until the `DeferredUpgradeTasks` batch process has run to completion, PolicyCenter reports errors during schema validation when starting. These include errors for column-based indexes existing in the data model but not in the physical database and mismatches between the data model and system tables.

Do not use the archiving feature until the `DeferredUpgradeTasks` batch process has completed successfully.

Check the status of the `DeferredUpgradeTasks` batch process to determine when it has completed successfully. You can find the status of the deferred upgrade in the upgrade logs and on the PolicyCenter **Upgrade Info** page. If the `DeferredUpgradeTasks` batch process fails, manually run the batch process again during non-peak hours.

If you do not opt to defer creation of nonessential indexes, PolicyCenter creates these indexes as part of the upgrade process that must complete before the application server is online. If you do not want to defer creating nonessential indexes, the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` must be set to `false`. This is the default setting.

## Configuring the upgrade on Oracle

### Configuring column removal for Oracle database upgrade

The database upgrade removes some columns. For Oracle, you can configure whether the removed columns are dropped immediately or are marked as unused. Marking a column as unused is a faster operation than dropping the column immediately. However, because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments. You can drop the unused columns after the upgrade during off-peak hours to free the space. Or, you can configure the database upgrade to drop the columns immediately during the upgrade. By default, the PolicyCenter database upgrade marks columns as unused.

To configure the PolicyCenter upgrade to drop columns immediately during the upgrade, set the `deferDropColumns` attribute of the `<upgrade>` block in `database-config.xml` to `false`. For example:

```
<database ...>
  ...
  <upgrade deferDropColumns="false">
    ...
  </upgrade>
</database>
```

By default, `deferDropColumns` is `true`.

### Configuring parallel dml and DDL statement execution for Oracle database upgrade

You can configure whether the upgrade executes DML (Data Manipulation Language) and DDL (Data Definition Language) statements in parallel or not and the degree of parallelism to use.

The `<upgrade>` element includes an `ora-parallel-dml` attribute. This attribute can be set to `disable`, `enable`, or `enable_all`. The default value is `enable`. If `ora-parallel-dml` is set to `disable`, the upgrade does not conduct parallel execution of DML statements. If `ora-parallel-dml` is set to `enable`, the upgrade executes DML statements in parallel if configured or coded for a version trigger. If `ora-parallel-dml` is set to `enable-all`, the upgrade executes DML statements in parallel in all cases unless turned off in the code or configuration for a version trigger.

The Boolean attribute `parallel-dml` of a `<versiontrigger>` element controls parallel execution for that version trigger. If `parallel-dml` is not set, the upgrade executes parallel DML statements if coded or if `ora-paralled-dml` is set to `enable_all` on the `<upgrade>` element. If `parallel-dml` is set to `false`, the upgrade does not execute DML statements in parallel. If `parallel-dml` is set to `true`, the upgrade executes DML statements in parallel if `ora-parallel` is set to `enable` or `enable_all`.

To configure the degree of parallelism for insert, update and delete operations, set the `degree-of-parallelism` attribute on the `<upgrade>` element. To configure the degree of parallelism for commands such as creating an index and enabling constraints using the alter table command, set the `degree-parallel-ddl` attribute on the `<upgrade>` element.

You can specify a value from `2` to `1000` to force that degree of parallelism. Specify a value of `1` to disable the use of parallel execution.

Setting either parameter to `0` configures PolicyCenter to defer to Oracle to determine the degree of parallelism for the operations that attribute configures. The Oracle automatic parallel tuning feature determines the degree based on the number of CPUs and the value set for the Oracle parameter `PARALLEL_THREADS_PER_CPU`.

The default for both attributes is `4`.

You can configure parallel DML execution on the `InsertSelectBuilder`, `BeforeUpgradeUpdateBuilder` and `BeforeUpgradeInsertSelectBuilder` of a custom version trigger using the `withParallelDml(`*`boolean`*`)` method. If not explicitly set to `true` or `false`, the upgrade uses parallel execution if configured. If set to `false`, the upgrade does not use parallel execution unless set to `true` for that version trigger. If set to `true`, it will be done unless set to false for that version trigger or `ora-parallel-dml` is set to `disable`.

Some version triggers read large amounts of data when running their version check. To improve performance, you can configure a version trigger to hint to the optimizer to use parallel queries when executing SQL queries. To do so, set the `parallel-query` attribute on the `<versiontrigger>` element to `true`.

### Collecting tablespace usage and object size for Oracle database upgrade

To enable collection of tablespace usage and object size data on Oracle, set the `collectstorageinstrumentation` attribute of the `<upgrade>` block to `true`. For example:

```
<database ...>
  ...
  <upgrade collectstorageinstrumentation="true">
    ...
  </upgrade>
</database>
```

A value of `true` enables PolicyCenter to collect tablespace usage and size of segments such as tables, indexes and LOBs (large object binaries) before and after the upgrade. The values can then be compared to find the utilization change caused by the upgrade.

### Disabling Oracle logging for Oracle database upgrade

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="true">
    ...
  </upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run statements with the `NOLOGGING` option.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. Some examples include Reporting, Disaster Recovery through Standby databases and Oracle Dataguard. To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

### Disabling statistics update for the database for Oracle database upgrade

Generating table statistics during upgrade is optional for Oracle databases. The overall time required to upgrade the database is shorter if the database upgrade does not update statistics. To disable statistics generation during the upgrade, set the `updatestatistics` attribute of the `<upgrade>` element to `false`:

```
<upgrade updatestatistics="false">
```

Setting `updatestatistics` to `true` enables the upgrader to update statistics on changed objects. It also allows the upgrade to maintain column level statistics consistent with what is allowed in the code, data model and configuration.

If statistics are not updated during the upgrade, PolicyCenter reports a warning that recommends that you run the database statistics batch process in incremental mode. Additionally, the **Upgrade Info** page shows that statistics were not updated as part of the upgrade. If statistics generation was not disabled, the **Upgrade Info** page reports the runs of the statistics batch process, including incremental runs.

You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded PolicyCenter in a production environment. If you defer generating statistics during the upgrade, Guidewire recommends that you generate full statistics as soon as possible after the upgrade. For instructions, see the *System Administration Guide*.

The **Upgrade Info** page does not identify the following case: You ran an upgrade with `updatestatistics=true` after running a previous upgrade with `updatestatistics=false`, but you did not update statistics first.

When you click the **Download** button on the **Upgrade Info** page, you get a more detailed report. This report shows the value of the `updatestatistics` attribute at the time of upgrade. Additionally, the report shows the update statistics SQL statements that were skipped as part of the upgrade. These statements are provided for reference. You typically do not need to review these statements if you run the incremental database statistics process following the upgrade.

### Disabling statistics update for tables with locked statistics for Oracle database upgrade

If you have tables that have locked statistics, specify to keep statistics on these tables before starting the database upgrade. To specify to keep statistics on a table, set the `action` attribute of the `<tablestatistics>` element for that table to `keep`. The `<tablestatistics>` element is nested within the `<databasestatistics>` element, which is within the `<database>` element in `database-config.xml`.

For example, if statistics are locked on `pc_someTable_EXT`, specify a `<tablestatistics>` element for that table with the `action` attribute set to `keep`:

```
<database>
  ...
  <databasestatistics>
    <tablestatistics name="pc_someTable_EXT" action="keep" />
  </databasestatistics>
</database>
```

## Configuring the upgrade on SQL Server

### Disabling SQL Server logging for SQL Server database upgrade

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="true">
    ...
  </upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run with minimal logging. This can improve the performance of the upgrade. During the upgrade, set the SQL Server recovery model to Simple or Bulk logged. Once the upgrade and deferred upgrade tasks are complete, you can revert the recovery model setting and back up the full database.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. If you require full logging due to the presence of solutions such as Database Mirroring, continue to use the Full recovery model and set `allowUnloggedOperations` to `false`.

To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

### Storing temporary sort results in tempdb for SQL Server database upgrade

For SQL Server databases, you can specify to store temporary sort results in tempdb by setting the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` block to `true`. By using tempdb for sort runs, disk input and output is typically faster, and the created indexes tend to be more contiguous. By default, `sqlserverCreateIndexSortInTempDB` is `false` and sort runs are stored in the destination filegroup.

If you set `sqlserverCreateIndexSortInTempDB` to `true`, you must have enough disk space available to tempdb for the sort runs, which for the clustered index include the data pages. You must also have sufficient free space in the destination filegroup to store the final index structure, because the new index is created before the old index is deleted. Refer to `http://msdn.microsoft.com/en-us/library/ms188281.aspx` for details on the requirements to use tempdb for sort results.

### Specifying filegroup to store sort results for clustered indexes for SQL Server database upgrade

For SQL Server databases, a version trigger recreates non-clustered backing indexes for primary keys as clustered indexes.

Before recreating the indexes, the version trigger automatically drops (and later rebuilds) any referencing foreign keys and drops any clustered indexes on tables with a primary key.

If you are using filegroups, the upgrade recreates the clustered index in the OP filegroup. By default, the upgrade also stores the intermediate sort results that are used to build the index in the OP filegroup. You can configure the upgrade to instead use the tempdb filegroup for the intermediate sort results.

If you want the upgrade to stores the intermediate sort results in the tempbdb filegroup, set the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` element to `true`.

```
<database ...>
  ...
  <upgrade sqlserverCreateIndexSortInTempDB="true" />
    ...
  </upgrade>
</database>
```

This option increases the amount of temporary disk space that is used to create an index. However, it might reduce the time that is required to create or rebuild an index when tempdb is on a different set of disks from that of the user database.

By default, `sqlserverCreateIndexSortInTempDB` is `false`.

### Downloading database upgrade instrumentation details

The database upgrade deletes upgrade instrumentation information for prior database upgrades. If the database upgrade detects any prior upgrade instrumentation data, it reports a warning and deletes the data. If you have run previous database upgrades, and you want to preserve upgrade instrumentation details, follow the procedure in "Viewing detailed database upgrade information" on page 60.

## Checking the database before upgrade

The upgrade runs a series of version checks prior to making any changes to the database. These version checks ensure that the database is in a state that can be upgraded. Guidewire includes a number of version checks with PolicyCenter and you can also add custom version checks.

You can configure PolicyCenter to run the version checks only, including custom version checks. Before upgrading the production database, run version checks on a clone of your production database to identify any issues with your data.

### Run version checks without database upgrade

#### Procedure

1. Start Studio for PolicyCenter 10.0.0 by running the following command from the PolicyCenter directory:

   ```
   gwb studio
   ```

2. Expand **configuration→config** and open `database-config.xml`.
3. Add the attribute `versioncheckonly=true` to the `database` element.
4. Verify that the database connection is pointing to a clone of your production database.
5. Save your changes.
6. Start the server.

   PolicyCenter reports the number of version check errors. For any errors reported PolicyCenter reports which version check resulted in the error along with the error message.
7. If PolicyCenter reports version check errors, fix the data and rerun the version checks. Repeat this process until no errors are reported on the production clone. Apply the fixes to your production database prior to upgrade.

   With `versioncheckonly=true` set, PolicyCenter runs all version checks regardless of a failure in one of the checks. During a regular upgrade, PolicyCenter stops the upgrade if an error is detected.
8. After you have fixed all version check errors, set `versioncheckonly` to `false` to run the actual upgrade.

## Disable the scheduler before upgrade

### About this task

Before you start the server to upgrade the database, disable the scheduler for batch processes and work queues. Disabling the scheduler prevents batch processes and work queues from launching immediately after the database upgrade.

### Procedure

1. Open the PolicyCenter 10.0.0 `config.xml` file in a text editor.
2. Set the `SchedulerEnabled` parameter to `false`.

   ```
   <param name="SchedulerEnabled" value="false"/>
   ```
3. Save `config.xml`.

### Next steps

After you have verified that your database upgrade completed successfully, enable the scheduler again.

### See also

"Enable the scheduler after upgrade" on page 64

## Suspend message destinations

### About this task

Suspend all event message destinations before you upgrade the database to prevent PolicyCenter from sending messages until you have verified a successful database upgrade.

### Procedure

1. Start the PolicyCenter server for the pre-upgrade version.
2. Log in to PolicyCenter with an account that has administrative privileges, such as the superuser account.
3. Click the **Administration** tab.
4. Click **Event Messages**.
5. Select the check box to the left of the **Destination** column to select all message destinations.
6. Click **Suspend**.
7. Resume messaging after you have verified a successful database upgrade.

## Starting the server to begin automatic database upgrade

The database upgrade is an automatic process that occurs as you start the server with the upgraded configuration of a new PolicyCenter version. The database upgrade normally completes in a few hours or less.

If the database upgrade stops before completing, then restore your database from the backup, correct any issues reported, and repeat the database upgrade.

**IMPORTANT** Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of PolicyCenter 10.0.0. See the *Supported Software Components* knowledge article for current system and patch level requirements. Visit the Guidewire Community and search for the *Supported Software Components* knowledge article.

**WARNING** Except for your first database upgrade trials, do not start the server until you have upgraded all rules. Otherwise, default validation rules execute. This could strand objects at a high validation level and make it impossible to edit parts of the object.

**WARNING** The database upgrade runs a series of version checks prior to making any changes. If any of these checks fail, the upgrade aborts and reports an error message. You can fix the issue, create an updated backup of the database and attempt the upgrade again without restoring from a backup. However, if you experience a failure during the version triggers or upgrade steps portion of the upgrade, refresh the database from a backup before attempting the upgrade again.

Guidewire requires that you use a separate mirror database for reporting. If you did not do so, then you can experience problems during a database upgrade that are severe enough to prevent the upgrade.

In particular, the Premium Accounting extension package SQL scripts create special tables in the reporting database that reporting uses for storing premium accounting accrual data. The reporting scripts use these tables, but the PolicyCenter application does not. If you created these tables in a production database, then any attempt to upgrade that production database will fail.

If you encounter this situation, move data from all `pcrt_` prefixed tables to another schema. Then, drop all `pcrt_` prefixed tables from the database that you want to upgrade before starting the server to launch the upgrade.

## Testing the database upgrade

In a development environment the database upgrade process records checkpoints of upgrade triggers that complete successfully. You can restart a failed database upgrade, and it resumes with the upgrade trigger that failed. This restart feature helps you test the upgrade with realistically large data sets. You avoid time spent to restore the database and rerun upgrade triggers that worked successfully.

Guidewire provides this feature for convenience while testing. However, it does not work for all failure scenarios. Even in development mode, under certain scenarios, you will have to restore a backup of the database taken prior to the upgrade attempts and then run the upgrade.

The database upgrade writes SQL executed by the failed trigger to the console. To restart a test database upgrade from a checkpoint reached in an earlier upgrade, manually roll back any database changes that occurred during the upgrade trigger that failed. Resolve the problem that caused the trigger to fail. Then start the server again to restart the upgrade. The upgrade skips successful upgrade triggers and continues by rerunning the trigger that failed.

### Test the database upgrade

#### About this task

Prior to attempting the database upgrade on a full-production database clone, test the database upgrade.

In a development environment the database upgrade process records checkpoints of upgrade triggers that complete successfully. You can restart a failed database upgrade, and it resumes with the upgrade trigger that failed. This restart feature helps you test the upgrade with realistically large data sets. You avoid time spent to restore the database and rerun upgrade triggers that worked successfully.

Guidewire provides this feature for convenience while testing. However, it does not work for all failure scenarios. Even in development mode, under certain scenarios, you will have to restore a backup of the database taken prior to the upgrade attempts and then run the upgrade.

The database upgrade writes SQL executed by the failed trigger to the console. To restart a test database upgrade from a checkpoint reached in an earlier upgrade, manually roll back any database changes that occurred during the upgrade trigger that failed. Resolve the problem that caused the trigger to fail. Then start the server again to restart the upgrade. The upgrade skips successful upgrade triggers and continues by rerunning the trigger that failed.

**WARNING** Never use the restart feature of database upgrade in a production environment.

### Procedure

1. Connected to the built-in Quickstart database, successfully start the built-in Quickstart application server with a merged configuration data model, including merged extensions, data types, field validators, and so forth.
2. Connected to an empty database on an Oracle or SQL Server database server, successfully start the Quickstart application server from the preceding step.
3. Connected to a restored backup of a production clone, start either the same Quickstart server from the preceding step or a supported third-party application server with your custom configuration.

### Next steps

A test run of your upgrade is successful only when it runs from start to finish without a restart.

## Integrations and starting the server

Disable all integrations during the automatic database upgrade. Integration points might require updates due to changes in Guidewire APIs. See the *New and Changed Guide* for specifics.

It is not necessary to have completely migrated integrations before attempting to start the server for the first time. If you have integrations that rely on non-Guidewire applications, do not expect these integrations to work the first time you start the server.

## Understanding the automatic database upgrade

As the database upgrade proceeds, it logs messages to the console as well as the log file describing its progress. The database upgrade process requires thousands of steps, divided into three phases. Due to the relational nature of a database, these phases must execute in a specific order for the upgrade to succeed.

During the first phase, the upgrader first executes custom `BeforeUpgradeVersionTrigger` version checks and triggers defined in the `IDatamodelUpgrade` plugin. The upgrader next runs version checks defined by Guidewire. Then, the upgrader uses a set of version triggers defined by Guidewire to determine the actions that are required. The database upgrader requires version triggers in order to perform the following types of tasks:

- Changing a datatype (other than just length)
- Migrating data
- Dropping a column
- Dropping a table
- Renaming a column
- Renaming a table

### See also

"Version trigger descriptions" on page 58

Many version triggers have version checks associated with them. These checks ensure that the database is ready for the associated version trigger. The database upgrade runs all checks before running any version triggers. If a check detects a problem, it reports the issue, including a sample SQL query to find specific problematic records. If a version check discovers an issue, the database upgrade stops before any version triggers are run. Therefore, it is not necessary to restore the database from a backup if a version check reports an error. Correct the issue and then create a new backup of the database. Then, if you encounter errors after the version check stage, you can restore a version of your database with the issue reported by the version check resolved.

In the second phase, the upgrader compares the target data model and the current database to determine how they differ. The upgrader makes changes to the database that do not require a version trigger during this phase.

Following this process, the third phase runs a subsequent set of version triggers. These triggers create actions that must be run last due to a dependency on an earlier phase.

After the database upgrade concludes, it reports issues that the upgrader encountered and did not complete.

You are responsible for correcting these issues. This might involve modifying the data model or altering the table manually. If you do not correct them, the next time you start the server you do *not* see a message that the database and the data model are out of sync. You must then use the `system_tools` command to verify the database schema.

> **Note:** Given the complexity of database upgrade, Guidewire does not expose specific upgrade actions/steps to clients either in SQL or Java form. Any manual attempts to recreate or control the upgrade process can result in problems in the PolicyCenter database. Recovery from such attempts is not supported.

## Version trigger descriptions

PolicyCenter uses version triggers to update the database during an upgrade. Review the version trigger descriptions to familiarize yourself with the changes that will be applied to your database. If a version trigger has an associated version check, the version check is described with the trigger. If a version check reports an issue, review the error message and consult the description of the relevant version trigger for more information.

### Renaming deferred upgrade batch process

The upgrade renames the `DeferredUpgrade` batch process type to `DeferredUpgradeTasks`.

### Truncating pc_Dynamic_Assign

The upgrade truncates the `pc_Dynamic_Assign` table.

### Dropping pc_tl_Template

The upgrade drops the `pc_tl_Template` table.

### Dropping columns from WorkItem tables

The upgrade drops the `AvailableSince` and `LastUpdateTime` columns from all `pc_WorkItem` tables.

### Upgrading shared typekey data

The upgrade checks for subtypes with typekeys that have the same field name, different column names, and only one column exists in the database. If any such records exist, the upgrade moves the data to the correct column.

### Dropping CityKanjiDenorm from Contact

The upgrade drops the `CityKanjiDenorm` search column from `pc_Contact`. The `CityKanjiDenorm` property has been removed from the `Contact` entity.

If you need to preserve this column, you must do the following in the target system before upgrading:

1. Add the `CityKanjiDenorm` property back to `Contact.etx`. If the the `CityKanjiDenorm` property exists in the target configuration, the upgrade will not remove this property.
2. Uncomment the `CityKanji` criterion (`( <Criterion property="CityKanji" targetProperty="CityKanjiDenorm" matchType="startsWith"/>)` ) in `search-config.xml` to re-enable the related search feature.

### Creating AgencyBillPlan records

For each record in `pc_Organization` with a non-null `AgencyBillPlanID`, the upgrade creates a record in `pc_AgencyBillPlan` with the organization `ID` and `AgencyBillPlanID`.

### Dropping rating worksheet tables

The upgrade drops tables and entities that store `RatingWorksheets` directly. As of PolicyCenter 8.0.1, rating worksheets are stored on a `WorksheetContainer` stored on a policy. The `RatingWorksheet` delegate has been removed.

## Moving CommissionPlanID from ProducerCode to ProducerCodeCurrency

The upgrade copies the `pc_ProducerCode.CommissionPlanID` to `pc_ProducerCodeCurrency.CommissionPlanID` and then deletes `pc_ProducerCode.CommissionPlanID`.

## Deleting checksums for product model lookups

The upgrade deletes checksums for product model lookups and checksums from `pc_parameter` from the database. This forces PolicyCenter to resynchronize the database with product model files.

## Moving PolicyPeriod.NewInvoiceStream.Selected to PolicyPeriod.CustomBilling

The upgrade moves and renames the `Selected` column from `PolicyPeriod.NewInvoiceStream.Selected` to `PolicyPeriod.CustomBilling`. The `Selected` column did not indicate that the user selected to send a new invoice stream to BillingCenter. Instead, the user indicated custom billing and PolicyCenter either sends an invoice stream or modifies an existing one. PolicyCenter did not send any invoice stream information otherwise. Therefore the column has been renamed to `PolicyPeriod.CustomBilling` to better reflect its purpose.

## Creating the SelectedPaymentPlan PaymentPlanSummary

In PolicyCenter 8.0.2, the relationship between `PolicyPeriod` and `PaymentPlanSummary` was changed to a one-to-one relationship. The upgrade first checks that each non-retired `PaymentPlanSummary` record has a unique combination of `PolicyPeriod` and `BillingId`. The upgrade reports an error if it finds `PaymentPlanSummary` records with matching `PolicyPeriod` and `BillingId`. If the upgrade reports this error, either retire or remove the duplicate rows and restart the upgrade. When a record is restored from the archive it is upgraded to the current version. If a duplicate `PaymentPlanSummary` is detected during restoration, PolicyCenter marks the duplicate `PaymentPlanSummary` as retired.

If the upgrade finds no errors, it creates a new `SelectedPaymentPlan PaymentPlanSummary` for all `PolicyPeriods` and removes obsolete `PaymentPlanSummary` types.

## Deleting checksums for product model lookups and lookup rows

PolicyCenter 8.0.3 has a restructured product model. The upgrade deletes checksums for product model lookups and lookup rows from the database to allow resynchronization of product model files when the server starts.

The upgrade deletes all rows from the following tables:

- `pc_CondLookup`
- `pc_CovLookup`
- `pc_CovTermLookup`
- `pc_CovTermOptLookup`
- `pc_CovTermPackLookup`
- `pc_ExclLookup`
- `pc_ModifierLookup`
- `pc_OfferingLookup`
- `pc_ProductLookup`
- `pc_ProductModifierLookup`
- `pc_ProdRateFactorLookup`
- `pc_RatingFactorLookup`
- `pc_QuestionLookup`
- `pc_QuestionSetLookup`

## Resetting BasedOnID

The upgrade resets the `BasedOnID` column value to `null` for every effective-dated table row where the `BasedOn` record has a `BranchID` that is not equal to its branch's `BasedOnID`. This upgrade trigger fixes a data issue introduced in 7.0.5 patch 6.

### Setting CascadedLookup on RateBook

The upgrade adds a new column `CascadedLookup` to the `RateBook` table. For existing `RateBook` records the upgrade sets `CascadedLookup` to `false` to ensure consistent behavior.

### Setting default rate table argument source sets

As of version 8.0.4, all rate table definitions must have an argument source set. The upgrade modifies existing rate table definitions and rate table lookups in rate routines that do not specify an argument source set.

The upgrade creates an `EmptyParameterSet` parameter set that applies to all policy lines. This parameter set contains no parameters.

For all rate table definitions that do not specify an argument source set, the upgrade:

- Sets `EmptyParameterSet` as the source of argument objects. The **Source of Argument Objects** appears on the **Argument Sources** tab. `EmptyParameterSet` is both the name and code of this parameter set.
- Adds a `<system default>` argument source set. The argument source for each parameter is set to null. `<system default>` is both the name and code of this argument source set.

For all rate table lookups in rate routines that do not specify an argument source set, the version trigger sets the argument source set to `<system default>`.

### Dropping ClaimID column from AuthorityLimit

Sets the `LastPolicies` column in the `UserSettings` table to `null`. The `LastPolicies` column contains a list of recent policies. Setting this value to `null` removes recent policies from user preferences.

### Marking underwriting rule as externally managed

This trigger sets `UWRule.ExternallyManaged` to `true` if the `UWRule` has no `RuleCondition` and `AvailableToRun` is `false`. Before upgrade, ensure that Gosu rules are not marked `AvailableToRun`.

The result will be incorrect for rules for which the rule condition is not yet defined, but is not externally managed. For example, a rule under development may not yet have a rule condition. If this happens during your upgrade, delete the rule and redefine it.

### Drop tableregistry table

Drops the `tableregistry` table.

### Drop all staging tables

Drops all staging tables.

### Drop RuleHead and RuleVersion subtypes

Updates the `RuleHead` and `RuleVersion` tables to drop the subtype column.

### Create ConfigFPColumn for RollingUpgrade

Creates the `ConfigFP` configuration fingerprint column in the `RollingUpgrade` table.

### Alter Datetime columns to Datetime2

On SQL Server ONLY, alters all `datetime` columns to `datetime2`.

### Add rule type to RuleExportImportTask

Replaces the obsolete the obsolete `RuleVersionType` attribute with the new `RuleType` attribute for the `RuleExportImportTask` entity. This trigger creates the column and sets its value to the existing single use case rule subtype entity typekey.

#### Add ActivityPattern

Adds the `async_quote_completed` activity pattern if it is absent.

#### Add InvoicingMethod typekey

Adds the `InvoicingMethod` typekey to `PolicyPeriod`, and converts `CustomBilling` to `InvoicingMethod`. Removes `CustomBilling` from `PolicyPeriod`. For upgrade from a release prior to 8.0.2, converts the `BillingInvoiceStream.selected` to `InvoicingMethod`.

#### Remove CreateUWIssuePermission

Removes all role privileges with the permission `createevalissue`.

#### Rename AccountFrozen to LockedFromMerge

Renames the `Frozen` column on `Account` to `LockedFromMerge`.

#### Drop LossRatioColumn from PolicyTermTable

Drops the `LossRatio` column from the `PolicyTerm` table.

#### Drop PrimaryDriver column from VehicleDriver table

Drops the `PrimaryDriver` column from the `VehicleDriver` table.

#### Drop ETLPurgedRoot Table

Drops the `ETLPurgedRoot` table if the table exists and is empty.

#### No duplicate AddressBookUID

Verifies that there are no duplicate rows with matching AddressBookUID in the pc_contact, pc_address, pc_contactaddress and pc_contacttag tables.

#### Dropping deprecated MotorVehicleRecord table

Upgrade drops the `pcx_motorvehiclerecord` table. The MotorVehicleRecord entity has been deprecated since PolicyCenter 7.0.0 and has been removed in PolicyCenter 10.0.0. There is no data migration.

#### See also

- *Motor vehicle records in personal auto* in the *PolicyCenter Application Guide*

### Two-step quoting upgrade triggers

These related upgrade triggers convert the data model for changes introduced by two-step quoting.

#### Add quote maturity level field

This upgrade trigger adds the `QuoteMaturityLevel` column to the `pc_policyperiod` table and populates it based on the value of the existing `ValidQuote` column. If `ValidQuote` is `false`, the trigger sets `QuoteMaturityLevel` to `unrated`. If `ValidQuote` is `true`, the trigger sets `QuoteMaturityLevel` to `quoted`.

#### Drop valid quote column from policy period table

This upgrade trigger drops the `ValidQuote` column from the `pc_policyperiod` table.

## Viewing detailed database upgrade information

PolicyCenter includes a Server Tools **Upgrade and Versions** screen that provides detailed information about each database upgrade. The **Upgrade and Versions** screen includes information on the following:

- Version number of upgrade
- Status of upgrade
- Type of upgrade
- Start and time of upgrade
- Status of Deferred Upgrade Tasks batch processing

From this screen, you can drill down into more specific details about the upgrade, or download a report of the upgrade details.

#### See also

- *System Administration Guide*

## Dropping unused columns on Oracle

By default, the PolicyCenter database upgrade on Oracle marks columns that have been removed from the data model as unused. Marking a column unused is a faster operation than dropping a column. Because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments.

You can configure database upgrade to drop removed columns immediately by setting the `deferDropColumns` parameter to `false` before running the upgrade. This parameter is within the `<upgrade>` block of the `<database>` block of `database-config.xml`.

If you did not set `deferDropColumns` to `true` before the upgrade, perform the procedure to drop unused columns after the upgrade.

You can drop the unused columns after the upgrade during off-peak hours to free the space. PolicyCenter does not have to be shutdown to perform this maintenance task. You can drop all unused columns in one procedure, or you can drop unused columns for individual tables.

### Drop all unused columns on Oracle

#### Procedure

1. Create the following Oracle procedure to purge all unused columns:

```
DECLARE
  dropstr VARCHAR2(100);
  CURSOR unusedcol IS
    SELECT table_name
    FROM user_unused_col_tabs;
BEGIN
  FOR tabs IN unusedcol LOOP
    dropstr := 'alter table '
               ||tabs.table_name
               ||' drop unused columns';
    EXECUTE IMMEDIATE dropstr;
  END LOOP;
END;
```

2. Run the procedure during a period of relatively low activity.

### Drop unused columns for a specific table on Oracle

#### Procedure

1. Start the server to run the schema verifier. The schema verifier runs each time the server starts. If there are unused columns, the schema verifier reports a difference between the physical database and the data model.

The schema verifier reports the name of each table and provides an SQL command to remove unused columns from each table.

2. Run the SQL command provided by the schema verifier. This command has the following format:

```
ALTER TABLE tableName DROP UNUSED COLUMNS
```

## Reload rating sample data

### About this task

If you are using any rating data provided by Guidewire, such as `calcRoutines`, `rateBooks`, `rateTableDefinition`, parameter sets, and so forth, remove all existing rating data, and reload new rating data.

### Procedure

1. Start the PolicyCenter server.
2. Remove the old rating sample data by running the following Gosu script against the database:

```
uses gw.transaction.Transaction
uses gw.api.database.Query

 function findEntity<T extends KeyableBean>() : List<T>{
  var q = Query.make(T)
  q.startsWith("PublicID", "pc:", false /*ignoreCase*/)
  return q.select().toList()
}

 Transaction.runWithNewBundle(\ bundle -> {
  findEntity<RateBook>().each(\ rb -> bundle.add(rb).remove())
  findEntity<RateTableDefinition>().each(\ rt -> bundle.add(rt).remove())
  findEntity<RateTableMatchOpDefinition>().each(\ rb -> bundle.add(rb).remove())
  findEntity<RateFactorRow>().each(\ rb -> bundle.add(rb).remove())
  findEntity<CoverageRateFactor>().each(\ rb -> bundle.add(rb).remove())
  findEntity<CalcRoutineDefinition>().each(\ rb -> bundle.add(rb).remove())
  findEntity<CalcRoutineParameterSet>().each(\ rb -> bundle.add(rb).remove())
}, "su")
```

3. Run the following Gosu script to reload the PolicyCenter 10.0.0 rating sample data:

```
uses gw.transaction.Transaction
uses gw.sampledata.small.SmallSampleRatingData
uses gw.sampledata.tiny.TinySampleRatingData

 Transaction.runWithNewBundle(\ bundle -> {
  var tinyData = new TinySampleRatingData()
  tinyData.load()
  var sampleData = new SmallSampleRatingData()
  sampleData.load()
}, "su")
```

## Export administration data for testing

### About this task

Guidewire recommends that you create a small set of administration data from an upgraded data set. Use this data for development and testing of rules and libraries with PolicyCenter 10.0.0. This procedure is optional.

You might have already created an upgraded administration data set by following the procedure “Upgrade administration data for testing” on page 26. If you followed that procedure, or you do not want an administration-only data set for testing purposes, you can skip this topic.

### Procedure

1. Export administration data from your upgraded production database.

   a. Start the PolicyCenter 10.0.0 server by navigating to `PolicyCenter` and running the following command:

      ```
      gwb runServer
      ```
   b. Open a browser to PolicyCenter 10.0.0.
   c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
   d. Click the **Administration** tab.
   e. Click **Utilities→Export Data**.
   f. Select the **Admin** data set to export.
   g. Click **Export** to download the `admin.xml` file.
2. Create a new database account for the development environment on a database management system supported by PolicyCenter 10.0.0. See the *Supported Software Components* knowledge article for current system and patch level requirements. Visit the Guidewire Community and search for the *Supported Software Components* knowledge article.

   See the *Installation Guide* for instructions to configure the database account.
3. Install a new PolicyCenter 10.0.0 development environment. Connect this development environment to the new database account that you created in "Export administration data for testing" on page 62. See the *PolicyCenter Installation Guide* for instructions.
4. Copy the `admin.xml` file that you exported to a location accessible from the new development environment.
5. Create an empty version of `importfiles.txt` in the `modules/configuration/config/import/gen` directory of the new development environment.
6. Create empty versions of the following CSV files:
   - `activity-patterns.csv`
   - `authority-limits.csv`
   - `reportgroups.csv`
   - `roleprivileges.csv`
   - `rolereportprivileges.csv`

   Leave `roles.csv` as the original complete file.
7. Import the administration data into the new database:
   a. Start the PolicyCenter 10.0.0 development server by navigating to `PolicyCenter` and running the following command:

      ```
      gwb runServer
      ```
   b. Open a browser to PolicyCenter 10.0.0.
   c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
   d. Click the **Administration** tab.
   e. Click **Utilities→Import Data**.
   f. Click **Browse...**.
   g. Select the `admin.xml` file that you exported from the upgraded production database and modified.
   h. Click **Open**.

## Final steps after the database upgrade is complete

After you have completed the upgrade procedure and migration of configurations and integrations, there are a final set of procedures to follow. These procedures provide you with a benchmark of the new system. Completing these steps is particularly important to going live in a production environment.

### Checking that contacts have unique addresses

An `Address` cannot be shared by more than one `Contact`. PolicyCenter 10.0.0 includes a commit-time check that does not allow a shared reference to an address instance even when one of the referring `Contact` or `ContactAddress` instances is retired. If you have multiple contacts at the same address, you can create separate address instances with the same field values.

A database consistency check on the `Contact` entity reports an error if it detects multiple `Contact` records using the same `PrimaryAddress`.

Before using PolicyCenter 10.0.0 in production, run database consistency checks to find any instances of shared references to address instances. If the consistency check reports shared addresses, contact Guidewire Support for assistance fixing your database.

## Completing deferred upgrade

If you have archiving enabled, and you did not set `deferCreateArchiveIndexes` to `false`, run the `Deferred Upgrade Tasks` batch process as soon as possible after the completion of the upgrade. To run the `Deferred Upgrade Tasks` batch process, use the `admin/bin/maintenance_tools` command:

```
maintenance_tools -password password -startprocess deferredupgradetasks
```

## Re-enabling database logging

You might have disabled logging of direct insert and create index operations during the database upgrade. After you complete the database upgrade successfully, you can re-enable logging by setting `allowUnloggedOperations` to `false` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="false">
    ...
  </upgrade>
</database>
```

For SQL Server, if you changed the recovery model from Full to Simple or Bulk logged during the upgrade, you can revert the recovery model. If you deferred migrating to 64-bit IDs, you might disable logging again when you perform the migration.

## Enable the scheduler after upgrade

### About this task

Before you perform these steps, verify that the database upgrade process succeeded.

Before you started the server to upgrade the database, you had to disable the scheduler to prevent batch processes and work queues from launching immediately after the database upgrade. After you have verified that the database upgrade process completed successfully, enable the scheduler again so that batch processes and work queues will resume normal a normal schedule.

### To enable the scheduler

### Procedure

1. Stop the server.
2. Create a new WAR or EAR file that differs from the former only by having `SchedulerEnabled` set to `true`.
   a. Open the PolicyCenter 10.0.0 `config.xml` file in a text editor.
   b. Set the `SchedulerEnabled` parameter to `true`, as follows:
      `<param name="SchedulerEnabled" value="true"/>`
   c. Save `config.xml`.
   d. Create a new WAR or EAR file as appropriate for your application server.
3. Deploy the new WAR or EAR file.
4. Restart the server to activate the scheduler.

#### See also

*Installation Guide*

### Backing up the database after upgrade

Finally, before going live, back up the upgraded database. This provides you with a snapshot of the initial upgraded data set, if an unanticipated event occurs just after going live.

# Upgrading PolicyCenter from 8.0 for ContactManager

This topic lists the manual tasks required to upgrade from PolicyCenter 8.0 and ContactManager 8.0 to PolicyCenter 10.0.0 and ContactManager 10.0.0. Before starting this upgrade process, you must have run the Guidewire upgrade and merge tools. Additionally, Guidewire recommends that you first upgrade ContactManager, integrate PolicyCenter with ContactManager, and refresh the ContactManager web APIs.

## Manually upgrade PolicyCenter to integrate with ContactManager

#### About this task

Use the following steps to upgrade PolicyCenter 8.0 with ContactManager installed.

#### Procedure

1. Run the Configuration Upgrade tools and complete the configuration upgrade for the PolicyCenter configuration. See “Upgrading the PolicyCenter 8.0 configuration” on page 67. Do not make changes yet to the files listed in this topic for PolicyCenter. You make those changes later as described in this topic.
2. Run the Database Upgrade tool to upgrade for the PolicyCenter database. See “Upgrading the PolicyCenter 8.0 database” on page 67.
3. You can perform any manual configuration upgrades except those related to files listed later in this topic. Before making those changes, wait until you configure ContactManager, regenerate its SOAP API, and refresh that API in PolicyCenter Studio.
4. Manually configure ContactManager. See “Upgrading ContactManager from 8.0” on page 110.
5. Make any manual PolicyCenter changes indicated in the next topic.
6. Integrate PolicyCenter and ContactManager as described at the *Guidewire Contact Management Guide*.

## File changes in PolicyCenter related to ContactManager

This topic includes:
- “Changes to AddressBookUID column” on page 109
- “Web service version changes” on page 110
- “Changes to PolicyCenter classes for ContactManager integration” on page 110

### Changes to AddressBookUID column

Entities that use the `AddressBookLinkable` delegate and declare their own `AddressBookUID` columns will have that column removed during the upgrade process.

Further manual changes might be required after this upgrade process has run. The upgrader is not able to detect an `AddressBookLinkable` delegate and an `AddressBookUID` column that are declared in different metadata files. In these cases, you must remove the `AddressBookUID` column from the entity metadata file manually.

#### See also

- *Guidewire Contact Management Guide*

### Web service version changes

PolicyCenter now uses `wsi.remote.gw.webservice.ab.ab900.wsc` to access the ContactManager web service `${ab}/ws/gw/webservice/ab/ab900/abcontactapi/ABContactAPI?wsdl`. See the *Guidewire Contact Management Guide*.

### Changes to PolicyCenter classes for ContactManager integration

For changes to PolicyCenter files used with ContactManager, see the *Guidewire Contact Management Guide*.

## Upgrading ContactManager from 8.0

This topic covers the manual steps needed to perform an upgrade of ContactManager 8.0 to ContactManager 10.0.0. Prior to performing these upgrade steps, you must run the upgrade software and perform automatic upgrades.

### Manually upgrading the ContactManager configuration

Because ContactManager has changed a number of the files used to integrate with the Guidewire applications, it is likely that you will need to manually update configuration files. In particular, you will need to make manual updates:

- If you have made changes to the `ABContact` data model. See the *Guidewire Contact Management Guide*.
- If you have changed any of the files that are referenced in “Merging changed files” on page 110.

This topic includes

- “Upgrade ContactManager 8.0” on page 110
- “Merging changed files” on page 110

### Upgrade ContactManager 8.0

#### Procedure

1. Run the configuration upgrade tool and perform an automatic upgrade of the ContactManager configuration. See “Upgrading the PolicyCenter 8.0 configuration” on page 67.
2. Run the database upgrade tool to upgrade the ContactManager database. See “Upgrading the PolicyCenter 8.0 database” on page 67.
3. Manually configure files in ContactManager. See “Merging changed files” on page 110.
4. Upgrade PolicyCenter as described at “Upgrading PolicyCenter from 8.0 for ContactManager” on page 109.

#### Merging changed files

Web services, APIs, and helper classes have changed between ContactManager 8.0 and 10.0. If you have customized any of these files or classes, you must merge your changes into the corresponding file at the new location.

#### See also

- For ContactManager changes in this release, see in the *Guidewire Contact Management Guide*.

chapter 5

# Upgrading from previous versions

You cannot upgrade to PolicyCenter 10.0.0 directly from a PolicyCenter 7.0 or previous release. This topic gives a brief overview of the process of upgrading from a previous version of PolicyCenter to 10.0.0.

If you are upgrading from PolicyCenter 8.0, see “Upgrading from 8.0” on page 67 instead.

If you are upgrading from PolicyCenter 9.0, see “Upgrading from 9.0” on page 25 instead.

## Upgrade from 7.0 or previous versions

### About this task

In order to upgrade to PolicyCenter 10.0.0 from a PolicyCenter 7.0 or previous release, you must first upgrade your data model and database to the most recent PolicyCenter 9.0 release. It is not necessary to upgrade your entire custom configuration to 9.0 and then test the entire application in 9.0 before upgrading to 10.0.0. This process would add significant effort to the project duration with no tangible benefit.

The process described in the following steps will minimize the effort associated with upgrading the database to 9.0, because only the data model files must be merged. This is the minimum work necessary to be able to start the 9.0 server and trigger the database upgrade to 9.0. All remaining (non data model) files need only be merged once, into the 10.0.0 release.

### Procedure

1. Upgrade all data model files (entities, typelists, field validators, data types) to 9.0.
    a. For **BOTH_ADD** files – Merge Guidewire changes with your custom changes, generally keeping both sets of changes.
    b. For **BOTH_EDIT** files – Merge Guidewire changes with your custom changes, generally keeping both sets of changes.
    c. For **EDIT_DELETE** files – Keep your custom version of the file, but make note of the files that are in this category.

       These files will appear under the **CUSTOMER_ADD** filter during the 10.0.0 merge process. Reevaluate the differences and decide whether to accept the deletion of the file when merging into 10.0.0.
    d. For all other categories of files – Accept all Guidewire changes and accept all your custom changes.
2. Upgrade all other (non data model) files, as follows.
    a. For **BOTH_ADD** files – Keep only your custom version of the file.
    b. For **BOTH_EDIT** files – Keep only your custom version of the file.
    c. For **EDIT_DELETE** files – Keep your custom version of the file, but make note of the files that are in this category.

These files will appear under the **CUSTOMER_ADD** files during the 10.0.0 merge process. In most cases, you will not accept the file into the 10.0.0 configuration. Instead, you will rebuild the desired logic in the appropriate files for the 10.0.0 release.

   **d.** For all other categories of files – Accept all Guidewire changes and accept all your custom changes.

3. Upgrade your database to 9.0, following the directions in the PolicyCenter 9.0 *Database Upgrade Guide*.

## Next steps

Upgrade your configuration and database to 10.0.0, following the directions in the PolicyCenter 10.0.0 *Configuration Upgrade Guide* and the PolicyCenter 10.0.0 *Database Upgrade Guide*.

**Note:** Be sure to follow the instructions above for files that appear as **CUSTOMER_ADD** during your configuration merge to 10.0.0 (due to being carried forward from **EDIT_DELETE** files during your 9.0 configuration upgrade).