RTI Database Integration Service

User's Manual

Version 6.1.0

Printed in U.S.A. First printing.
April 2021.

Trademarks

Copy and Use Restrictions

Technical Support

Real-Time Innovations, Inc.
232 E. Java Drive
Sunnyvale, CA 94089
Phone: (408) 990-7444
Email: support@rti.com
Website: https://support.rti.com/

Contents

Chapter 1 Welcome to RTI Database Integration Service

Welcome to *RTI® Database Integration Service*—a high-performance solution for integrating applications and data across real-time and enterprise systems from RTI.

Database Integration Service is the integration of two complementary technologies: data-centric publish-subscribe middleware and relational database management systems (RDBMS). This powerful integration allows your applications to uniformly access data from real-time/embedded and enterprise data sources via *RTI Connext® DDS*, or via database interfaces. Since both these technologies are data-centric and complementary, they can be combined to enable a new class of applications. In particular, *Connext DDS* can be used to produce a truly decentralized, distributed RDBMS, while RDBMS technology can be used to provide persistence for real-time data.

1.1 Intended Readers

This document presents the general concepts behind *Database Integration Service*'s architecture and provides basic information on how to develop applications using *Database Integration Service*.

The chapters assume general knowledge of relational databases and SQL, familiarity with the ODBC API, IDL and the *Connext DDS* API, and a working knowledge of the C/C++ programming languages.

1.2 Paths Mentioned in Documentation

The documentation refers to:

- **<NDDSHOME>**

  This refers to the installation directory for *RTI® Connext® DDS*. The default installation paths are:

- macOS® systems:
  /Applications/rti_connext_dds-6.1.0
- Linux systems, non-*root* user:
  /home/<*your user name*>/rti_connext_dds-6.1.0
- Linux systems, *root* user:
  /opt/rti_connext_dds-6.1.0
- Windows® systems, user without Administrator privileges:
  <*your home directory*>\rti_connext_dds-6.1.0
- Windows systems, user with Administrator privileges:
  C:\Program Files\rti_connext_dds-6.1.0

You may also see **$NDDSHOME** or **%NDDSHOME%**, which refers to an environment variable set to the installation path.

Wherever you see **<NDDSHOME>** used in a path, replace it with your installation path.

Note for Windows Users: When using a command prompt to enter a command that includes the path **C:\Program Files** (or any directory name that has a space), enclose the path in quotation marks. For example:

```
"C:\Program Files\rti_connext_dds-6.1.0\bin\rtiddsgen"
```

Or if you have defined the **NDDSHOME** environment variable:

```
"%NDDSHOME%\bin\rtiddsgen"
```

- ***<path to examples>***

  By default, examples are copied into your home directory the first time you run *RTI Launcher* or any script in **<NDDSHOME>/bin**. This document refers to the location of the copied examples as ***<path to examples>***.

  Wherever you see ***<path to examples>***, replace it with the appropriate path.

  Default path to the examples:

  - macOS systems: **/Users/<*your user name*>/rti_workspace/6.1.0/examples**
  - Linux systems: **/home/<*your user name*>/rti_workspace/6.1.0/examples**
  - Windows systems: **<*your Windows documents folder*>\rti_workspace\6.1.0\examples**

    Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 10, the folder is **C:\Users\<*your user name*>\Documents**.

Note: You can specify a different location for **rti_workspace**. You can also specify that you do not want the examples copied to the workspace. For details, see *Controlling Location for RTI Workspace and Copying of Examples* in the *RTI Connext DDS Installation Guide*.

1.3 Available Documentation

The following documentation is available for *RTI® Database Integration Service*:

- The Release Notes. This document provides an overview of the current release's features and lists changes since the previous release, system requirements, and supported architectures. (For migration and compatibility information, see the Migration Guide on the RTI Community Portal: https://-community.rti.com/documentation.)
- The Getting Started Guide. This document provides installation instructions, a short 'Hello World' tutorial, and troubleshooting tips.
- This User's Manual starts with an overview of *RTI Database Integration Service*'s basic concepts, terminology, and unique features. It then describes how to develop and implement applications that use *RTI Database Integration Service*.

Additional Resources:

- The *ODBC API Reference* from Microsoft is available from http://msdn.microsoft.com/en-us/library/ms714562(VS.85).aspx.
- The documentation for MySQL™ databases can be found here: http://dev.mysql.com/doc/refman/5.7/en/index.html
- The documentation for PostgreSQL® databases can be found here: https://www.postgresql.org/.

1.4 Background Reading

For information on distributed systems and databases:

- George Coulouris, Jean Dollimore, Tim Kindberg. *Distributed Systems: Concepts and Design (3rd edition)*. Addison-Wesley, 2000
- M. Tamer Ozsu, Patrick Valduriez. *Principles of Distributed Database Systems (2nd Edition).* Prentice Hall, 1999
- Andrew S. Tanenbaum, Maarten van Steen. *Distributed Systems: Principles and Paradigms (1st edition).* Prentice Hall, 2002

For information on real-time systems:

- Qing Li, Caroline Yao. *Real-Time Concepts for Embedded Systems.* CMP Books, 2003
- Doug Abbott. *Linux for Embedded and Real-Time Applications.* Butterworth-Heinemann, 2002
- David E. Simon. *An Embedded Software Primer.* Addison-Wesley, 1999

For information on SQL:

- Joe Celko. *Joe Celko's SQL for Smarties: Advanced SQL Programming (expanded 2nd Edition).* Morgan Kaufmann, 1999
- Rick van der Lans. *Introduction to SQL: Mastering the Relational Database Language (3rd edition).* Addison-Wesley, 1999

For information on ODBC:

- Microsoft Corporation. *Microsoft ODBC 2.0 Software Development Kit and Programmer's Reference.* Microsoft Press, 1997

For information on the C programming language:

- Brian W. Kernighan, Dennis M. Ritchie. *The C programming Language (2nd edition).* Prentice Hall Software Series, 1988

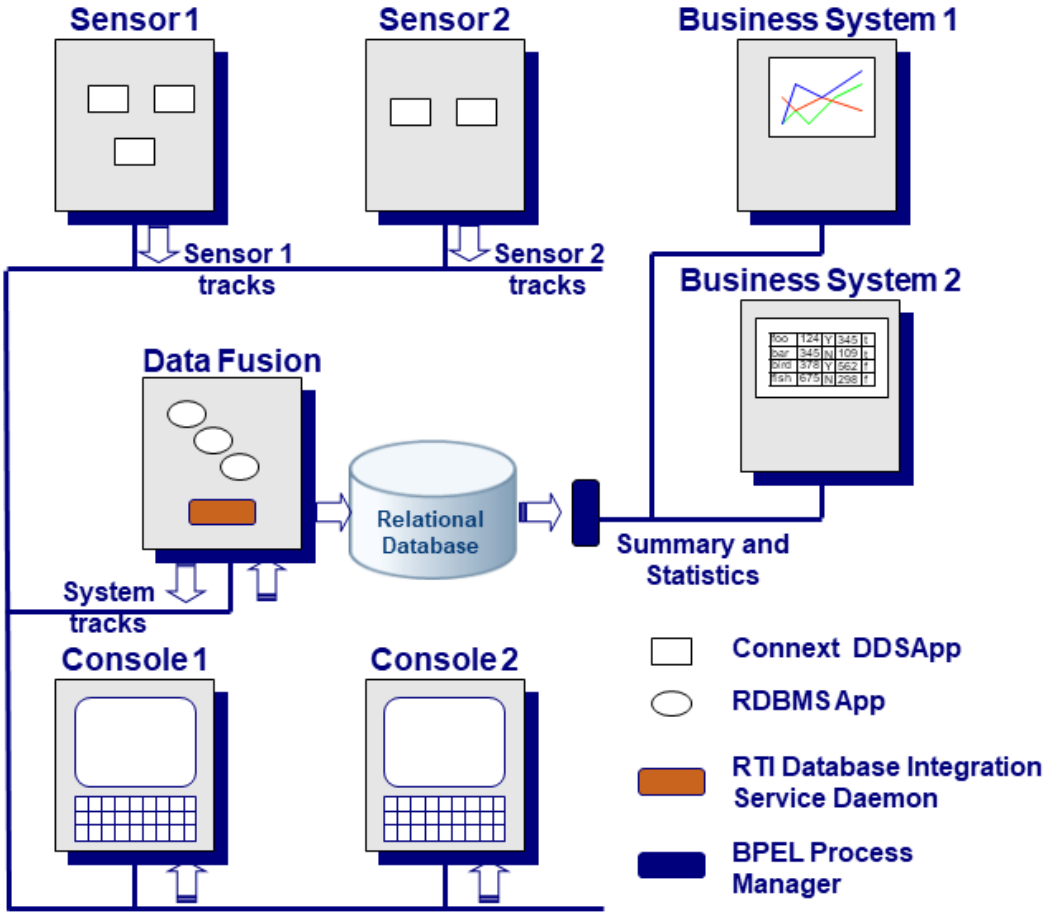
Chapter 2 Introduction

In this section, a few of the unique qualities and features of *Database Integration Service* are discussed in greater detail. Figure 2.1: Example System Using Database Integration Service on the next page shows an example system where *Database Integration Service* serves as the central integration technology to interconnect the real-time, embedded world with the analysis and high-level decision-making processes of the enterprise world.

In Figure 2.1: Example System Using Database Integration Service on the next page, sensors of physical processes produce data that must be filtered, fused, and stored for use in business processes. In addition, multiple user consoles must have concurrent access to both raw and fused data.

Database Integration Service is the bridge that connects real-time/high performance to complex analysis, edge devices to business systems, and embedded to enterprise.

Figure 2.1: Example System Using Database Integration Service



2.1 Interconnecting Standards

Until recently, distributed real-time systems were built using custom-developed data structures and algorithms to store and manipulate data in combination with a commercial, or even proprietary, data-distribution middleware layer. This was necessary to meet real-time performance requirements. However, in recent years, DDS, a standard for data distribution, has emerged as the premier method to integrate and build distributed real-time systems.

For decades in enterprise systems, standards for communications, data representation and data storage has enabled the tremendous growth of software applications for business processes worldwide. The standards such as SQL, ODBC, JMS, HTML, XML, and WDSL have greatly increased the interoperability of those business systems.

Database Integration Service is the first commercial product that interconnects the DDS standard newly established in the embedded world to the common standards of the enterprise world. With *Database Integration Service*, enterprise applications have direct access to real-time data, and real-time applications have

access to the plethora of processes and logic that has been developed to configure and direct actions based on business decisions.

2.2 Connectivity To Edge Devices

For edge devices, such as sensors and hand-helds, *Database Integration Service* integrates *Connext DDS* applications with databases. Applications can publish data into relational databases and subscribe to changes in relational databases using the standard *Connext DDS* application programming interface. Integration between *Connext DDS* and relational database applications is supported by an IDL-to-SQL mapping that allows both types of applications to access a uniform data model.

2.3 Flexibility and Scalability

By leveraging *Connext DDS* Quality-of-Service (QoS) settings, *Database Integration Service* supports an unprecedented variety of deployment configurations to accommodate a wide range of scenarios, from reliable point-to-point delivery to best-effort multicasting that enables real-time transaction streaming to large numbers of subscribers. By setting QoS policies, system throughput, response time, reliability, footprint, and network bandwidth consumption can be tuned to meet application requirements. Previously, a system was hard-coded with parameters set for a specific operation profile during integration. In contrast, *Database Integration Service* provides run-time configurable policy settings, which greatly enhances system deployment flexibility.

2.4 High Availability

Availability is an essential requirement for most distributed real-time applications. Systems built in the Defense and Aerospace industries are typically safety critical and are required to operate in crisis situations. In telecommunications, a minute of system downtime may mean many millions of dollars in lost revenue. With *Database Integration Service*, automatic data caching and replication can serve as the foundation technology for high-availability. Applications can use *Database Integration Service* to maintain copies of SQL database tables on two or more hosts in the network. In the event of a host failure, copies of the tables are available from other hosts to continue operation.

Database Integration Service's automated replication management and no-single-point-of-failure guarantees the availability of critical information. With *Database Integration Service*, tables can be stored on multiple hosts, allowing applications and services to concurrently read and write in multiple tables. Conflict resolution can be based on application-defined timestamps.

2.5 Additional Benefits of Database Integration Service

- Achieve quick time-to-market
    - Start application development immediately using well-known interfaces.
    - Minimize time-consuming custom programming.

    - Easily integrate into existing solutions using industry- standard interfaces.
- Reduce development costs
    - Use widely available modeling and database tools.
    - Eliminate expensive complex coding for real-time data management and communication.
    - Integrate edge devices, distributed real-time data management, and enterprise databases using a single set of standard Application Programming Interfaces.
- Deliver cutting-edge solutions
    - Process massive amounts of information across networks in real-time.
    - Turn near-instantaneous responses to (remote) critical events into a business advantage.
    - Seamlessly integrate networked applications, services, and devices.
- Minimize operational costs
    - Maintain complex networked applications with near-zero administration.
    - Dynamically add or change system components.
    - Run on common hardware platforms and networks.
- Reduce risks
    - Guarantee continuous system availability through dynamic replication management.
    - Rely on continuous high-quality technical support.
    - Build on years of experience in the world’s most demanding real-time application domains.

Chapter 3 Architecture

This chapter presents a more detailed view of the *RTI Database Integration Service* architecture and highlights the different ways that *RTI Database Integration Service* can be used to integrate systems. It includes the following sections:

3.1 Database Integration Service Architecture

The *Database Integration Service* architecture is designed to integrate existing systems that use the *Connext DDS* API or relational databases with minimal modification to working applications. In many situations, existing applications do not have to change at all.

Database Integration Service consists of a daemon that acts like bridge between two software development domains. One uses the OMG Data Distribution Service API to publish and subscribe to data that may be generated at high rates with real-time constraints. The other applies algorithms and logic representing business processes to megabytes, gigabytes or terabytes of data stored in relational databases.

3.1.1 Database Integration Service daemon

The *Database Integration Service* daemon oversees the incoming (subscribed) data and outgoing (published) data. It enables automatic storage of data published by *Connext DDS* applications in a database by mapping a Topic to a table in the database and storing an instance of a Topic as a row in that table. Also, the daemon can automatically publish changes in a database table as a Topic.

Users have total control of the Quality of Service that the daemon uses for publishing and subscribing to *Connext DDS* data.

The *Database Integration Service* daemon uses the *Connext DDS* API, as well as SQL through the ODBC API. In addition, there is a custom interface for each supported database management system (DBMS). The currently supported DBMSs are MySQL and PostgreSQL. There is a separate daemon executable for each of the specific DBMSs.

3.1.2 Database Integration Service's Unique Features

Database Integration Service offers a unique set of features that enable seamless integration of real-time/embedded *Connext DDS* applications and enterprise services:

- **Storage of Connext DDS Data in a DBMS**

  Database Integration Service automatically stores received values of specified Topics in a database. Once the data is propagated to the database, it can be accessed by a user application via regular SQL queries.
- **Publication of DBMS Data via** Connext DDS

  Database Integration Service automatically publishes changes in specified database tables. Changes made via the SQL API (with the INSERT, UPDATE and DELETE statements) will be published into the network via *Connext DDS*, so real-time/embedded applications and devices can respond to time-critical changes with near-zero latency.
- **Mapping Between IDL to SQL Data Types**

  Database Integration Service provides automatic mapping between an IDL data type representation and a SQL table schema representation. This mapping is used to directly translate a table record to a *Connext DDS* data structure and vice-versa. Previously, this translation had to be done by custom-developed code.
- **History**

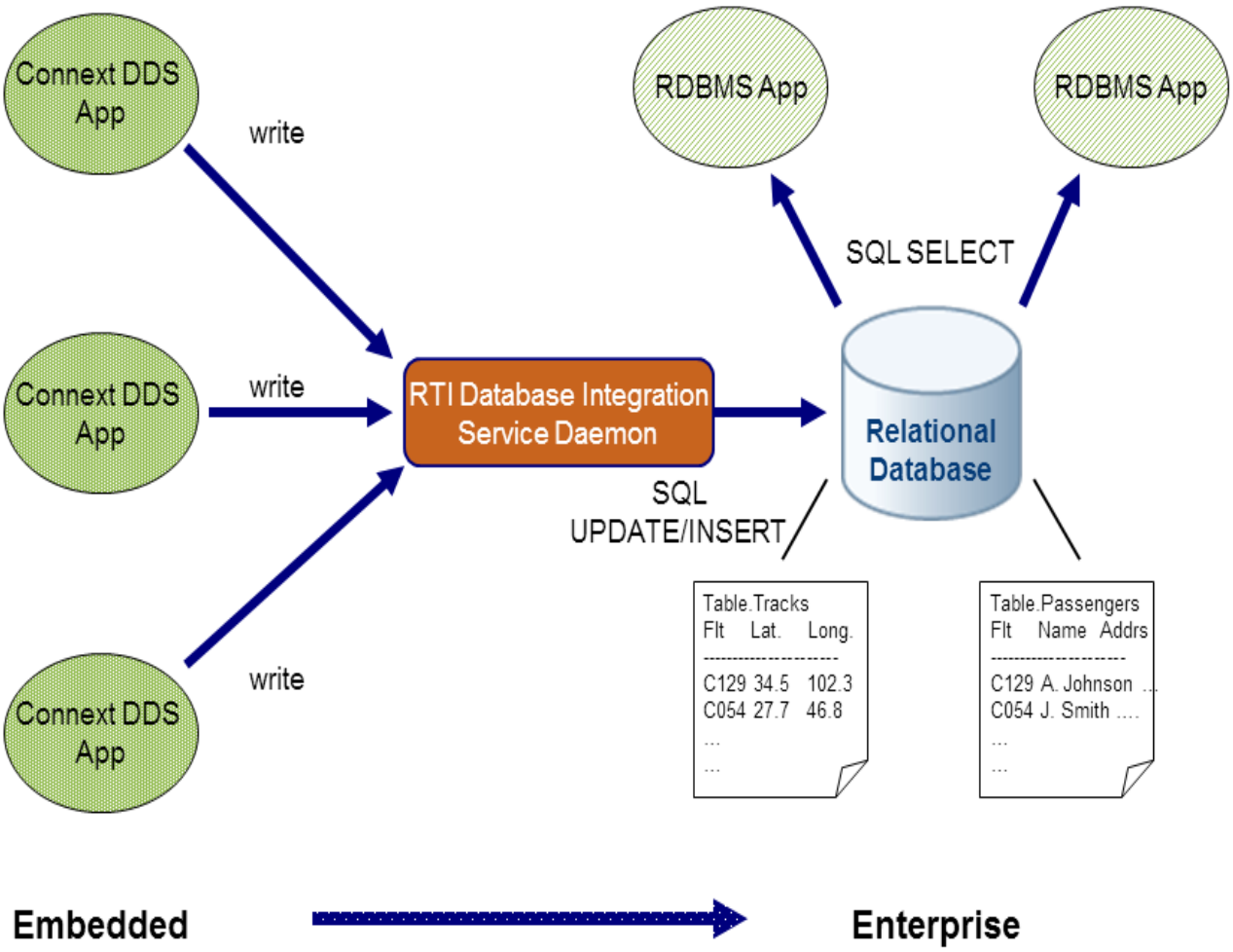
  Database Integration Service can store a history of received values of a data instance. Normally, an instance of a topic is mapped to a single row in the associated database with the IDL key used as the primary key for the table. But when *Database Integration Service*'s data history feature is enabled, multiple samples of a topic instance can be stored across multiple rows in the same table of the database, supporting both real-time and off-line analysis based on historical data.
- **Configurable QoS**

  Database Integration Service exposes many of the QoS attributes defined by the DDS standard. This gives the user full control over the quality of service when capturing real-time data or subscribing to changes in the database. Supported QoS attributes include reliability, durability, multicasting, delivery ordering, and many others.

3.2 Capturing Real-Time Data in a DBMS

Figure 3.1: Storing Published Connext DDS Data in a SQL Database below shows how *Database Integration Service* can be used to capture real-time data streams generated by embedded *Connext DDS* applications into one or more tables in a [in-memory] DBMS. In this scenario, the *Database Integration Service* daemon has been configured with user-customizable QoSs to subscribe to Topics. When new values arrive, the daemon stores the data in the appropriate table in the database. Mapping the Topic described by IDL to the equivalent SQL table schema is done automatically by the daemon with no user configuration necessary.

Figure 3.1: Storing Published Connext DDS Data in a SQL Database

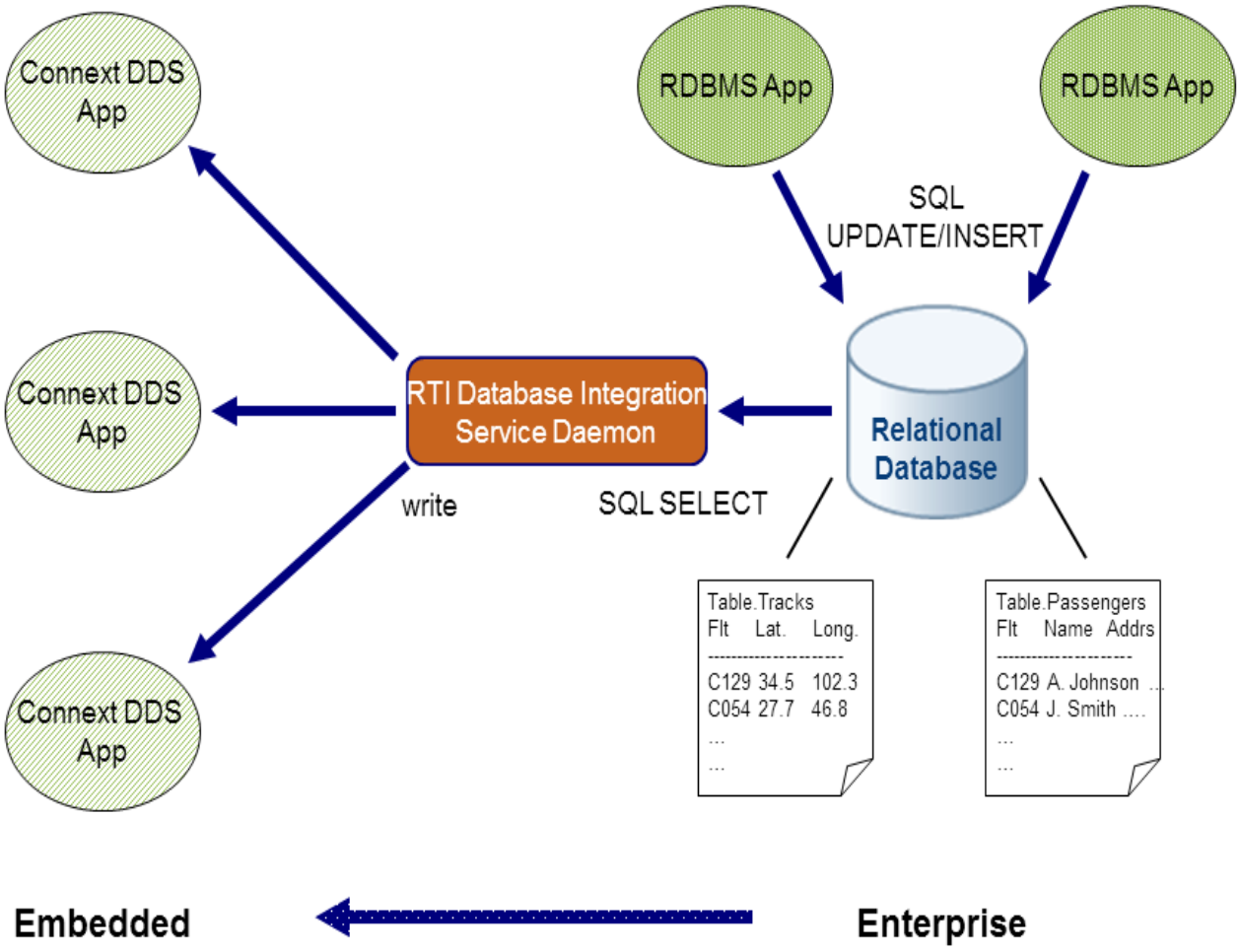


3.3 Remote Real-Time Notification of Table Changes

Figure 3.2: Notifying Remote Applications of Changes in a Database on the next page shows how *Database Integration Service* can be used to notify remote *Connext DDS* applications running in embedded

devices of time-critical changes in the database. In this scenario, the *Database Integration Service* daemon has been configured with user-customizable QoSs to publish Topics whenever the specified table changes in the database. Mapping the SQL table schema to the equivalent Topic described by IDL is done automatically by the daemon—no user configuration necessary.

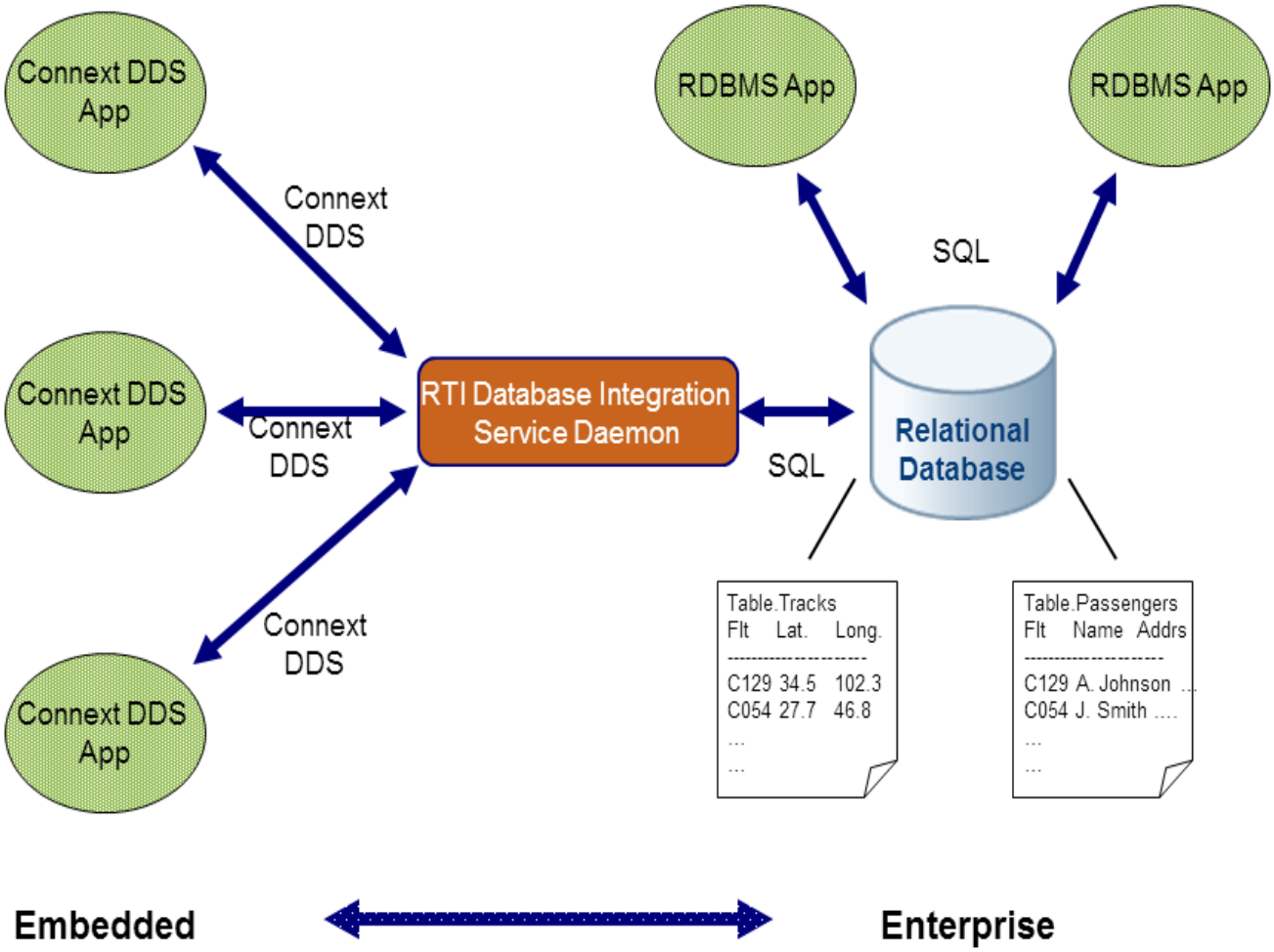
Figure 3.2: Notifying Remote Applications of Changes in a Database



3.4 Bidirectional Integration

Figure 3.3: Bidirectional Integration on the next page shows a system that integrates the capabilities described in the last two sections. *Database Integration Service* provides bidirectional dataflow between embedded *Connext DDS* applications and enterprise database systems. This approach can typically be used to create a closed-loop system, where sensory data is collected, processed, and analyzed in an in-memory database, and the resulting analysis creates state changes that are fed back to remote sensors and devices to control their behavior and mode of operation.

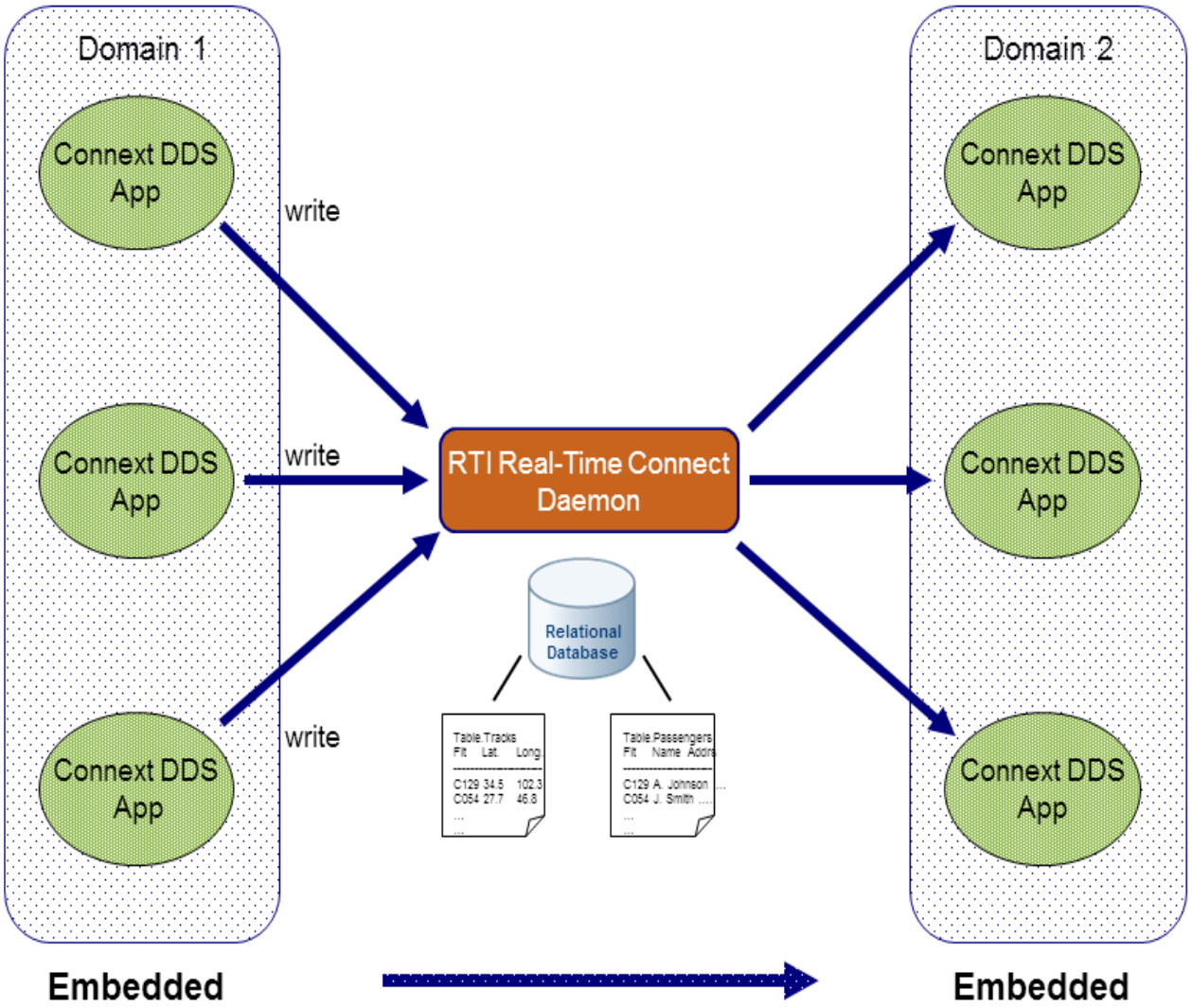
Figure 3.3: Bidirectional Integration



3.5 Bridging between Domains

Figure 3.4: Bridging Domains on the next page shows how *Database Integration Service* can be used as a bridge between two domains by configuring the *Database Integration Service* daemon to subscribe to data in one domain and publishing the same data in a different domain. Data sent by *Connext DDS* applications in the first domain are stored by the daemon in a local in-memory table. Since changes in the table are sent by the daemon into a second domain, the data is ultimately received by *Connext DDS* applications in the second domain. There is no feedback cancellation needed since the data is being bridged across domains. Usually domain bridges have to be written by users and modified whenever data types or Topics change. Using *Database Integration Service*, no programming is required to create a high performance bridge for any topic of any data type between any domains.

Figure 3.4: Bridging Domains

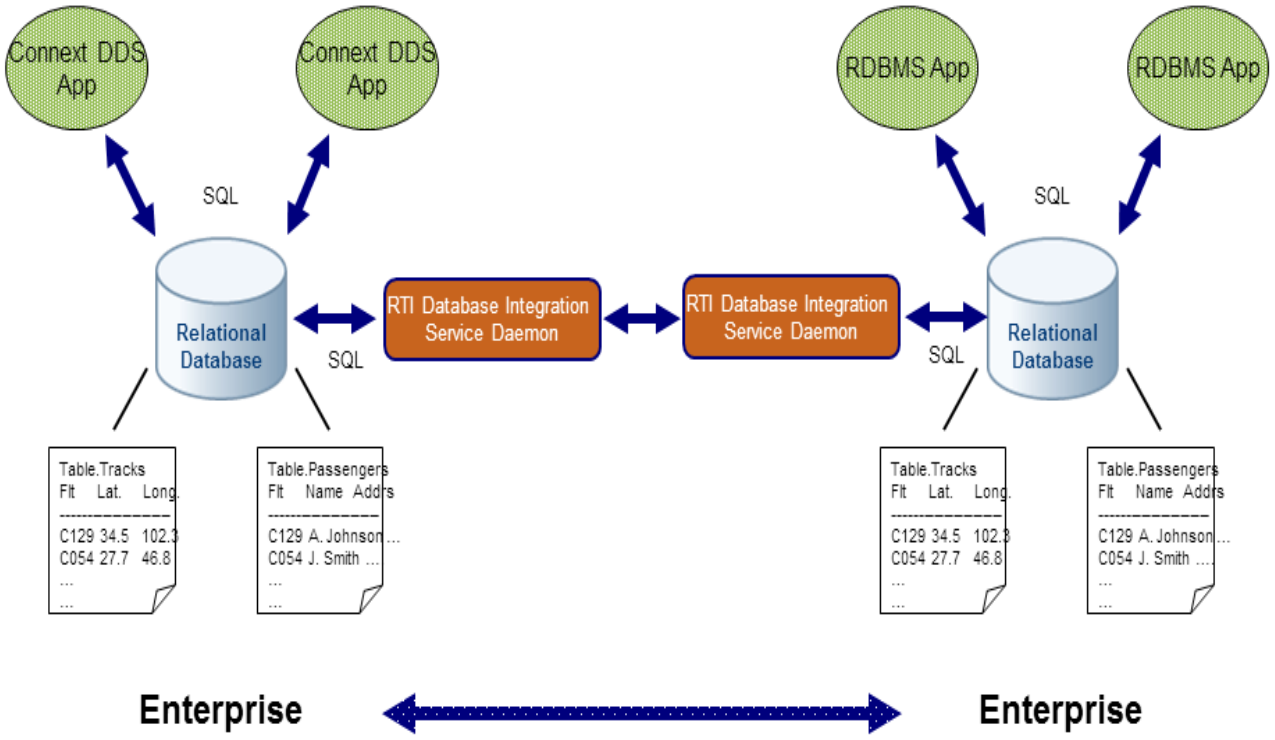


3.6 Real-Time Database Replication

By running multiple *Database Integration Service* daemons on different nodes connected to different databases, and configuring all of the daemons to publish and subscribe to the same table, changes made by applications to a table on one node can be automatically replicated to tables on all of the other nodes. Figure 3.5: Replicating Tables Across Databases on the next page shows how *Database Integration Service* can be used to perform lazy table replication between distributed databases.

With lazy replication, an update is sent to the subscribers after the transaction is committed into the local database. The advantages of lazy replication are short response time and high concurrency, since locks in the data cache are immediately released after a transaction commits and before it is sent on the network.

Figure 3.5: Replicating Tables Across Databases



If you need remote table initialization and application timestamp-based conflict resolution, you can enable this either by using the *Database Integration Service* configuration properties or by setting individual QoS values. This is described further in 4.4 Configuration File on page 25.

Chapter 4 Using Database Integration Service

This chapter provides detailed information on using the *Database Integration Service* daemon to subscribe to and store data received as Topics into relational databases, as well as to publish as Topic changes in relational database tables.

The contents of this chapter assume you have a working knowledge of *Connext DDS* and relational databases. The chapter also assumes familiarity with IDL (Interface Definition Language), the DDS and SQL specifications and APIs. Finally, you should be able to create and run applications using *Connext DDS* to publish and subscribe to data.

Users can configure the *Database Integration Service* daemon to subscribe to Topics and store received values in a table, or to publish database changes as Topics using a combination of methods:

- Command-line parameters
- Environment variables
- Configuration file
- Configuration tables in the database

This chapter includes the following sections:

4.1 Introduction to the Database Integration Service daemon

Database Integration Service bridges the world of *Connext DDS* and the world of relational databases. The main element of the bridge is an executable that must run on the same host as the database management system (DBMS). This executable is called the *Database Integration Service* daemon.

Database Integration Service uses *Connext DDS* and supports these databases: MySQL and PostgreSQL. There is a separate executable that you must run depending on which database you are using.

- MySQL: **rtirtc_mysql[.exe]**
- PostgreSQL: **rtirtc_postgresql[.exe]**

These executables can be executed as foreground processes during development or as background processes or as a service on Windows systems. You can configure the general behavior of the *Database Integration Service* daemon by using command-line parameters, environment variables and configuration files. Meta-tables in the database are used to configure the specific topics and tables that are bridged by the daemon.

Besides using compatible versions of *Connext DDS* and MySQL/PostgreSQL databases (see the *Release Notes* for a list of compatible versions), the *Database Integration Service* daemon expects that *typecodes* for the IDL types used by *Connext DDS* applications have been generated and are being propagated. If typecodes for IDL types were not generated, you must create the tables (used by the daemon for storing or publishing data) yourself or declare the types in the configuration files.

4.1.1 How to Run the Database Integration Service daemon with MySQL

Before *Database Integration Service* will run correctly with a MySQL database, the procedures described in this section must be completed.

4.1.1.1 Installing MySQL ODBC driver

The *Database Integration Service* daemon requires the installation of the MySQL ODBC; see ODBC Driver Requirements, in the RTI Database Integration Service Release Notes for the required version. The driver is not bundled with the MySQL server and must be installed separately.

The ODBC connector can be downloaded from http://dev.mysql.com/downloads/connector/odbc/.

The installation guide can be found at https://dev.mysql.com/doc/connector-odbc/en/connector-odbc-installation.html.

The MySQL ODBC driver requires an ODBC driver manager. On Windows systems, the ODBC driver manager is automatically installed with the OS. For Linux systems, we recommend using UnixODBC 2.2.12 (or higher), a complete, free/open ODBC solution for Linux systems. You can download the driver manager from http://www.unixodbc.org.

4.1.1.2 Installing and Configuring the MySQL Server to Access (lib)rtirtc_mysqlq[.so,.dll]

To work with a MySQL database, there is a shared library distributed with *Database Integration Service* that must be installed correctly on the host of the MySQL database server. Communication by the *Database Integration Service* daemon with the MySQL server is accomplished through user-defined functions (UDF) executed by the MySQL server when triggers installed by the *Database Integration Service* daemon are fired. These functions are provided in a shared library (on Linux systems) or DLL (on Windows systems) called **[lib]rtirtc_mysqlq[.so,.dll]**.

This library is distributed with *Database Integration Service* and can be found in the **lib/<platform>** directory of the installation directory. The correct version of the library to use depends on the platform on which MySQL server is running. For example, **<platform>** can be:

- **x64Linux2.6gcc4.4.5** for a Red Hat Enterprise Linux system
- **x64Win64VS2017** for Windows systems

To install **[lib]rtirtc_mysqlq[.so,.dll]** copy the appropriate version of [**lib]rtirtc_mysqlq[.so,.dll]** into the MySQL server's plugin directory (the directory named by the **plugin_dir** system variable). The plugin directory can be changed by setting the value of **plugin_dir** when the MySQL server is started. For example, you can set its value in the **my.cnf** configuration file:

```
[mysqld]
plugin_dir=/path/to/plugin/directory
```

For additional information about the plugin directory see the following link:
http://dev.mysql.com/doc/refman/5.7/en/install-plugin.html

4.1.1.3 Installing libnddsc[.so,.dll) and libnddscore(.so,.dll) on MySQL Server

Since the library **librtirtc_mysqlq[.so,.dll]** internally uses *Connext DDS*, the corresponding shared libraries **[lib]nddsc[.so,.dll]** and **[lib]nddscore[.so,.dll]** distributed with *Connext DDS* also need to be installed on the MySQL server host.

Notes:

- Please see the Release Notes for specific details of the supported platforms for your release of *Database Integration Service* to MySQL.

- The libraries **nddsc** and **nddscore** must be located in a directory that is searched by the system dynamic linker.
- If you are using a license managed version of *Connext DDS* (e.g., an evaluation installer), make sure that you define the RTI_LICENSE_FILE environment variable to point to a valid license file. Also make sure that this environment variable is visible from the MySQL server process. In order to take effect, on some systems you may need to define it as a system-wide environment variable and restart the MySQL database process. You can find more information about using a license file in the RTI Connext DDS Installation Guide.

Linux Systems:

- Copy the shared libraries to a directory such as **/usr/lib**.
- Alternatively, add the libraries to the environment variable LD_LIBRARY_PATH that must be set for the user who starts the MySQL server. This method requires restarting the MySQL server.

Windows Systems:

- Copy the .dll files to the system directory (**WINDOWS\System32** or **WINDOWS\System**).
- Alternatively, you can add the directories containing the libraries to the **System** variable **Path** as follows:

  Using the dialog opened with **Start, Settings, Control Panel, System, Advanced tab, Environment Variables** button, add the directories (with backslash '\' and semicolon separators '**;**') containing the libraries to the **System** variable "**Path**". If the MySQL server is running as a service, you will need to reboot the computer for this change to take effect.

4.1.1.4 Starting the MySQL Server in ANSI_QUOTES mode

The MySQL server can operate in different sql modes. The *Database Integration Service* daemon requires the MySQL server to be configured in ANSI_QUOTES mode. Under that configuration, the MySQL server treats '**"**' as an identifier quote character instead of a string quote character.

To verify if the MySQL server is already configured in ANSI_QUOTES mode, run the following SQL statement:

```
SELECT @@global.sql_mode;
```

If the string 'ANSI_QUOTES' is not part of the result, the MySQL server needs to be configured in ANSI_QUOTES mode using the option **--sql_mode='ANSI_QUOTES'** to start the server

That same effect can be achieved at runtime by executing the following SQL statement:

```
SET GLOBAL sql_mode = 'ANSI_QUOTES'
```

Note: The specific configuration of the MySQL server may require the use of additional SQL mode strings when starting the server.

4.1.2 How to Run Database Integration Service daemon with PostgreSQL

Before *Database Integration Service* will run correctly with a PostgreSQL database, the PostgreSQL ODBC driver must be installed. The driver is not bundled with the PostgreSQL server and must be installed separately.

You can download the PostgreSQL ODBC driver from https://odbc.postgresql.org/.

The PostgreSQL ODBC driver requires an ODBC driver manager. For Linux systems, we recommend using UnixODBC, a complete, free/open ODBC solution for Linux systems. You can download the driver manager from http://www.unixodbc.org.

4.1.3 How to Run the Database Integration Service daemons as Windows Services

On Windows, you can run the *Database Integration Service* daemons **rtirtc_mysql.exe** and **rtirtc_sqlserver** as system services. To install it as a service, you need to run the daemon with the argument **-installService**:

```
rtirtc_mysql -installService
[SC] CreateService SUCCESS.
```

This will install *Database Integration Service* as a Windows service and you will be able to control it from: **Start, Programs, Administrative Tools, Services application**.

By default, the services will be installed in manual mode. Use the Services application to change this to automatic to have the services start when the Windows machine boots up.

When running *Database Integration Service* as a Windows service, you will need to take two things into account:

- You will need need to setup a System DSN, in order to make the data source accessible to *Database Integration Service*.
- User Applications won't be able to communicate with Database Integration Service daemon using shared memory.

The configuration file used by the Windows services is the default file, **<Database Integration Service installation directory>/resource/xml/RTI_REAL_TIME_CONNECT.xml**.

You can change the location of the configuration file by running the Windows service with the command line option, **-cfgFile** (see 4.2 Command-Line Parameters on page 22).

To uninstall *Database Integration Service* as a Windows service, can use the argument **-uninstallService**.

```
rtirtc_mysql -uninstallService
[SC] DeleteService SUCCESS.
```

4.1.4 Typecode and TypeObject

Database Integration Service uses the type representation for a *Topic* sent over the network to create subscriptions or publications to that *Topic*.

Earlier versions of *Connext DDS* (4.5f and lower) used TypeCodes as the wire representation to communicate types over the network. The OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3 uses TypeObjects as the wire representation. It is possible to configure a *Connext DDS* application to send both TypeCode and TypeObjects. However, sending TypeCodes is disabled by default.

The maximum allowed size for TypeCodes is controlled by the QoS value **DomainParticipantQos::resource_limits.type_code_max_serialized_length**. The maximum allowed size for TypeObjects is controlled by the resource limits **DomainParticipantQos::resource_limits.type_object_max_serialized_length** and **type_object_max_deserialized_length**. To see the default values, check the RTI Connext DDS Core Libraries User's Manual. In *Database Integration Service*, you can change these resource limits using XML QoS Profiles (see Table 4.2 Top-Level Tags).

An important note is that TypeCodes and TypeObjects can become quite large as the corresponding IDL type becomes more complex.

If the *Database Integration Service* daemon discovers *Topics* that have (a) TypeCodes or TypeObjects larger than what it has been configured to handle or (b) have no associated TypeCodes or TypeObjects at all, the daemon will not be able to subscribe to or publish those *Topics* unless you manually create the corresponding tables in the database or define the topic types in the configuration file (see Table 4.2 Top-Level Tags). The only way to determine whether or not this situation exists is to examine the log messages printed by the daemon.

To fix this problem, adjust the resource limits described above in *Database Integration Service* and the *Connext DDS* applications interacting with *Database Integration Service*.

The *Database Integration Service* daemon will store the type information that it receives with discovered *Topics*. This type information may be used by the daemon to create user-accessible tables in the database from which changes are published or where changes received via *Connext DDS* are stored. See 4.5.1 Publications Table on page 46 and 4.5.2 Subscriptions Table on page 59 for more information on how TypeCodes and TypeObjects are used by the daemon.

> **Note:** Throughout this document, the term TypeCode will be used to refer to both TypeCode and TypeObject representations, unless stated otherwise.

4.2 Command-Line Parameters

Any user can start a *Database Integration Service* daemon. The user name/ID and password with which it connects to a database is specified in the configuration file, see Table 4.9 Common Tags for all Database Connections on page 36.

When starting a *Database Integration Service* daemon, the following command-line parameters are supported; the **-cfgName** parameter is required.

For rtirtc_<database> (see below), replace <database> with one of these:

- mysql
- sqlserver
- postgresql

```
Usage: rtirtc_<database> [options]
Options:
 -cfgFile    <file>  Configuration file. This parameter is optional
                     since the configuration can be loaded from
                     other locations
-cfgName    <name>   Configuration name. This parameter is required
                     and it is used to find a <real_time_connect>
                     matching tag in the configuration files
-appName    <name>   Application name
                     Used to name the domain participants
                     Default: -cfgName
-logFile    <file>   Log file
-nodaemon            Run as a regular process. Messages are sent to
                     stdout and stderr
-queueDomainId <int> Domain ID of the channel connecting the MySQL
                     server with RTI Database Integration Service
                     Default: 1
-dbTransport <1|2>   Transport used to communicate the MySQL server
                     with RTI Database Integration Service
                      * 1: UDPv4
                      * 2: Shared Memory
                     Default: 2 (Shared memory)
-heapSnapshotPeriod <sec>
                           Enables heap monitoring. Database Integration
                     Service will generate a heap snapshot every <sec>
                           Default: heap monitoring is disabled
                     Valid range: [1, 86400]
-heapSnapshotDir <dir>
                           When heap monitoring is enabled this parameter
                     configures the directory where the snapshots will
                     be stored. The snapshot filename format is
                     RTI_heap_<appName>_<processId>_<index>.log
                     Default: current working directory
-verbosity  [0-6]    RTI Database Integration Service verbosity
                       * 0 - silent
```

```
                        * 1 - exceptions (Core Libraries and Service)
                        * 2 - warnings (Service)
                        * 3 - information (Service)
                        * 4 - warnings (Core Libraries and Service)
                        * 5 - tracing (Service)
                        * 6 - tracing (Core Libraries and Service)
                       Default: 1 (exceptions)
-version               Prints the RTI Database Integration Service version
-help                  Displays this information
```

Table 4.1 Command-line Options

Option	Description
-appName <application name>	Assigns a name to the *Database Integration Service* execution. The application name is used to set the EntityNameQosPolicy of the *DomainParticipants* created by *Database Integration Service*.
-cfgFile <configuration file>	Specifies an XML configuration file for *Database Integration Service*. The parameter is optional since the *Database Integration Service* configuration can be loaded from other locations. See 4.4.1 How to Load the XML Configuration on page 25 for further details.
-cfgName <configuration name>	**Required** Specifies the name of the configuration to load. The *Database Integration Service* daemon will look for the first tag <real_time_connect> with that name. (See 4.4 Configuration File on page 25.)
-dbTransport <1|2>	This parameter is only available for the **rtirtc_mysql-[.exe]** for MySQL. By default, *Database Integration Service* uses shared memory to communicate with the MySQL database servers. The **-dbTransport** parameter can be used to change the communication transport. There are two possible values: 1: UDPv4 2: Shared memory (default) **Note:** Shared memory communication between the *Database Integration Service* daemon and the database servers does not work on Windows 7 systems when the *Database Integration Service* daemon runs with the option **-nodaemon** and the database server runs as a service. For this use case, communication can be enabled by using UDPv4 as the transport.
-heapSnapshotPeriod	Enables heap monitoring. *Database Integration Service* will generate a heap snapshot every <sec>. Default: heap monitoring is disabled.
-heapSnapshotDir	When heap monitoring is enabled, this parameter configures the directory where the snapshots will be stored. The snapshot filename format is RTI_<configurationName><processId><index>.log. Default: current working directory
-help	Prints out a usage message listing the command-line parameters.
-logFile <log file>	Pathname of the file to be used for log messages. If specified, log messages will automatically be stored in the file.

Table 4.1 Command-line Options

Option	Description
-nodaemon	Start as a normal process. Without this option, running the *Database Integration Service* daemon executable will start a daemon process on Linux systems, or start a service on Windows systems. As a daemon, no log messages of any kind are printed to stdout or stderr. However, by specifying this option, the daemon will start as a regular process, which can be run as a background process using the standard OS with the command-line option ("**&**"), and log messages will be printed to stdout and stderr.
-queueDomainId <domain ID>	This parameter is only available for the **rtirtc_mysql-[.exe]** for MySQL. The *Database Integration Service* daemon uses *Connext DDS* to communicate with the MySQL server. This command-line option sets the domain ID used for the connection between the daemon and the servers. Default: 1
-verbosity <verbosity level>	*Database Integration Service* verbosity level: 0 - No verbosity 1 - Exceptions (*Connext DDS* and *Database Integration Service*) (default) 2 - Warnings (*Database Integration Service*) 3 - Information (*Database Integration Service*) 4 - Warnings (*Connext DDS* and *Database Integration Service*) 5 - Tracing (*Database Integration Service*) 6 - Tracing (*Connext DDS* and *Database Integration Service*) Each verbosity level, **n**, includes all the verbosity levels smaller than **n**. As the *Database Integration Service* daemon runs, it may generate log messages reflecting error conditions, warning messages or general execution status. The messages may be produced by the daemon or by *Connext DDS*. The messages produced by the daemon can be redirected to three possible destinations: stdout/stderr, a file, and log tables in the databases to which it is connected.Each of these destinations may be enabled independently of each other. The first two, stdout/stderr and file, are controlled by command line parameters discussed above, and the last, log table, is controlled in the configuration of a connection, as discussed in 4.4.4.3 Database Connection Options on page 35. In this *Database Integration Service* version, the messages produced by *Connext DDS* can be redirected only to stdout/stderr.
-version	Prints the *Database Integration Service* version.

4.3 Environment Variables

Since the *Database Integration Service* daemon will be making connections to databases using ODBC, on Linux systems, the following environment variables may be used to find DSNs (data source names) via ODBCINI files.

- **ODBCINI**: location of INI file for database connections. If not set, ODBCINI will be set to "**$HOME/.odbc.ini**", where **$HOME** is the home directory of the user who started the daemon.
- **SYSODBCINI**: location of system INI file, used if the DSN is not found in the file specified by **ODBCINI**.

If the *Database Integration Service* daemon cannot find a valid DSN in any ODBC.INI file, then no connections to any databases can be made.

On a Windows system, the equivalent functionality of the ODBCINI file is found in the Windows registry. You create and modify DSNs using the application found in **Start, Programs, Administrative Tools, Data Sources (ODBC)**.

4.4 Configuration File

When you start *Database Integration Service*, you can provide a configuration file in XML format (it is not required). Among other things, this file can be used to specify the set of databases to which the daemon will connect and the properties of the database connections.

This section describes:

4.4.1 How to Load the XML Configuration

Database Integration Service loads its XML configuration from multiple locations. This section presents the various approaches, listed in load order.

The first three locations only contain QoS Profiles and are inherited from *Connext DDS* (see *Configuring QoS with XML*, in the RTI Connext DDS Core Libraries User's Manual).

- **<NDDSHOME>/resource/xml/NDDS_QOS_PROFILES.xml**

  This file contains the *Connext DDS* default QoS values; it is loaded automatically if it exists. *(First to be loaded.)*

  <NDDSHOME> is described in 1.2 Paths Mentioned in Documentation on page 1.

- File in **NDDS_QOS_PROFILES**

  The files (or XML strings) separated by semicolons referenced in this environment variable are loaded automatically.

- **<working directory>/USER_QOS_PROFILES.xml**

  This file is loaded automatically if it exists.

The next locations are specific to *Database Integration Service*.

- **<NDDSHOME>/resource/xml/RTI_REAL_TIME_CONNECT.xml**

  This file contains the default *Database Integration Service* configuration and QoS Profiles; it is loaded if it exists. The default configuration does not work out-of-the-box because it requires setting the parameters that configure the database connections such as dsn, username and password (see 4.4.4 Database Configuration with Database Integration Service XML Tag on page 29).
- **<*working directory*>/USER_REAL_TIME_CONNECT.xml**

  This file is loaded automatically if it exists.
- File specified using the command line parameter **-cfgFile**

  The command-line option **-cfgFile** (see 4.2 Command-Line Parameters on page 22) can be used to specify a configuration file.

You may use a combination of the above approaches.

4.4.2 XML Syntax and Validation

The XML configuration file must follow these syntax rules:

- The syntax is XML; the character encoding is UTF-8.
- Opening tags are enclosed in <>; closing tags are enclosed in </>.
- A tag value is a UTF-8 encoded string. Legal values are alphanumeric characters. The routing service's parser will remove all leading and trailing spaces[1] from the string before it is processed.

  For example, " **<tag> value </tag>**" is the same as "**<tag>value</tag>**".
- All values are case-sensitive unless otherwise stated.
- Comments are enclosed as follows: **<!-- comment -->**.
- The root tag of the configuration file must be **<dds>** and end with **</dds>**.

Database Integration Service provides DTD and XSD files that describe the format of the XML content. We recommend including a reference to one of these documents in the XML file that contains the *Database Integration Service*'s configuration—this provides helpful features in code editors such as Visual Studio and Eclipse, including validation and auto-completion while you are editing the XML file.

[1] Leading and trailing spaces in enumeration fields will not be considered valid if you use the distributed XSD document to do validation at run-time with a code editor.

The DTD and XSD definitions of the XML elements are in
<*Database Integration Service installation directory***>/resource/schema/rti_real_time_connect.dtd**
and
<*Database Integration Service installation directory***>/resource/schema/rti_real_time_connect.xsd**,
respectively.

To include a reference to the XSD document in your XML file, use the attribute **xsi:noNamespaceSchemaLocation** in the **<dds>** tag. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation= "<NDDSHOME>/resource/schema/rti_real_time_connect.xsd">
   ...
</dds>
```

<NDDSHOME> is described in 1.2 Paths Mentioned in Documentation on page 1.

To include a reference to the DTD document in your XML file, use the **<!DOCTYPE>** tag.

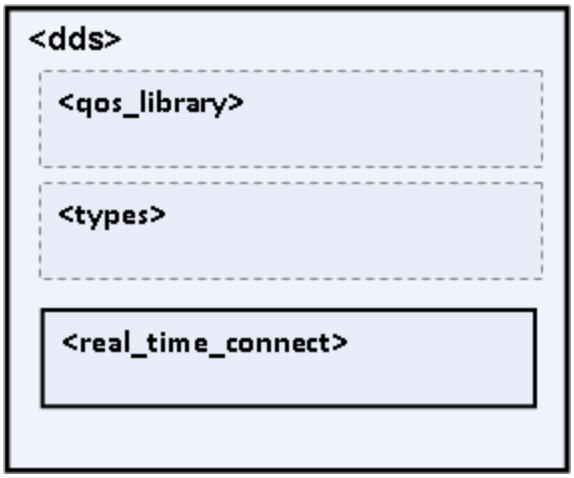
For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dds SYSTEM "<NDDSHOME>/resource/schema/rti_routing_service.dtd">
<dds>
   ...
</dds>
```

We recommend including a reference to the XSD file in the XML documents; this provides stricter validation and better auto-completion than the corresponding DTD file.

4.4.3 Top-Level XML Tags

Let's look at an example configuration file. You will learn the meaning of each line as you read the rest of the sections.



```
<?xml version="1.0"?>
<dds>
    <real_time_connect name="Example">
        <database_mapping_options>
            <identifier_separator_char>$
            </identifier_separator_char>
        </database_mapping_options>
        <mysql_connection>
            <dsn>Example_MySQL</dsn>
            <user_name>Student</user_name>
            <password>mypsswrd</password>
        </mysql_connection>
        <sqlserver_connection>
            <dsn>Example_PostgreSQL</dsn>
            <user_name>Student</user_name>
            <password>mypsswrd</password>
        </sqlserver_connection>
        <postgresql_connection>
            <dsn>Example_PostgreSQL</dsn>
            <user_name>Student</user_name>
            <password>mypsswrd</password>
        </postgresql_connection>
    </real_time_connect>
</dds>
```

Table 4.2 Top-Level Tags describe the top-level tags allowed within the root **<dds>** tag. Notice that the <real_time_connect> tag is required.

Table 4.2 Top-Level Tags

Tags within <dds>	Description	Number of Tags Allowed
<qos_library>	Specifies a QoS library and profiles. The contents of this tag are specified in the same manner as for a *Connext DDS* QoS profile file–see *Configuring QoS with XML*, in the RTI Connext DDS Core Libraries User's Manual. The profiles you specify here can be used in three ways. By setting the attribute **is_default_qos** in the tag <qos_profile> to true. In this case, that profile is the default configuration for all the Entities created by the *Database Integration Service* daemon. By referring to a profile using the XML tag <profile_name> within <publication> and <subscription> (see 4.4.4.4 Initial Subscriptions and Publications on page 38). By referring to a profile in the **profile_name** column of the tables RTIDDS_PUBLICATIONS or RTIDDS_SUBSCRIPTIONS (see 4.5.1 Publications Table on page 46 and 4.5.2 Subscriptions Table on page 59).	0 or more
<real_time_connect>	Specifies a *Database Integration Service* configuration. This tag is used to specify the set of databases to which the daemon will connect. Note: There is no way to dynamically configure the *Database Integration Service* daemon to connect to a database after it has started. All database connections must be specified within this tag before the daemon starts. See Table 4.3 Database Integration Service Tags on page 30 for a description of the elements contained inside <real_time_connect>.	1 or more **(required)**

Table 4.2 Top-Level Tags

Tags within <dds>	Description	Number of Tags Allowed

<types>	Defines types that can be used to create database tables.
	The type description is done using the *Connext DDS* XML format for type definitions. For more information, see Creating User Data Types with Extensible Markup Language (XML) in the chapter on *Data Types and DDS Data Samples*, in the RTI Connext DDS Core Libraries User's Manual.
	For example:
	`<types>`
	`  <struct name="Point">`
	`  <member name="x" type="long"/>`
	`  <member name="y" type="long"/>`
	`  </struct>`
	`</types>`
	Database Integration Service supports automatic table creation by using the types defined within this tag or the typecode sent by *Connext DDS* applications.
	See 4.6 User-Table Creation on page 82 for additional information on user table creation.
		0 or 1

Because a configuration file may contain multiple **<real_time_connect>** tags, one file can be used to configure multiple daemon executions. When you start *RTI Database Integration Service*, you must use the **-cfgName** option to specify which **<real_time_connect>** tag to use.

For example:

```
<dds>
    ...
    <real_time_connect name="rtcA">
        ...
    </ real_time_connect >
    <real_time_connect name="rtcB">
        ...
    </real_time_connect>
    ...
</dds>
```

Starting *Database Integration Service* with the following command will use the <real_time_connect> tag with the name rtcA:

```
rtirtc_mysql -cfgFile example.xml -cfgName rtcA
```

If there is no <real_time_connect> tag matching the name provided with the command line option **–cfgName**, the daemon will report an error and it will list the available configurations.

4.4.4 Database Configuration with Database Integration Service XML Tag

Table 4.3 Database Integration Service Tags describes the tags allowed with the <real_time_connect> section of the XML file.

Table 4.3 Database Integration Service Tags

Tags within <real_time_ connect>	Description	Number of Tags Allowed
<database_mapping_ options>	Configures how the IDL identifier names are mapped to the database column names. See 4.4.4.2 Database Mapping Options on page 34.	0 or 1
<general_options>	Contains attributes that are independent of any particular database connection made by the *Database Integration Service* daemon. See 4.4.4.1 General Options below.	0 or 1
<mysql_connection>	Configures a connection to a MySQL database. See 4.4.4.3 Database Connection Options on page 35.	1 or more **(required)** if running **rtirtc_mysql**; 0 or more (ignored) if running other DBMS version of the daemon
<postgresql_connection>	Configures a connection to a PostgreSQL database. See 4.4.4.3 Database Connection Options on page 35.	1 or more **(required)** if running **rtirtc_postgresql**; 0 or more (ignored) if running other DBMS version of the daemon

4.4.4.1 General Options

Table 4.4 General Options Tags describes the general options; these attributes are independent of any particular database connection made by the *Database Integration Service* daemon.

Table 4.4 General Options Tags

Tags within <general_ options >	Description	Number of Tags Allowed
<administration>	See Table 4.5 Administration Tags	0 or 1
<enable_table_ replication>	*Database Integration Service* can be configured to perform real-time, lazy database replication (see 3.6 Real-Time Database Replication on page 14) by setting this attribute to true. Default: false	0 or 1
<max_objects_ per_thread>	This parameter controls the maximum number of objects per thread that *Connext DDS* can store. If you run into problems related to the creation of Entities, increasing this number may be necessary. See the System Resource Limits QoS policy in the *Working with Domains* chapter of the RTI Connext DDS Core Libraries User's Manual for more information. Default: *Connext DDS* default (2048)	0 or 1

Table 4.4 General Options Tags

Tags within <general_ options >	Description	Number of Tags Allowed
<typecode_ from_table_ schema>	This tag can be used to enable typecode generation from table schemas. If this parameter is set to true and a publication or subscription is created for a database table without an associated typecode, *Database Integration Service* will create the typecode from the table schema. The new typecode will be made available to other *Connext DDS* applications or *Database Integration Service* daemons via discovery traffic. When *Database Integration Service* is used for table replication, the default value for this parameter is true allowing automatic table creation in the replicas. Default: false (except when **enable_table_replication** is set to true).	0 or 1
<wait_set>	Enables the use of Waitsets to store DDS samples in the database tables (see 4.4.4.5 Configuring WaitSets on page 43). Use this option to increase concurrency and scalability. Default: Waitset usage is disabled.	0 or 1

Table 4.5 Administration Tags

Tags within <administration>	Description	Number of Tags Allowed
<distributed_logger>	Configures *RTI Distributed Logger*. See 4.8 Enabling RTI Distributed Logger in Database Integration Service on page 85.	0 or 1
<domain_id>	Specifies which domain ID *Database Integration Service* will use to send log messages when *Distributed Logger* is enabled.	1 (required)
<profile_name>	Determines which QoS profile to use when creating the DomainParticipant that will be used for *Distributed Logger* (when enabled). The value is the fully qualified name of the QoS Profile, represented as a string with this form: <QoS profile library name>::<QoS profile name>	0 or 1

4.4.4.1.1 Enabling Table Replication

Enabling database replication will automatically configure the QoS values of publications and subscriptions to provide conflict resolution and table initialization (see Table 4.6 DataWriter QoS Changes when <enable_table_replication> is True and Table 4.7 DataReader QoS Changes when <enable_table_replication> is True). The attribute also enables automatic table creation (see **<typecode_from_table_schema>** in Table 4.4 General Options Tags) and propagation of NULL values.

Table 4.6 DataWriter QoS Changes when <enable_table_replication> is True

QoS Change	Purpose
reliability.kind = RELIABLE_RELIABILITY_QOS	Enables reliability
destination_order.kind = BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS destination_order.source_timestamp_tolerance.sec = 0 destination_order.source_timestamp_tolerance.nanosec = 0 ownership.kind = SHARED_OWNERSHIP_QOS	Performs conflict resolution
protocol.serialize_key_with_dispose = true writer_data_lifecycle.autodispose_unregistered_instances = false	Propagates delete operations
durability.kind = TRANSIENT_LOCAL_DURABILITY_QOS	Sends table contents to late joiners (table initialization)
history.depth = 1 history.kind = KEEP_LAST_HISTORY_QOS	Keeps one record per primary key value
writer_resource_limits.instance_replacement = DDS_DISPOSED_INSTANCE_REPLACEMENT writer_resource_limits.replace_empty_instances = DDS_BOOLEAN_FALSE	Enables replacement of deleted rows

Table 4.7 DataReader QoS Changes when <enable_table_replication> is True

QoS Change	Purpose
reliability.kind = RELIABLE_RELIABILITY_QOS	Enables reliability
destination_order.source_timestamp_tolerance.sec = DURATION_INFINITE_SEC destination_order.source_timestamp_tolerance.nanosec = DURATION_INFINITE_ NSEC ownership.kind = SHARED_OWNERSHIP_QOS;	Performs conflict resolution Note: <enable_table_replication> sets some QoS related to conflict resolution, but it does not enable the feature. See 4.4.4.1.2 Conflict Resolution on the next page for additional details
protocol.propagate_dispose_of_unregistered_instances = true	Enables propagation of delete operations
durability.kind = TRANSIENT_LOCAL_DURABILITY_QOS;	Sends table contents to late joiners (table initialization)
history.kind = KEEP_LAST_HISTORY_QOS;	Keeps one sample per primary key value

These QoS changes have priority over the values set using QoS Profiles. However, they can be overwritten per publication and per subscription by setting the corresponding fields in the RTIDDS_PUBLICATIONS and RTIDDS_SUBSCRIPTIONS meta tables (see 4.5.1 Publications Table on page 46 and 4.5.2 Subscriptions Table on page 59).

4.4.4.1.2 Conflict Resolution

Because there are no global (network-wide) locks on records when a transaction is being executed, conflicts can occur. The best way to avoid conflicts is to have only one host modify a specific row (instance) or table (topic), but that is not always possible. The second best way is to design the application in such a way that conflicts can never occur, for instance due to data flow dependencies. But that also is often hard to achieve.

By default, conflict resolution is not enabled when you set **<enable_table_replication>** to true. You can enable conflict resolution by setting the column **dr.destination_order.kind** in **RTIDDS_SUBSCRIPTIONS** to **BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS** (see 4.5.2.1.18 dr.destination_order.kind on page 73). With this setting, eventual consistency can be guaranteed. Conflicts can cause a temporary inconsistency between the databases, but eventually these are resolved by the *Database Integration Service* conflict-resolution mechanism. By default, conflicts are resolved using a timestamp corresponding to the system time when the update occurred. You can overwrite this behavior by providing your own timestamp in a separate database column (see 4.5.1.1.6 resolution_column on page 52).

If you do not need conflict resolution, you can disable it by setting the column **dr.destination_order.kind** in **RTIDDS_SUBSCRIPTIONS** to **BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS**.

4.4.4.1.3 Table Initialization

When a host starts using a table in the distributed shared database, it is essential that the local table is up-to-date. *Database Integration Service* supports two approaches to filling the local table's contents:

1. If all the rows in the table are updated frequently, it is sufficient to apply these updates to the data cache.
2. The table can be synchronized by explicitly requesting the table's contents from the other hosts. This is called table initialization.

If table initialization is not needed, you can disable it by setting the columns **dw.durability.kind** in RTIDDS_PUBLICATIONS and **dw.durability.kind** in RTIDDS_SUBSCRIPTIONS to VOLATILE_DURABILITY_QOS.

4.4.4.2 Database Mapping Options

Table 4.8 Database Mapping Options describes the options that are allowed with the **<database_mapping_options>** tag.

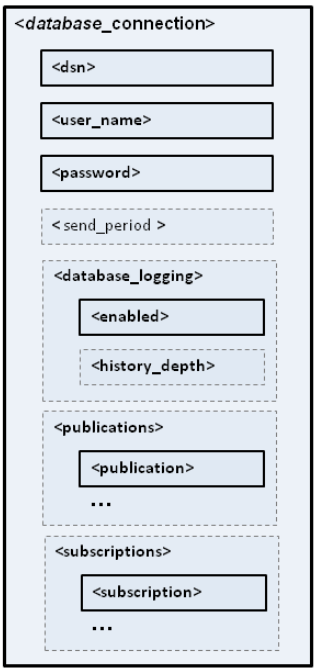
Table 4.8 Database Mapping Options

Tags within <database_mapping_options>	Description	Number of Tags Allowed
<close_bracket_char>	Sets the closing bracket character that is used in the index component of the arrays and sequences members names. See 5.2.6 Array Fields on page 98 and 5.2.7 Sequence Fields on page 98 for more information about the mapping of IDL arrays and sequences into SQL columns. The default value of ']' will generate columns names that must be referenced using double quotes. Default: ']'	0 or 1
<identifier_separator_char>	Sets the character that is used as a separator in the hierarchical names generated when mapping IDL fields into SQL table columns. The attribute is also used to configure the separator character for the columns in the meta tables. Default: '$' for MySQL	0 or 1
<idl_member_prefix_max_length>	Controls the prefix length of the IDL member identifiers that will be used to truncate column names when a table is automatically created. If the default value (-1) is used, *Database Integration Service* will not truncate IDL member identifiers when these are used to create column names. If a positive value, *n,* is provided, *Database Integration Service* will use the first *n* characters from the IDL member identifier to compose the associated column name. A value of 0 tells *Database Integration Service* to compose the column name using only the last characters of the identifiers, as defined by **<idl_member_suffix_max_length>**. This value can be overridden per table by assigning a value to the **idl_member_prefix_max_length** column in the meta-tables. Default: -1 (unlimited)	0 or 1
<idl_member_suffix_max_length>	Controls the suffix length of the IDL member identifiers that will be used to truncate column names when a table is automatically created. If a positive value, *n,* is provided, *Database Integration Service* will use the last *n* characters from the IDL member identifier to compose the associated column name. A value of 0 tells *Database Integration Service* to compose the column name using only the first characters of the identifiers, as defined by **<idl_member_prefix_max_length>**. This value can be overridden per table by assigning a value to the **idl_member_suffix_max_length** column in the meta-tables. Note that although **<idl_member_prefix_max_length>** and **<idl_member_suffix_max_length>** can be individually set to zero, they cannot be both zero at the same time. Default: -1 (unlimited)	0 or 1

Table 4.8 Database Mapping Options

Tags within <database_mapping_options>	Description	Number of Tags Allowed
<open_bracket_char>	Sets the opening bracket character that is used in the index component of the arrays and sequences members names. See 5.2.6 Array Fields on page 98 and 5.2.7 Sequence Fields on page 98 for more information about the mapping of IDL arrays and sequences into SQL columns. The default value of '[' will generate columns names that must be referenced using double quotes. Default: '['	0 or 1

4.4.4.3 Database Connection Options



The database connection tags in the XML file direct the *Database Integration Service* daemon to connect to a particular database as specified by a DSN (data source name) and configure the connection.

The database connection tags are DBMS-specific:

- <mysql_connection>
- <sqlserver_connection>
- <postgresql_connection>

A **<real_time_connect>** tag may have multiple database connection tags. The DBMS-specific *Database Integration Service* daemon will only parse the tags that apply to it. As explained earlier, the *Database Integration Service* daemon will make a connection to a database using the DSN attribute for every connection tag that it parses. This is the only way to direct the daemon to connect to a database. No other connections will be made after startup.

Example:

```
<real_time_connect name="MyRtc">
    <mysql_connection>
        <dsn>Example_MySQL</dsn>
        <user_name>Student</user_name>
        <password>mypsswrd</password>
        <send_period>100</send_period>
        <database_logging>
            <enabled>true</enabled>
            <history_depth>100</history_depth>
        </database_logging>
        <publications>
            <publication>...</publication>
```

```
        </publications>
        <subscriptions>
            <subscription>...</subscription>
        </subscriptions>
    </mysql_connection>
</real_time_connect>
```

Table 4.9 Common Tags for all Database Connections describes tags allowed within all three types of ***<database_connection>*** tags. Table 4.10 through Table 4.12 describe additional tags for each connection type.

Table 4.9 Common Tags for all Database Connections

Common Tags for <mysql_connection>, <sqlserver_connection>, <postgresql_connection>	Description	Number of Tags Allowed
<database_logging> <enabled> <history_depth>	If enabled, the *Database Integration Service* daemon's log messages will be stored in a table named "RTIRTC_LOG" in the database specified by the DSN. Optionally, you can specify the history depth of the log. This value limits the size of the table, RTIRTC_LOG, in the database that the daemon uses for logging messages. The default is 1000 rows, and a value of -1 implies no limit. When the table is filled, new log messages will replace the oldest messages, effectively using the table as a circular buffer. Default: disabled	0 or 1
<dsn>	You must specify a valid DSN that is found in a ODBCINI file or the Windows registry (see 4.3 Environment Variables on page 24). The *Database Integration Service* daemon will make a connection to this DSN.	1 **(required)**
<password>	Specifies the password to connect to the database.	1 **(required)** for all database connections
<publications>	This tags allows inserting publications in the table RTIRTC_PUBLICATIONS when the daemon starts up. See 4.4.4.4 Initial Subscriptions and Publications on page 38.	0 or 1
<send_period>	The **send_period** value specifies the milliseconds interval at which the *Database Integration Service* daemon publishes database changes. The value must be greater than or equal to 0. With a value of 0 the daemon publishes database changes as soon as they are available. A shorter time interval reduces latency. Default: 100 ms	0 or 1
<subscription_default_settings>	Configures the default settings for subscriptions inserted under the <subscriptions> tags. See Table 4.13 Default Subscriptions Settings	0 or 1
<subscriptions>	This tags allows inserting subscriptions in the table RTIRTC_SUBSCRIPTIONS when the daemon starts up. See Table 4.14 Subscriptions Tags on page 40.	0 or 1
<user_name>	Specifies the user name to connect to the database. This attribute is mandatory for all databases.	1 **(required)** for all database connections

Table 4.10 Tags for MySQL Connections

Additional Tags Allowed within <mysql_ connection>	Description	Number of Tags Allowed
<transaction_ max_duration>	Provides an estimation of the maximum duration of a database transaction. If a table change is not committed in the interval specified by this attribute, it will not be published to *Connext DDS*. Uncommitted table changes are stored in a per-table queue. The maximum size of that queue can be configured setting the value of the **changes_queue_maximum_size** column in the RTIDDS_PUBLICATIONS table (see 4.5.1.1.18 changes_queue_maximum_size on page 58). If a change in the uncommitted changes queue has not been committed after **transaction_max_duration** milliseconds, it will be discarded by the *Database Integration Service* daemon. With a value of -1, the *Database Integration Service* daemon will not discard changes into the uncommitted queue until they are committed. Default: 5000	0 or 1

Table 4.11 Common Tags for all MySQL and PostgreSQL Connections

Common Tags for <mysql_ connection>, <postgresql_ connection>	Description	Number of Tags Allowed
<json_default_size>	Specifies the size in bytes of the buffer preallocated to store a sample in JSON format. By default, when you configure a subscription to store a sample in JSON format using the table_schema field (see 4.5.2.1.30 table_schema on page 78), *Database Integration Service* preallocates one buffer per subscription to store the JSON representation of the sample before it gets into the database. The size of the buffer is configured by this parameter. If the JSON representation of the incoming sample is greater than this value, *Database Integration Service* allocates a new buffer from the heap that is released after the sample is added to the database. Default: 65000 bytes	0 or 1

Table 4.12 Tags for PostgreSQL Connections

Additional Tags Allowed within <postgresql_connection>	Description	Number of Tags Allowed
<force_autocommit>	When set to true, this parameter forces autocommit mode when storing DDS samples into a table. When autocommit mode is enable, the setting <commit_type> configured per <subscription> (see 4.5.2.1.6 process_batch, process_period, commit_type on page 67) is not relevant. The difference between setting this parameter to true versus commit_type to COMMIT_ON_SAMPLE is that in the first case *Database Integration Service* will not call explicitly COMMIT for every sample added to the table. For PostgreSQL, setting this value to true seems to provide better performance. Default: false	0 or 1

4.4.4.4 Initial Subscriptions and Publications

As explained in 4.5 Meta-Tables on page 45, the daemon is configured to publish and subscribe to data in the database by inserting entries in two meta-tables, RTIDDS_PUBLICATIONS and RTIDDS_SUBSCRIPTIONS. In your XML configuration you can specify initial values to insert in these tables.

For example:

```
<mysql_connection>
...
    <subscriptions delete="true">
        <subscription>
            <table_owner>user</table_owner>
            <table_name>mytable1</table_name>
            <domain_id>54</domain_id>
            <topic_name>mytopic1</topic_name>
            <type_name>mytype1</type_name>
        </subscription
        ...
        <subscription>
            ...
        </subscription>
        ...
    </subscriptions>
    <publications>
        <publication overwrite="true">
            <table_owner>user</table_owner>
            <table_name>mytable2</table_name>
            <domain_id>54</domain_id>
            <topic_name>mytopic2</topic_name>
            <type_name>mytype2</type_name>
        </publication>
        <publication>
        ...
```

```
        </publication>
        ...
    </publications>
</mysql_connection>
```

Within **<subscriptions>** and **<publications>** tags, you can specify as many **<subscription>** and **<publication>** tags as you want. The content of each tag inside **<subscription>/<publication>** represents the value for a column with the same name in the table RTIDDS_SUBSCRIPTIONS/RTIDDS_PUBLICATIONS. Each of these **<subscription>/<publication>** tags may result in the insertion or update of a row in the corresponding meta-table.

All the rows in the tables can be deleted before inserting new rows if the attribute "delete" in **<publications>/<subscriptions>** is set to true.

If a **<publication>** or **<subscription>** already exists in its table (the primary key is the same), then the insertion won't succeed. However you can set the attribute "overwrite" to true. In that case, if the insertion fails, an update is performed on that row.

Table 4.13 Default Subscriptions Settings

Tags Allowed within <subscription_default_settings>	Description	Number of Tags Allowed
<table_owner>	See 4.5.2 Subscriptions Table on page 59	0 or 1
<domain_id>		
<cache_initial_size>		
<cache_maximum_size>		
<commit_type>		
<delete_on_dispose>		
<filter_duplicates>		
<idl_member_prefix_max_length>		
<idl_member_suffix_max_length>		
<ordered_store>		
<persist_state>		
<process_batch>		
<process_period>		
<profile_name>		
<table_history_depth>		
<table_schema>		
<metadata_fields>		

Table 4.14 Subscriptions Tags

Tags Allowed within <subscriptions>	Description	Number of Tags Allowed
<subscription>	Configures a subscription by inserting or updating a row in the table RTIDDS_SUBSCRIPTIONS. See Table 4.15 Subscription Tags.	1 or more

Note that there are columns in the tables RTIDDS_PUBLICATIONS and RTIDDS_SUBSCRIPTIONS that don't have a corresponding tag inside **<publications>** and **<subscriptions>**. Those columns represent configuration of QoS. However, you can configure the quality of service by using **<profile_name>**,

where you can refer to a QoS profile in your own XML configuration file or in any of the other QoS profile files loaded by the daemon (see 4.4.1 How to Load the XML Configuration on page 25).

Table 4.15 Subscription Tags

Tags Allowed within <subscription>	Description	Number of Tags Allowed
<domain_id>	Inserts the tag value into the column with the same name in the table RTIDDS_SUBSCRIPTIONS. See 4.5.2 Subscriptions Table on page 59.	1 **(required)**
<table_name>		
<table_owner>		
<topic_name>		
<cache_initial_size>	Inserts the tag value into the column with the same name in the table RTIDDS_SUBSCRIPTIONS. If the value is not specified, the corresponding value in <subscription_default_settings> (see Table 4.13 Default Subscriptions Settings) is used if specified. Otherwise, NULL is inserted. See 4.5.2 Subscriptions Table on page 59.	0 or 1
<cache_maximum_size>		
<commit_type>		
<delete_on_dispose>		
<filter_duplicates>		
<idl_member_prefix_max_length>		
<idl_member_suffix_max_length>		
<ordered_store>		
<persist_state>		
<process_batch>		
<process_period>		
<profile_name>		
<table_history_depth>		
<type_name>		
<table_schema>	Configure how a DDS sample will be stored into a database table. See 4.5.2.1.30 table_schema on page 78	0 or 1
<metadata_fields>	Controls whether or not metadata is stored into the UserTables. This feature is only supported in MySQL and PostgreSQL. See Table 4.18 Metadata Tags.	0 or 1

Table 4.16 Publications Tags

Tags Allowed within <publications>	Description	Number of Tags Allowed
<publication>	Configures a publication by inserting or updating a row in the table RTIDDS_PUBLICATIONS. See Table 4.17 Publication Tags.	1 or more

Table 4.17 Publication Tags

Tags Allowed within <publication>	Description	Number of Tags Allowed
<domain_id>	Inserts the tag value into the column with the same name in the table RTIDDS_PUBLICATIONS. See 4.5.1 Publications Table on page 46	1 **(required)**
<table_name>		
<table_owner>		
<topic_name>		
<idl_member_prefix_max_length>	Inserts the tag value into the column with the same name in the table RTIDDS_PUBLICATIONS. If the value is not specified, NULL is inserted. See 4.5.1 Publications Table on page 46	0 or 1
<idl_member_suffix_max_length>		
<resolution_column >		
<profile_name>		
<table_history_depth>		
<type_name>		
<metadata_fields>	Controls whether or not metadata is stored into the UserTables. This feature is only supported in MySQL. See Table 4.18 Metadata Tags.	0 or 1

Table 4.18 Metadata Tags

Tags Allowed within <metadata_fields>	Description	Number of Tags Allowed
<timestamp_type>	Insert the tag value into the column **metadata.timestamp_type** in RTIDDS_SUBSCRIPTIONS and RTIDDS_PUBLICATIONS. If the value is not specified, NULL is inserted. See 4.5.2.1.28 metadata.timestamp_type on page 77.	0 or 1
<included>	This tag specifies the included metadata fields for user topic tables as a sequence of field names. These names will be concatenated, comma-separated, and stored in the **metadata.include_fields** column in RTIDDS_SUBSCRIPTIONS and RTIDDS_PUBLICATIONS. For example: <included> <field>SOURCE_TIMESTAMP</field> <field>RECEPTION_TIMESTAMP</field> </included> If the value is not specified, NULL is inserted. See 4.5.2.1.29 metadata.include_fields, metadata.exclude_fields on page 78	0 or 1
<excluded>	This tag specifies the excluded metadata fields for user topic tables as a sequence of field names. These names will be concatenated, comma-separated, and stored in the **metadata.exclude_fields** column in RTIDDS_SUBSCRIPTIONS and RTIDDS_PUBLICATIONS. For example: <excluded> <field>RECEPTION_TIMESTAMP</field> </excluded> If the value is not specified, NULL is inserted. See 4.5.2.1.29 metadata.include_fields, metadata.exclude_fields on page 78	0 or 1

4.4.4.5 Configuring WaitSets

By default, *Database Integration Service* stores DDS samples in database tables using the DataReader's **on_data_available()** listener callback. This limits the scalability of the service because the thread invoking the callback (DDS receive thread) is shared by all the DataReaders created by the service. With the listener approach, using a different thread per DataReader to increase concurrency requires changing the receive port for each DataReader, which poses a usability issue. For example:

```
<qos_library name="Library1">
  <qos_profile name="Profile1">
    <datareader_qos>
      <unicast>
        <value>
          <element>
            <receive_port>6782</receive_port>
```

```
          </element>
        </value>
      </unicast>
    </datareader_qos>
  </qos_profile>
  <qos_profile name="Profile2">
    <datareader_qos>
      <unicast>
        <value>
          <element>
            <receive_port>6783</receive_port>
          </element>
        </value>
      </unicast>
    </datareader_qos>
   </qos_profile>
</qos_library>
<real_time_connect name="default">
  <postgresql_connection>
    <subscriptions>
        <subscription>
          <table_name>Table1</table_name>
          <profile_name>Library1::Profile1</profile_name>
        </subscription>
        <subscription>
          <table_name>Table2</table_name>
          <profile_name>Library1::Profile2</profile_name>
        </subscription>
      </subscriptions>
   </postgresql_connection>
</real_time_connect>
```

The above example creates two DDS receive threads to store samples in Table1 and Table2. The storage is concurrent. However, this configuration requires assigning a separate receive port to each one of the subscriptions. In addition, the solution is not friendly to multicast configuration, since assigning two different multicast ports would cause data to be sent twice.

You can use the <wait_set> tag to switch from a callback model to a WaitSet in order to read and store data in the database tables. When <wait_set>/<enabled> is set to true, *Database Integration Service* creates a thread pool to process the samples read using a WaitSet. The number of threads in this pool can be configure using <wait_set>/<thread_pool_size>; this is what provides concurrency when storing samples in multiple tables. The above example could be rewritten as following using a WaitSet:

```
<real_time_connect name="default">
  <postgresql_connection>
    <general_options>
      <wait_set>
        <enabled>true</enabled>
        <thread_pool_size>3</thread_pool_size>
      </wait_set>
    </general_options>
    <subscriptions>
```

```
        <subscription>
          <table_name>Table1</table_name>
        </subscription>
        <subscription>
          <table_name>Table2</table_name>
        </subscription>
      </subscriptions>
   </postgresql_connection>
</real_time_connect>
```

For more information, see Conditions and WaitSets in the *DDS Entities* chapter of the RTI Connext DDS Core Libraries User's Manual.

Table 4.19 WaitSet Tags

Tags within <wait_set>	Description	Number of Tags Allowed
<enabled>	Enables the usage of a DDS WaitSet to read data when set to true. Default: false	0 or 1
<thread_pool_size>	Configures the number of threads that will process and store samples concurrently. Increase this number to increase scalability. Default: 1	0 or 1
<max_event_ count>	Maximum number of trigger events to cause the WaitSet to awaken. Default: 1	0 or 1
<max_event_delay>	Maximum delay from occurrence of first trigger event to cause the DDS WaitSet to awaken. For example: <max_event_delay> <sec>1</sec> <nanosec>0</nanosec> </max_event_delay> Default: DDS_DURATION_INFINITE	0 or 1
<wait_timeout>	Maximum duration for the wait. If this duration is exceeded and no samples have been recieved, the DDS WaitSet will wake up. For example: <wait_timeout> <sec>1</sec> <nanosec>0</nanosec> </wait_timeout> Default: DDS_DURATION_INFINITE	0 or 1

4.5 Meta-Tables

After the *Database Integration Service* daemon has started and successfully made a connection to a database, the user will still need to configure the daemon to publish table changes as Topics as well as subscribe to Topics for storing received data into a table. This configuration is done by inserting entries into two tables **RTIDDS_PUBLICATIONS** and **RTIDDS_SUBSCRIPTIONS**. These tables will be created by the *Database Integration Service* daemon if they do not already exist in the database.

The two tables are referred to as *meta-tables* since their data is not user data but information used by the daemon to create DataWriters and DataReaders, as well as corresponding user tables in the database. The

tables are just ordinary tables that users can create themselves before starting the *Database Integration Service* daemon if so desired. However, if the user chooses to do so, it is important that the tables be created with the exact tables schemas presented below, otherwise the daemon may not work correctly. If the daemon finds existing meta-tables upon startup, it will process every row in the tables as if they were newly inserted. The meta-tables RTIDDS_PUBLICATIONS and RTIDDS_SUBSCRIPTIONS can be populated by using the <publication> and <subscription> tags in the configuration file (see 4.4.4.4 Initial Subscriptions and Publications on page 38) or by running Insert/Update SQL statements.

There are two more meta-tables created by the *Database Integration Service* daemon:

- The meta-table **RTIRTC_TBL_INFO** will store the typecode associated with the user tables created automatically by the *Database Integration Service* daemon (see 4.5.3 Table Info on page 78).
- The meta-table **RTIRTC_LOG** will be created to store log messages generated by the *Database Integration Service* daemon. Use of this table is controlled by command-line parameters and the connection discussed in 4.2 Command-Line Parameters on page 22 and 4.4.4.3 Database Connection Options on page 35.

The following sections discuss the usage of these tables and describe the actions taken by the daemon when these tables are modified:

4.5.1 Publications Table

When entries (rows) are added to the meta-table **RTIDDS_PUBLICATIONS**, the *Database Integration Service* daemon will try to create a DataWriter (and Publisher along with a DomainParticipant if required) and use it to send changes to the designated user table via the *Connext DDS*.

If the **RTIDDS_PUBLICATIONS** table does not exist at startup, the *Database Integration Service* daemon will create it with the table owner set to the user name of the database connection as specified in the daemon's configuration file, see 4.4 Configuration File on page 25. The schema and meaning of the columns of this table are described in the next section.

Users may insert new rows or modify the column values of existing rows in this table at anytime. For a new row, the daemon will first check to see if the designated user table exists. If so, it will immediately create the DataWriter with the QoS values as specified by the entry. The name of the Topic to publish may be specified by the **topic_name** column or be automatically constructed as **<table_owner>.<table_name>** if the **topic_name** entry is NULL.

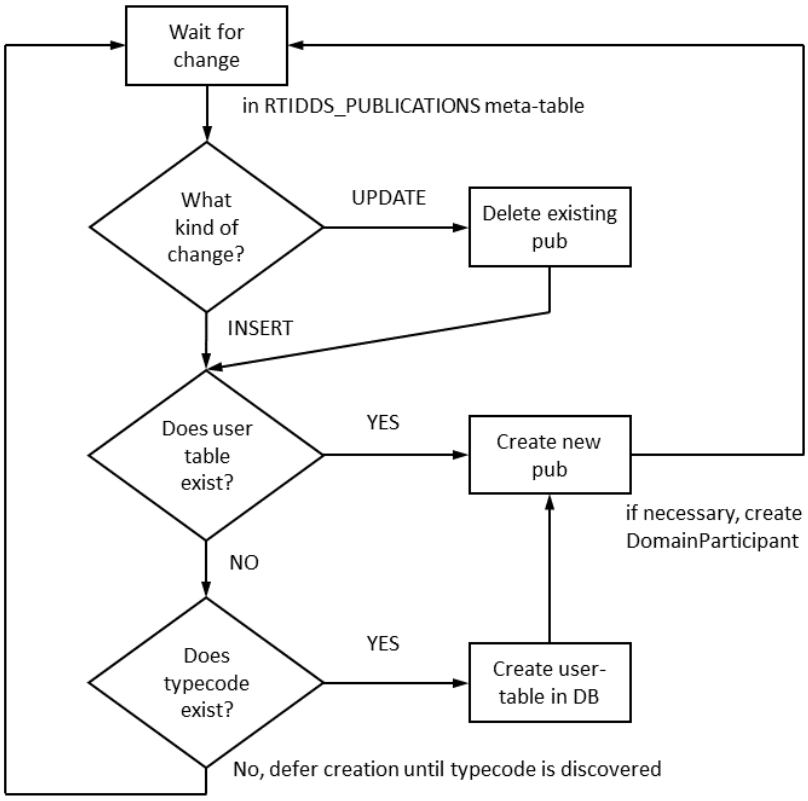
If the user table does not exist, the *Database Integration Service* daemon will look for the typecode associated with the topic defined in the **topic_name** column. If it finds the typecode, the daemon will create the user table with a SQL table schema derived from the typecode following the IDL type to SQL type mapping described in Chapter 5 IDL/SQL Semantic and Data Mapping on page 88. Then the daemon will proceed to create the associated DataWriter. More about the creation of user tables by the daemon can be found in 4.6 User-Table Creation on page 82.

How the daemon discovers and stores typecodes is described in 4.1.4 Typecode and TypeObject on page 21. If the *Database Integration Service* daemon has not yet have a typecode associated with the **topic_name**, it will defer the creation of the DataWriter until the typecode is discovered. When a new typecode is discovered, the daemon will scan all rows in the **RTIDDS_PUBLICATIONS** meta-table and create the user tables and DataWriters for entries that were pending on the discovery of the typecode.

The daemon will also create the DataWriter if there is an entry in the **RTIDDS_PUBLICATIONS** table without an associated typecode, but the user subsequently creates the corresponding table.

If user applications modify an existing row in the **RTIDDS_PUBLICATIONS** table, the *Database Integration Service* daemon will first delete the DataWriter that was created for that entry (if it exists) and then go through the same process of trying to create the user table and DataWriter as if the row was newly inserted. If user applications delete an existing row in the **RTIDDS_PUBLICATIONS** table, the *Database Integration Service* daemon will delete the associated DataWriter (if it exists).

A flow chart describing this logic is provided below.



4.5.1.1 Publications Table Schema

The **RTIDDS_PUBLICATIONS** table is created with the following SQL statement.

MySQL[1]:

```
Create Table RTIDDS_PUBLICATIONS (
    table_owner VARCHAR(128) NOT NULL,
    table_name VARCHAR(128) NOT NULL,
    domain_id INTEGER NOT NULL,
    topic_name VARCHAR(200),
    type_name VARCHAR(200),
    table_history_depth INTEGER,
    resolution_column VARCHAR(255),
    idl_member_prefix_max_length INTEGER,
    idl_member_suffix_max_length INTEGER,
    profile_name VARCHAR(255),
    "pub.present.access_scope" VARCHAR(25),
    "pub.present.ordered_access" TINYINT,
    "pub.partition.name" VARCHAR(256),
    "dw.durability.kind" VARCHAR(30),
```

[1] See 4.1.1.4 Starting the MySQL Server in ANSI_QUOTES mode on page 19.

```
    "dw.liveliness.lease_dur.sec" INTEGER,
    "dw.liveliness.lease_dur.nsec" INTEGER,
    "dw.deadline.period.sec" INTEGER,
    "dw.deadline.period.nsec" INTEGER,
    "dw.history.kind" VARCHAR(21),
    "dw.history.depth" INTEGER,
    "dw.ownership.kind" VARCHAR(23),
    "dw.ownership_strength.value" INTEGER,
    "dw.publish_mode.kind" VARCHAR(29),
    "dw.res_limits.max_samples" INTEGER,
    "dw.res_limits.max_instances" INTEGER,
    "metadata.timestamp_type" VARCHAR(20),
    "metadata.include_fields" VARCHAR(1000),
    "metadata.exclude_fields" VARCHAR(1000),
    changes_queue_maximum_size INTEGER,
    RTIRTC_SCN BIGINT DEFAULT 0,
    PRIMARY KEY(table_owner,table_name,domain_id,topic_name)
)
```

PostgreSQL:

Publication of database changes is not supported.

You should use the same SQL statement in your own applications if you want to create and populate this table before the *Database Integration Service* daemon is started. Table 4.20 RTIDDS_PUBLICATIONS Table Schema describes how each column is used by the daemon in creating and using DataWriters that publish table changes. Detailed descriptions of the columns follow the table.

Table 4.20 RTIDDS_PUBLICATIONS Table Schema

Column Name	SQL Type	Null-able	Default if NULL
table_owner[a]	VARCHAR(128)	No	N/A
table_name[b]	VARCHAR(128)	No	N/A
domain_id[c]	INTEGER	No	N/A
topic_name[d]	VARCHAR(200)	Yes	<table_owner>.<table_name>
type_name	VARCHAR(200)	Yes	<topic_name>
table_history_depth	INTEGER	Yes	0

[a]Primary key column

[b]Primary key column

[c]Primary key column

[d]Primary key column

Table 4.20 RTIDDS_PUBLICATIONS Table Schema

Column Name	SQL Type	Null-able	Default if NULL
resolution_column	VARCHAR(255)	Yes	None
idl_member_prefix_max_length	INTEGER	Yes	Value specified in the configuration file
idl_member_suffix_max_length	INTEGER	Yes	Value specified in the configuration file
profile_name	VARCHAR(255)	Yes	Database Integration Service will not use a profile to create the publication
pub.present.access_scope	VARCHAR(25)	Yes	INSTANCE_PRESENTATION_QOS
pub.present.ordered_access	TINYINT	Yes	0 (false)
pub.partition.name	VARCHAR(256)	Yes	Empty string partition
dw.durability.kind	VARCHAR(30)	Yes	VOLATILE_DURABILITY_QOS
dw.liveliness.lease_dur.sec	INTEGER	Yes	Infinite
dw.liveliness.lease_dur.nsec	INTEGER	Yes	Infinite
dw.deadline.period.sec	INTEGER	Yes	Infinite
dw.deadline.period.nsec	INTEGER	Yes	Infinite
dw.history.kind	VARCHAR(21)	Yes	KEEP_LAST_HISTORY_QOS
dw.history.depth	INTEGER	Yes	1
dw.ownership.kind	VARCHAR(23)	Yes	SHARED_OWNERSHIP_QOS
dw.ownership_strength.value	INTEGER	Yes	0
dw.publish_mode.kind	VARCHAR(29)	Yes	SYNCHRONOUS_PUBLISH_ MODE_QOS
dw.res_limits.max_samples	INTEGER	Yes	Infinite
dw.res_limits.max_instances	INTEGER	Yes	Infinite
metadata.timestamp_type	VARCHAR(20)	Yes	BIGINT
metadata.include_fields	VARCHAR(1000)	Yes	None
metadata.exclude_fields	VARCHAR(1000)	Yes	None
changes_queue_maximum_size	INTEGER	Yes	Infinite
RTIRTC_SCN	BIGINT	Yes	Next SCN number

4.5.1.1.1 table_owner, table_name

These columns specify the user table for which changes will be published using a DataWriter. Because a DBMS uses a combination of <table_owner>.<table_name> to identify a table, both of these columns

must have valid values should the user want these entries to refer to an existing table.

If no table exists in the database with the identifier "<table_owner>.<table_name>" at the time that the daemon sees this entry in the **RTIDDS_PUBLICATIONS** meta-table, it will create a user table with this name automatically, see 4.6 User-Table Creation on page 82.

Note: In MySQL, the value of the table_owner column corresponds to the table schema or database name.

4.5.1.1.2 domain_id

This column specifies the domain ID that will be used to publish changes in the table. Before creating a DataWriter, if no DomainParticipant has previously been created with the domain ID, the *Database Integration Service* daemon will create a DomainParticipant with the specified ID.

If the publications entry has associated a QoS profile, *Database Integration Service* will use the values in this profile to create the participant. The participant will also be configured using the QoS values of a profile when the attribute, **is_default_qos**, is set to 1 in that profile (see the *RTI Connext DDS Core Libraries User's Manual* for additional details).

4.5.1.1.3 topic_name

These column defines the Topic that will be used to publish the changes in the associated table. The **<topic_name>** need to match the Topic used by subscriptions in user applications that expect to received data changes from the table. If the *Database Integration Service* daemon has discovered the typecode associated with the **<topic_name>** and the user table does not exist in the database, the daemon will use the typecode to create the table using entries in the **<table_owner>** and **<table_name>** column. See 4.6 User-Table Creation on page 82 for more details.

4.5.1.1.4 type_name

This column defines the registered name of the type associated with the Topic defined using the column **<topic_name>**. If the user table does not exist in the database, the daemon will use the type name to find a typecode in the XML configuration file. See 4.6 User-Table Creation on page 82 for more details.

4.5.1.1.5 table_history_depth

The **<table_history_depth>** column in the **RTIDDS_PUBLICATIONS** determines whether or not the *Database Integration Service* daemon will create the user table with additional meta-columns that support the storing of historic, or past, values of instances of Topics by DataReaders created with the **RTIDDS_SUBSCRIPTIONS** table. It is only used if the daemon creates the table because it does not exist.

More about the ability to store historic data in the table as well as the added meta-columns can be found in 4.5.2.1.5 table_history_depth on page 65 and in 4.6 User-Table Creation on page 82.

This column is useful in the case that the user wants the *Database Integration Service* daemon to both publish and subscribe to a Topic for the same user table. The value set in **<table_history_depth>** will enable the daemon to create the user table correctly if the user wants to store more than a single value for an instance of the Topic in the table.

The possible values for **<table_history_depth>** column are:

- NULL or 0

  These values should be used if the user does not want to store more than a single value for an instance of a Topic in the table.

  If the *Database Integration Service* daemon creates the table, it will not add any meta-columns for table history to the table schema.
- Any other value

  For any non-zero value in this column, the *Database Integration Service* daemon will add meta-columns for table history to the table schema when it creates the user table automatically.

  A table's schema or definition cannot be changed to accommodate the table-history meta-columns after a table has been created. So a non-zero value for this column is useful if the user wants the table to be created with the ability to store historic values in support of entries in the **RTIDDS_SUBSCRIPTIONS** table that may be made later.

4.5.1.1.6 resolution_column

This column is used to designate one of the columns of the user table for use as the timestamp when data changes are published with the DataWriter. Instead of using the system time, when a row in the user table changes, the *Database Integration Service* daemon will take the current value of the designated column and use it in the **DataWriter::write_w_timestamp()** method when publishing the value of the row.

The possible values for the **<resolution_column>** column are:

- NULL

  If this column is NULL, then the *Database Integration Service* daemon will just call **DDSDataWriter::write()** to publish the table changes. This implies that the source timestamp used by *Connext DDS* will be the system time when the write occurred.
- "column_name"

  The column name of any column in the user table that has a valid type. The column must be one of the following SQL types: INTEGER, SMALLINT, BIGINT, or TIMESTAMP.

  If the user directs the daemon to use a column from the user table as the timestamp, then it is *imperative* to the proper operation of the publication that the value in the timestamp column is *monotonically increasing* with every table change. So when a change is made to a row of the table, the value in the column <resolution_column> must be larger than the last value of this column that was published.

The conversion to seconds and nanoseconds needed for DDS Time depends on the column type. An INTEGER or SMALLINT value will be used directly as the seconds, with the nanoseconds set to 0. A BIGINT value will extract the time as:

```
sec = (bigintTimestamp >> 32);
nanosec = (bigintTimestamp & 0x00000000FFFFFFFF);
```

A TIMESTAMP value is adjusted to the POSIX epoch of 1970.

The <resolution_column> can be used with the **<dr.destination_order.kind>** column of the **RTIDDS_SUBSCRIPTIONS** table to implement a conflict resolution policy in a system where *Database Integration Service* is used to implement database table replication across a network. See 4.5.2.1.18 dr.destination_order.kind on page 73 and 4.4.4.1.1 Enabling Table Replication on page 31 for more information.

4.5.1.1.7 idl_member_prefix_max_length, idl_member_suffix_max_length

These columns define how *Database Integration Service* maps IDL member identifiers into column names. In particular, they control how the column names are formed by using as a prefix *n* characters from the identifier's prefix and *m* characters from the identifier's suffix.

They can assume any value greater than or equal to -1. They cannot both be set to zero.

If a positive value *n* is provided for **idl_member_prefix_max_length**, *Database Integration Service* will use the first *n* characters from the IDL member identifier to compose the associated column name. A value of 0 tells *Database Integration Service* to compose the column name using only the last characters of the identifiers, as defined by the '**idl_member_suffix_max_length**' column. A value of -1, instructs *Database Integration Service* to use all the available characters.

If a positive value *n* is provided for **idl_member_suffix_max_length**, *Database Integration Service* will use the last *n* characters from the IDL member identifier to compose the associated column name. A value of 0 tells *Database Integration Service* to compose the column name using only the first characters of the identifiers, as defined by the '**idl_member_prefix_max_length**' column. A value of -1, instructs *Database Integration Service* to use all the available characters.

4.5.1.1.8 profile_name

This column specifies the name of the QoS Profile that *Database Integration Service* will use to create the publication.

The name must have the following format: **<QoS profile library name>::<QoS profile name>**

See the *Connext DDS* documentation for a complete description of QoS Profiles.

The QoS values specified in the publication table (if they are not NULL) take precedence over the same values specified in the QoS profile.

4.5.1.1.9 pub.present.access_scope, pub.present.ordered_access

These two columns map directly to the DDS_PresentationQosPolicy of the DDS_PublisherQos used by the Publisher that is created with the DataWriter for publishing changes to the table. The DDS_PresentationQosPolicy specifies how the samples representing changes to data instances are presented to a subscribing application.

The specific columns affect the relative order of changes seen by subscribers to the table. The values of these columns must be coordinated with the values of the DDS_PresentationQosPolicy used by the Subscriber in the receiving application or else published data may not be received by the subscriber.

The possible values for the **<pub.present.access_scope>** column are:

- "INSTANCE_PRESENTATION_QOS" (default value if the column is NULL)
- "TOPIC_PRESENTATION_QOS"
- "GROUP_PRESENTATION_QOS"

The possible values for the <pub.present.ordered_access> column are:

- 0 (default value if the column is NULL)
- 1

For the best performance of the *Database Integration Service* daemon, you should set <pub-.present.access_scope> to "TOPIC_PRESENTATION_QOS" and <pub.present.ordered_access> to 1. This will require that the corresponding values in the DDS_PresentationQosPolicy of the Subscriber in the receiving applications to be changed to those values as well.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.2.1.14 sub.present.access_scope, sub.present.ordered_access on page 71.

4.5.1.1.10 pub.partition.name

For publishing table changes, *Database Integration Service* creates a DataWriter per table. The **pub-.partition.name** column maps directly to the DDS_PartitionQosPolicy of the DDS_PublisherQos used by the Publisher that is created with the DataWriter. The DDS_PartitionQosPolicy introduces a logical partition concept inside the 'physical' partition concept introduced by the domain ID. A Publisher can communicate with a Subscriber only if they have some partition in common. The value of the **pub.partition.name** column specifies a list of partitions separated by commas to which the Publisher belongs.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.2.1.15 sub.partition.name on page 71

4.5.1.1.11 dw.durability.kind

This column maps directly to the DDS_DurabilityQosPolicy of the DataWriter created to publish table changes. By changing this policy, the *Database Integration Service* daemon can be configured to resend past changes to the database table to remote applications as soon as their subscriptions are discovered.

Only changes made by local applications to the table will be sent. That is, if the daemon is configured to subscribe to and store changes into the table from remote DataWriters, those changes are not sent. In addition, the changes will only be sent if the DataReader is created with a reliable setting for its DDS_ReliabilityQosPolicy.

The number of past changes that will be sent is limited by the values of the **<dw.history.kind>**, **<dw.history.depth>** and **<dw.res_limits.max_samples>** columns.

The possible values for the **<dw.durability.kind>** column are:

- "VOLATILE_DURABILITY_QOS"

  This value prevents the daemon from sending past changes to the table to newly discovered DataReaders.

  This also the default value if the column is NULL.
- "TRANSIENT_LOCAL_DURABILITY_QOS"

  This value will enable the daemon to send past changes to the table to newly discovered DataReaders. The DataReaders must be created with reliable DDS_ReliabilityQosPolicy.

Note: If a table exists when the *Database Integration Service* daemon creates a DataWriter, the daemon will initialize the DataWriter with the current contents of the table such that those values will be sent to new DataReaders with their DDS_DurabilityQosPolicy set to "TRANSIENT_LOCAL_DURABILITY_QOS."

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.2.1.16 dr.durability.kind on page 72 and 4.5.2.1.17 dr.reliability.kind on page 72.

4.5.1.1.12 dw.liveliness.lease_dur

These columns specify the lease duration for the DDS_LivelinessQosPolicy for the DataWriter created to publish table changes. The user may need to change the lease duration if remote applications have modified their DataReaders' corresponding DDS_LivelinessQosPolicy to non-default values.

The possible values of the **<dw.liveliness.lease_dur.sec>** (seconds) and **<dw.liveliness.lease_dur.nsec>** (nanoseconds) columns are:

- An infinite lease duration is specified if both columns are NULL or contain the value 2147483647 (2^{31} - 1). This is the DDS default value.

- A non-zero value representing the number of seconds and nanoseconds for the lease duration.

Note: DDS_LivelinessQosPolicy.kind is always set to DDS_AUTOMATIC_LIVELINESS_QOS.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.2.1.19 dr.liveliness.lease_dur on page 74.

4.5.1.1.13 dw.deadline.period

These columns specify the deadline period for the DDS_DeadlineQosPolicy for the DataWriter created to publish table changes. The user may need to change the deadline period if remote applications have modified their DataReaders' corresponding DDS_DeadlineQosPolicy to non-default values.

The possible values of the **<dw.deadline.period.sec>** (seconds) and **<dw.deadline.period.nsec>** (nanoseconds) columns are:

- An infinite deadline period is specified if both columns are NULL or contain the value 2147483647 (2^{31} - 1). This is the DDS default value.
- A non-zero value representing the number of seconds and nanoseconds for the deadline period.

The DDS_DeadlineQosPolicy sets a commitment by the DataWriter to publish a value for every data instance to DataReaders every deadline period. If this value is set to a non-infinite value, user applications must update the value of every instance of the Topic stored in the table within each deadline period or the contract with DataReaders that subscribe to the changes to the table will be violated.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.2.1.20 dr.deadline.period on page 74.

4.5.1.1.14 dw.history.kind, dw.history.depth

These columns directly map to the DDS_HistoryQosPolicy for the DataWriter created to publish table changes. The values set for this QoS policy affect the DDS_ReliabilityQosPolicy and the DDS_DurabilityQosPolicy.

Using a "KEEP_ALL_HISTORY_QOS" will ensure that reliable DataReaders will receive every change to the table reliably. With a "KEEP_LAST_HISTORY_QOS," the *Database Integration Service* daemon will only guarantee that the last **<dw.history.depth>** changes for each data instance are sent reliably.

If the **<dw.durability.kind>** column of the row is set to "TRANSIENT_LOCAL_DURA-BILITY_QOS", then these columns determine how many past data changes are sent to new subscribers to table changes.

The possible values of the **<dw.history.kind>** and **<dw.history.depth>** columns are:

- "KEEP_LAST_HISTORY_QOS"

  For this setting, the column **<dw.history.depth>** determines how many published changes for each data instance in the table are stored in the DataWriter to support reliability or durability.

  <dw.history.depth> should be set to an integer greater than 0. The default value for history depth is 1 if this column is NULL.
- "KEEP_ALL_HISTORY_QOS" (default value if the column is NULL)

  This setting implies that the DataWriter created to publish table changes will store all of the changes to the table that it has sent. The total number of changes that can be stored is limited by the value in the
  <dw.res_limits.-max_samples> column.

  For this setting, the value in <dw.history.depth> is ignored.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.11 dw.durability.kind on page 55 and 4.5.1.1.17 dw.res_limits.max_samples, dw.res_limits.max_ instances on the next page.

4.5.1.1.15 dw.ownership.kind, dw.ownership_strength.value

These columns directly map to the DDS_OwnershipQosPolicy and DDS_Ownership-StrengthQosPolicy for the DataWriter created to publish table changes. These policies control whether or not DataReaders are allowed to receive changes to an instance of a Topic from multiple DataWriters simultaneously.

The possible values of the <dw.ownership.kind> and <dw.ownership_strength.value> columns are:

- "SHARED_OWNERSHIP_QOS" (default value if the column is NULL)

  This setting allows DataReaders to receive updates for an instance of a Topic from multiple DataWriters at the same time.
- "EXCLUSIVE_OWNERSHIP_QOS"

  This setting prevents a DataReader from receiving changes from more than a single DataWriter for an instance of a Topic at the same time.

  The DataReader will receive changes for a topic instance from the DataWriter with the greatest value of ownership strength. If the liveliness of the DataWriter fails or if the DataWriter fails to write within a deadline period, then the DataReader will receive published changes to the topic instance from the DataWriter with the next highest ownership strength.

  The ownership strength set is set in the **<dw.ownerhip_strength.value>** column. The default value is 0 if the column is NULL.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.12 dw.liveliness.lease_dur on page 55, 4.5.1.1.13 dw.deadline.period on page 56, and 4.5.2.1.22 dr.ownership.kind on page 75.

4.5.1.1.16 dw.publish_mode.kind

This column controls the type of DataWriter that *Database Integration Service* will create for publishing data. This column can take the following values:

- ASYNCHRONOUS_PUBLISH_MODE_QOS
- SYNCHRONOUS_PUBLISH_MODE_QOS (default value)

Asynchronous DataWriters were introduced in *Connext DDS* to support large data packets (greater than 64Kb). If IDL data types exceed the 64Kb limit and reliable communication is used, **dw.publish_mode.kind** must be set to 'ASYNCHRONOUS_PUBLISH_MODE_QOS'.

See the *Connext DDS* documentation for more details on the differences between synchronous and asynchronous DataWriters.

4.5.1.1.17 dw.res_limits.max_samples, dw.res_limits.max_instances

These columns set some parameters for the DDS_ResourceLimitsQosPolicy for the DataWriter created to publish table changes. In particular, they control the amount of memory that the system is allowed to allocate for storing published data values as well as the total number of data instances (different primary keys) that can be handled by the DataWriter.

A value of -1 for either of these columns means infinite. An infinite setting means that the DataWriter is allowed to allocate memory as needed to store published table changes and manage new keys.

- The default value for dw.res_limits.max_samples (if set to NULL) is 32.
- The default value for dw.res_limits.max_instances (if set to NULL) is -1.

The number of keys that the DataWriter is allowed to manage places an upper limit on the number of rows that the related table in the database can have.

See the *Connext DDS* documentation for more details on how this QoS policy may be used.

4.5.1.1.18 changes_queue_maximum_size

This column is available only for connections to a MySQL database. The value of the column configures the maximum size of the queue that maintains the list of uncommitted changes. Note that there is a separate queue per table.

A value of -1 is used to indicate unlimited size.

4.5.1.1.19 RTIRTC_SCN

The System Change Number (SCN) column is available only for connections to a MySQL database. The value of this column is automatically maintained by *Database Integration Service* and is usually of no interest to the application. For more information about the RTIRTC_SCN column see 4.6 User-Table Creation on page 82.

4.5.2 Subscriptions Table

When entries (rows) are added to the meta-table **RTIDDS_SUBSCRIPTIONS**, the *Database Integration Service* daemon will try to create a DataReader (and Subscriber along with a DomainParticipant if required) and use it to receive data via the *Connext DDS* for a Topic and store values into the designated user table.

If the **RTIDDS_SUBSCRIPTIONS** table does not exist at startup, the *Database Integration Service* daemon will create it with the table owner set to the user name of the database connection as specified in the daemon's configuration file, see 4.4 Configuration File on page 25. The schema and meaning of the columns of this table are described in the next section.

You may insert new rows or modify the column values of existing rows in this table at any time. For a new row, the daemon will first check to see if the designated user table exists. If so, it will immediately create the DataReader with the QoS values specified by the entry. The name of the Topic to subscribe to may be specified by the **topic_name** column or automatically constructed as **<table_owner>.<table_name>** if the **topic_name** entry is NULL.

If the user table does not exist, the *Database Integration Service* daemon will look for the typecode associated with the type defined in the **topic_name** column. If it finds the typecode, the daemon will create the user table.

The table schema of the user table is determined by the value of the column **table_schema** (see 4.5.3 Table Info on page 78). These supported schemas are:

- FLATTEN: The SQL table schema is derived from the typecode following the IDL type to SQL type mapping described in 5.2 Flatten Data Representation Mapping on page 90.
- [Only MySQL and PostgreSQL] JSON: The DDS sample content will be stored in a column with JSON type following the format described in 5.3 JSON Data Representation Mapping on page 99.
- [Only PostgreSQL] JSONB: The DDS sample content will be stored in a column with JSONB type (binary JSON representation).

After the user table is created, the daemon will proceed to create the associated DataReader. More about the creation of user tables by the daemon can be found in 4.6 User-Table Creation on page 82.

How the daemon discovers and stores typecodes is described in 4.1.4 Typecode and TypeObject on page 21.

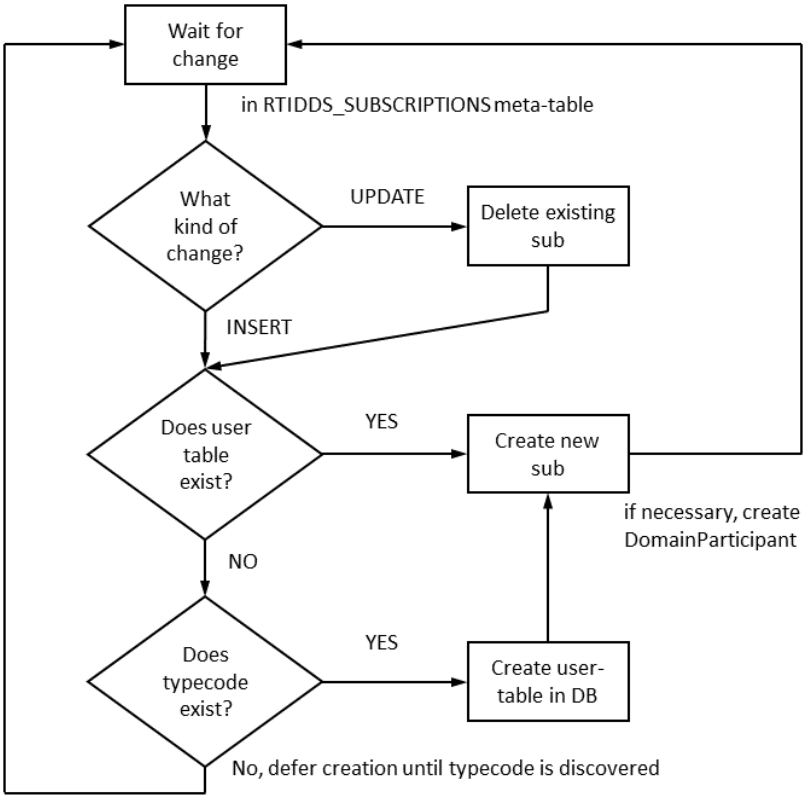
If the *Database Integration Service* daemon does not yet have a typecode associated with the **topic_name**, it will defer the creation of the DataReader until the typecode is discovered. When a new typecode is discovered, the daemon will scan all rows in the **RTIDDS_SUBSCRIPTIONS** meta-table and create the user tables and DataReaders for entries that were pending on the discovery of the typecode.

The daemon will also create the DataReader if there is an entry in the **RTIDDS_SUBSCRIPTIONS** table without an associated typecode, but the user subsequently creates the corresponding table.

If user applications modify an existing row in the **RTIDDS_SUBSCRIPTIONS** table, the *Database Integration Service* daemon will first delete the DataReader that was created for that entry (if it exists) and then go through the same process of trying to create the user table and DataReader as if the row was newly inserted.

If user applications delete an existing row in the **RTIDDS_SUBSCRIPTIONS** table, the *Database Integration Service* daemon will delete the associated DataReader (if it exists).

A flow chart describing this logic is provided below.



4.5.2.1 Subscriptions Table Schema

The **RTIDDS_SUBSCRIPTIONS** table is created with the following SQL statement.

MySQL (see 4.1.1.4 Starting the MySQL Server in ANSI_QUOTES mode on page 19):

```
Create Table RTIDDS_SUBSCRIPTIONS (
    table_owner VARCHAR(128) NOT NULL,
    table_name VARCHAR(128) NOT NULL,
    domain_id INTEGER NOT NULL,
    topic_name VARCHAR(200),
    type_name VARCHAR(200),
    table_history_depth INTEGER,
    process_batch INTEGER,
    "process_period.sec" INTEGER,
    "process_period.nsec" INTEGER,
    commit_type VARCHAR(17),
    cache_maximum_size INTEGER,
    cache_initial_size INTEGER,
    delete_on_dispose INTEGER,
    idl_member_prefix_max_length INTEGER,
    idl_member_suffix_max_length INTEGER,
    profile_name VARCHAR(255),
    filter_duplicates TINYINT,
    ordered_store TINYINT,
    persist_state TINYINT,
    "sub.present.access_scope" VARCHAR(25),
    "sub.present.ordered_access" TINYINT,
    "sub.partition.name" VARCHAR(256),
    "dr.durability.kind" VARCHAR(30),
    "dr.reliability.kind" VARCHAR(27),
    "dr.destination_order.kind" VARCHAR(43),
    "dr.liveliness.lease_dur.sec" INTEGER,
    "dr.liveliness.lease_dur.nsec" INTEGER,
    "dr.deadline.period.sec" INTEGER,
    "dr.deadline.period.nsec" INTEGER,
    "dr.history.kind" VARCHAR(21),
    "dr.history.depth" INTEGER,
    "dr.ownership.kind" VARCHAR(23),
    "dr.time_filter.min_sep.sec" INTEGER,
    "dr.time_filter.min_sep.nsec" INTEGER,
    "dr.res_limits.max_samples" INTEGER,
    "dr.res_limits.max_instances" INTEGER,
    "dr.unicast.receive_port" INTEGER,
    "dr.multicast.receive_address" VARCHAR(39),
    "dr.multicast.receive_port" INTEGER,
    "metadata.timestamp_type" VARCHAR(20),
    "metadata.include_fields" VARCHAR(1000),
    "metadata.exclude_fields" VARCHAR(1000),
    table_schema VARCHAR(7),
    PRIMARY KEY(table_owner,table_name,domain_id,topic_name)
    RTIRTC_SCN BIGINT DEFAULT 0,
)
```

PostgreSQL:

```
Create Table RTIDDS_SUBSCRIPTIONS (
    table_owner VARCHAR(128) NOT NULL,
    table_name VARCHAR(128) NOT NULL,
    domain_id INTEGER NOT NULL,
    topic_name VARCHAR(200),
    type_name VARCHAR(200),
    table_history_depth INTEGER,
    process_batch INTEGER,
    "process_period.sec" INTEGER,
    "process_period.nsec" INTEGER,
    commit_type VARCHAR(17),
    cache_maximum_size INTEGER,
    cache_initial_size INTEGER,
    delete_on_dispose INTEGER,
    idl_member_prefix_max_length INTEGER,
    idl_member_suffix_max_length INTEGER,
    profile_name VARCHAR(255),
    filter_duplicates BOOLEAN,
    ordered_store BOOLEAN,
    persist_state BOOLEAN,
    "sub.present.access_scope" VARCHAR(25),
    "sub.present.ordered_access" BOOLEAN,
    "sub.partition.name" VARCHAR(256),
    "dr.durability.kind" VARCHAR(30),
    "dr.reliability.kind" VARCHAR(27),
    "dr.destination_order.kind" VARCHAR(43),
    "dr.liveliness.lease_dur.sec" INTEGER,
    "dr.liveliness.lease_dur.nsec" INTEGER,
    "dr.deadline.period.sec" INTEGER,
    "dr.deadline.period.nsec" INTEGER,
    "dr.history.kind" VARCHAR(21),
    "dr.history.depth" INTEGER,
    "dr.ownership.kind" VARCHAR(23),
    "dr.time_filter.min_sep.sec" INTEGER,
    "dr.time_filter.min_sep.nsec" INTEGER,
    "dr.res_limits.max_samples" INTEGER,
    "dr.res_limits.max_instances" INTEGER,
    "dr.unicast.receive_port" INTEGER,
    "dr.multicast.receive_address" VARCHAR(39),
    "dr.multicast.receive_port" INTEGER,
    "metadata.timestamp_type" VARCHAR(20),
    "metadata.include_fields" VARCHAR(1000),
    "metadata.exclude_fields" VARCHAR(1000),
    table_schema VARCHAR(7),
    PRIMARY KEY(table_owner,table_name,domain_id,topic_name)
)
```

You should use the same SQL statement in your own applications if you want to create and populate this table before the *Database Integration Service* daemon is started. Table 4.21 RTIDDS_SUBSCRIPTIONS

Table Schema describes how each column is used by the daemon in creating DataReaders and storing received data into tables. Detailed descriptions of the columns follow the table.

Table 4.21 RTIDDS_SUBSCRIPTIONS Table Schema

Column Name	SQL Type	Null-able	Default if NULL
table_owner[a]	VARCHAR(128)	No	N/A
table_name[b]	VARCHAR(128)	No	N/A
domain_id[c]	INTEGER	No	N/A
topic_name[d]	VARCHAR(200)	YES	<table_owner>.<table_name>
type_name	VARCHAR(200)	YES	<topic_name>
table_history_depth	INTEGER	YES	0
process_batch	INTEGER	YES	10
process_period.sec	INTEGER	YES	0
process_period.nsec	INTEGER	YES	100000000
commit_type	VARCHAR(17)	YES	COMMIT_ON_PROCESS
cache_maximum_size	INTEGER	YES	0
cache_initial_size	INTEGER	YES	0
delete_on_dipose	INTEGER	YES	0
idl_member_prefix_max_length	INTEGER	YES	Value specified in the configuration file
idl_member_suffix_max_length	INTEGER	YES	Value specified in the configuration file
profile_name	VARCHAR(255)	YES	Database Integration Service will not use a profile to create the publication
filter_duplicates	TINYINT	YES	0
ordered_store	TINYINT	YES	1
persist_state	TINYINT	YES	0
sub.present.access_scope	VARCHAR(25)	YES	INSTANCE_PRESENTATION_QOS
sub.present.ordered_access	TINYINT	YES	0 (false)

[a]Primary key column.

[b]Primary key column.

[c]Primary key column.

[d]Primary key column.

Table 4.21 RTIDDS_SUBSCRIPTIONS Table Schema

Column Name	SQL Type	Null-able	Default if NULL
sub.partition.name	VARCHAR(256)	YES	Empty partition string
dr.durability.kind	VARCHAR(30)	YES	VOLATILE_DURABILITY_QOS
dr.reliability.kind	VARCHAR(27)	YES	BEST_EFFORT_RELIABILITY_QOS
dr.destination_order.kind	VARCHAR(43)	YES	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
dr.liveliness.lease_dur.sec	INTEGER	YES	Infinite
dr.liveliness.lease_dur.nsec	INTEGER	YES	Infinite
dr.deadline.period.sec	INTEGER	YES	Infinite
dr.deadline.period.nsec	INTEGER	YES	Infinite
dr.history.kind	VARCHAR(21)	YES	KEEP_LAST_HISTORY_QOS
dr.history.depth	INTEGER	YES	1
dr.ownership.kind	VARCHAR(23)	YES	SHARED_OWNERSHIP_QOS
dr.time_filter.min_sep.sec	INTEGER	YES	0
dr.time_filter.min_sep.nsec	INTEGER	YES	0
dr.res_limits.max_samples	INTEGER	YES	Infinite
dr.res_limits.max_instances	INTEGER	YES	Infinite
dr.unicast.receive_port	INTEGER	YES	0
dr.multicast_receive_address	VARCHAR(15)	YES	None
dr.multicast.receive_port	INTEGER	YES	0
metadata.timestamp_type	VARCHAR(20)	YES	BIGINT
metadata.include_fields	VARCHAR(1000)	YES	None
metadata.exclude_fields	VARCHAR(1000)	YES	None
table_schema	VARCHAR(7)	YES	FLATTEN
RTIRTC_SCN	BIGINT	YES	Next SCN number

4.5.2.1.1 table_owner, table_name

These columns specify the user table into which data received by a DataReader will be stored. Because a DBMS uses a combination of **<table_owner>.<table_name>** to identify a table, both of these columns must have valid values should the user want these entries to refer to an existing table.

If no table exists in the database with the identifier **<table_owner>.<table_name>** at the time that the daemon sees this entry in the **RTIDDS_SUBSCRIPTIONS** meta-table, it will create a user table with this name automatically, see 4.6 User-Table Creation on page 82.

Notes:

- In MySQL, the value of the **table_owner** column corresponds to the table schema or database name.
- In PostgreSQL, the value of the **table_owner** column corresponds to **<database name>[.<schema_name>]**. If the user does not provide a schema, *Database Integration Service* will use "public".

4.5.2.1.2 domain_id

This column specifies the domain ID that will be used to subscribe to Topics whose values will be stored in the table. Before creating a DataReader, if no DomainParticipant has previously been created with the domain ID, the *Database Integration Service* daemon will create a DomainParticipant with the specified ID.

If the subscriptions entry has an associated QoS profile, *Database Integration Service* will use the values in this profile to create the participant. The participant will also be configured using the QoS values of a profile when the attribute, **is_default_qos**, is set to 1 in that profile (see the *RTI Connext DDS Core Libraries User's Manual* for additional details).

4.5.2.1.3 topic_name

This column defines the Topic that will be subscribed to and whose received values will be stored in the associated table. The **<topic_name>** entry needs to match the Topic used by the DataWriters that are sending data changes.

If the *Database Integration Service* daemon has discovered the typecode associated with the **<topic_name>** and the user table does not exist in the database, the daemon will use the typecode to create the table using entries in the **<table_owner>** and **<table_name>** column. See 4.6 User-Table Creation on page 82 for more details.

4.5.2.1.4 type_name

This column defines the registered name of the type associated with the Topic defined using the column **<topic_name>**. If the user table does not exist in the database, the daemon will use the type name to find a typecode in the XML configuration file. See 4.6 User-Table Creation on page 82 for more details.

4.5.2.1.5 table_history_depth

This column determines the number of values of each instance received by the DataReader that can be stored in the table by the *Database Integration Service* daemon. For non-keyed Topics, there is only a

single instance, thus the **<table_history_depth>** would correspond to the maximum size of the table (in rows).

For keyed Topics, the *Database Integration Service* daemon may store up to **<table_history_depth>** values of each instance of the Topic that the DataReader receives. When the history depth is reached, the rows are reused as a circular buffer with the newest values replacing the oldest.

To support this capability, the associated user table may be created with additional columns, *meta-columns*, to help the *Database Integration Service* daemon manage history for a table. Whether or not meta-columns need to be added to support table history is based on the value of the entry in **<table_history_depth>**.

The two meta-columns for supporting table history are:

- **RTIRTC_HISTORY_SLOT**: INTEGER

  This column is also added to the Primary Key of the table. There is usually no need for users to access this column, it is only used by the daemon. It is only needed since many DBMS systems do not allow you to alter the value of a Primary Key column.
- **RTIRTC_HISTORY_ORDER**: INTEGER

  This value of this column is maintained by the *Database Integration Service* daemon when it stores data received via *Connext DDS* into the table. The column stores a strictly incrementing counter that represents the received sequence number (starting at 0) of the data that is stored in that row.

  You should use a combination of the instance key and the value of **RTIRTC_HISTORY_ORDER** to find the latest data received for an instance in the table.

The possible values for the **<table_history_depth>** column are:

- NULL or 0

  Only the current value of an instance of the Topic is stored. For non-keyed topics, this implies the table will only have a single row. For keyed topics, each instance will correspond to a single row in the table. This is the most common value for tables that are published with *Connext DDS*.

  No meta-columns are added to help manage history.
- 1

  Exactly the same behavior as NULL or 0, a single value is stored in the table per instance of the Topic. However, table-history meta-columns *are* added to the table schema if the *Database Integration Service* daemon creates the user table automatically.

  This value is useful for preparing the table to store more than a single value per instance after the table is created. Because table schema cannot be changed to accommodate the table-history meta-

columns after a table has been created, using a value of 1 for this column is useful if the user wants to store historic values of instances, but does not know how many instances to store at the time the entry is made.

- n > 1

  Meta-columns will be added to accommodate the storing of historic values for instances. The last n values received for an instance will be stored by the table.

- -1

  Meta-columns will be added to accommodate the storing of historic values for instances. *All* values received by the DataReader will be stored by the table.

See 4.6 User-Table Creation on page 82 for more information on meta-columns.

4.5.2.1.6 process_batch, process_period, commit_type

These columns allow users to tune the *Database Integration Service* daemon for optimal throughput performance. When the daemon receives data from a DataReader, it may be configured to delay storing the data into a table and/or committing the transaction until more data arrives. For a data streams with high throughput, thousands of samples per seconds, the ability for the daemon to process incoming data in batches greatly improves the efficiency and ultimately the maximum sustainable throughput rate for a given Topic.

The trade-off is latency. The more data that is processed in a single batch, the more efficiently the processing can occur. However, A greater delay between the receiving of the data by the daemon and the time that it can be accessed by user applications in the database.

The column **<process_batch>** controls how many data samples are processed at a time by the *Database Integration Service* daemon. Instead of executing SQL UPDATE or INSERT every time data is received, the daemon only stores the data after it receives a certain number of samples set by **<process_batch>**. If the value **<process_batch>** is greater than 1, then it is essential that the **<process_period.[sec,nsec]>** is set to be non-zero. Thus, the daemon will process stored data periodically, even if the total number of data samples received is less than **<process_batch>**.

<process_period.[sec,nsec]> is an upper limit on the amount of delay that will be incurred before received data is stored in the database. The period can be set to 0 only if **<process_batch>** is set to 1. This means that the daemon will store each data sample as it is received so there is no need for periodic processing of the received samples. Use these values to have the daemon store the data with minimal latency (at the cost of lower overall throughput).

Finally, using the column **<commit_type>**, you can choose whether or not the SQL UPDATE/INSERT statements are committed when each data sample is stored or after all of the data being processed have been stored. There is significant performance enhancement if the storing of multiple data samples is committed as a single transaction.

However, if there is a problem during an SQL commit, for example, the transaction log of the database is full, then the entire transaction is rolled back which means that none of the received data in that batch will be stored in the table. If the storing of each data sample is committed separately, then an error committing any one sample will only result in the loss of that sample.

The possible values of the **<process_batch>** column are:

- n > 0

  The daemon will process data samples in batches of n. A value less than or equal to 0 will result in an error that is logged by the daemon.

  A value of n = 1 means that the daemon will store each data sample as it arrives.

  The default value is 10 if this column is NULL.

The possible values of the **<process_period.sec>** (seconds) and **<process_period.nsec>** (nanoseconds) columns are:

- 0

  If both columns are 0, then the daemon will not commit received samples periodically.
- n > 0

  A background thread will process received but un-stored data at the period specified by these columns. It is essential that a non-zero period be used if **<process_batch>** is greater than 1 to insure that all received data is eventually stored.

  The default value for process period is 0.1 seconds (0 sec, 100000000 nanosec) if both columns are NULL.

The possible values of the **<commit_type>** column are:

- “COMMIT_ON_PROCESS” (default value if the columns are NULL)

  This value will direct the *Database Integration Service* daemon to commit the storage of a batch of data as a single transaction. This will result in higher performance at the risk of losing more data than necessary when the transaction is rolled-back because an error with the database.
- “COMMIT_ON_SAMPLE”

  This value will direct the daemon to commit the storage of each data sample as a separate transaction. Although the daemon will use more resources, if an error occurs when a transaction is committed, only that data sample is lost.

4.5.2.1.7 cache_maximum_size, cache_initial_size

These columns control the size of a cache, used to store keys that exist in the table, that the *Database Integration Service* daemon maintains for each DataReader. When a data instance is received, the daemon first checks the cache to see if a row corresponding to the data already exists in the table. If the key is in the cache, then the daemon executes an SQL UPDATE to store the data in the table.

If the key does not exist in the cache, then the *Database Integration Service* daemon will INSERT a row with the key instead. The key cache can greatly enhance the performance of the daemon in storing data into the database by saving an SQL operation each time data is received. Without a cache, the daemon would need to execute 2 SQL statements. to store data; with the cache, only 1.

The trade off is the memory used to store keys versus the performance gain.

The default values of **<cache_maximum_size>** and **<cache_initial_size>** are 0 if the columns are NULL. The sizes are specified as the number of keys.

For small tables, the cache could be sized to hold all of the keys. Thus the size of the cache would be the maximum number of rows in the table. However, this is not practical for large tables and thus the cache will be smaller.

4.5.2.1.8 delete_on_dispose

This column configures the behavior of the *Database Integration Service* daemon when a DataWriter disposes an instance stored into the database. When **delete_on_dispose** is initialized to 0 (the default value), the rows corresponding to the instance will not be deleted from the database. If **delete_on_dispose** is initialized to 1, all the rows associated with the instance will be deleted from the database.

4.5.2.1.9 idl_member_prefix_max_length, idl_member_suffix_max_length

These columns define how *Database Integration Service* maps IDL member identifiers into column names. In particular, they control how the column names are formed by using as a prefix *n* characters from the identifier's prefix and *m* characters from the identifier's suffix.

They can assume any value greater than or equal to -1. They cannot both be set to zero.

If a positive value *n* is provided for **idl_member_prefix_max_length**, *Database Integration Service* will use the first *n* characters from the IDL member identifier to compose the associated column name. A value of 0 tells *Database Integration Service* to compose the column name using only the last characters of the identifiers, as defined by the '**idl_member_suffix_max_length**' column. A value of -1, instructs *Database Integration Service* to use all the available characters.

If a positive value *n* is provided for **idl_member_suffix_max_length**, *Database Integration Service* will use the last *n* characters from the IDL member identifier to compose the associated column name. A value of 0 tells *Database Integration Service* to compose the column name using only the first characters of the identifiers, as defined by the '**idl_member_prefix_max_length**' column. A value of -1, instructs *Database Integration Service* to use all the available characters.

4.5.2.1.10 profile_name

This column specifies the name of the QoS Profile that *Database Integration Service* will use to create the subscription.

The name must have the following format:

<QoS profile library name>::<QoS profile name>

See *Configuring QoS with XML*, in the RTI Connext DDS Core Libraries User's Manual for a complete description of QoS Profiles.

QoS values specified in the subscription table (if they are not NULL) take precedence over the same values specified in the QoS profile.

4.5.2.1.11 filter_duplicates

There are multiple scenarios in which *Database Integration Service* may receive duplicate samples (see *Mechanisms for Achieving Information Durability and Persistence*, in the RTI Connext DDS Core Libraries User's Manual). For example, if *RTI Persistence Service* is used in the system, *Database Integration Service* could receive the same sample from the original writer and from *RTI Persistence Service*.

The **filter_duplicates** column specifies whether or not duplicates should be filtered by the *Database Integration Service* daemon. If duplicates are not filtered, the subscription data table may end up containing duplicates rows.

Note: Durable Reader State configurations are ignored by *Database Integration Service*. Duplicate filtering and subscription state persistence are implemented by the *Database Integration Service* daemon.

4.5.2.1.12 ordered_store

This column specifies whether or not the samples associated with a DataWriter identified by a virtual GUID 'x' (see Durability and Persistence Based on Virtual GUIDs, in the chapter on *Mechanisms for Achieving Information Durability and Persistence* in the RTI Connext DDS Core Libraries User's Manual) must be stored in the database in order. The field only applies when **filter_duplicates** (4.5.2.1.11 filter_duplicates above) is set to 1.

Ordered storage means that given a DataWriter with virtual GUID 'x', a sample with virtual sequence number 'sn+1' will be stored after a virtual sample with virtual sequence number 'sn'. If there is only one DataWriter with virtual GUID 'x' in the system (for example, if there are no *RTI Persistence Service* instances) the value of this column does not affect behavior.

Note: *Database Integration Service* stores samples in the database as soon as they are received by the *Database Integration Service* subscriptions (*Connext DDS* DataReaders). If **ordered_store** is set to 1 and there are multiple DataWriters with the same virtual GUID in the system, old samples will not be stored in the database. A sample with sequence number 'sn' will be ignored if a sample with sequence number 'sn+1' for the same virtual writer has been already stored in the database.

4.5.2.1.13 persist_state

This column specifies whether or not the state of a *Database Integration Service* subscription must be persisted into the database. The field only applies when **filter_duplicates** (4.5.2.1.11 filter_duplicates on the previous page) is set to 1. The subscription state is used on restore by *Database Integration Service* in order to avoid receiving duplicate samples.

4.5.2.1.14 sub.present.access_scope, sub.present.ordered_access

These two columns map directly to the DDS_PresentationQosPolicy of the DDS_SubscriberQos used by the Subscriber that is created with the DataReader for storing received data in the table. The DDS_PresentationQosPolicy specifies how the data instances sent by a publishing application are ordered before they are received.

The values of these columns must be coordinated with the values of the DDS_PresentationQosPolicy used by the Publisher in the sending application or else published data may not be received by the DataReader.

The possible values for the **<sub.present.access_scope>** column are:

- "INSTANCE_PRESENTATION_QOS"

  This is also the default value if the column is NULL.
- "TOPIC_PRESENTATION_QOS"
- "GROUP_PRESENTATION_QOS"

The possible values for the <sub.present.ordered_access> column are:

- 0 (default value if the column is NULL)
- 1

For the best performance of the *Database Integration Service* daemon, you should set **<sub-.present.access_scope>** to "TOPIC_PRESENTATION_QOS" and **<sub.present.ordered_access>** to 1. This will require that the corresponding values in the DDS_PresentationQosPolicy of the Publisher in the sending applications to be changed to those values as well.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.9 pub.present.access_scope, pub.present.ordered_access on page 54.

4.5.2.1.15 sub.partition.name

For capturing data in a table, *Database Integration Service* creates a DataReader per Topic. The *sub-.partition.name* column maps directly to the DDS_PartitionQosPolicy of the DDS_SubscriberQos used by the Subscriber that is created with the DataReader. The DDS_PartitionQosPolicy introduces a logical partition concept inside the 'physical' partition concept introduced by the domain ID. A Subscriber can communicate with a Publisher only if they have some partition in common. The value of the

sub.partition.name column specifies a list of partitions separated by commas to which the Subscriber belongs.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.10 pub.partition.name on page 54

4.5.2.1.16 dr.durability.kind

This column maps directly to the DDS_DurabilityQosPolicy of the DataReader created to subscribe to Topic data that is stored in the table. By changing this policy, the DataReader can be configured to request for past values published for the Topic to be sent by existing applications soon as their matching DataWriters are discovered.

The DataWriter's DDS_DurabilityQosPolicy most also be set appropriately to permit the sending of historic, or past, published data. In addition, the column <dr.reliability.kind> for the entry must be set to "RELIABLE_RELIABILITY_QOS" for historic data to be received.

The possible values for the **<dr.durability.kind>** column are:

- "VOLATILE_DURABILITY_QOS" (default value if the column is NULL)

  This value means that the DataReader does not request past data to be sent.
- "TRANSIENT_LOCAL_DURABILITY_QOS"

  This value requests that existing DataWriters of the Topic send past data that they are storing to the DataReader.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.11 dw.durability.kind on page 55 and 4.5.2.1.17 dr.reliability.kind below.

4.5.2.1.17 dr.reliability.kind

This column sets the DDSReliabilityQosPolicy for the DataReader created to subscribe to Topic data that is stored in the table. The value in this column determines whether or not DataWriters will send their data reliably to the DataReader.

If the value for **<dr.durability.kind>** is "TRANSIENT_LOCAL_DURABILITY_QOS", then the value for this column must be set to "RELIABLE_RELIABILITY_QOS".

The possible values for the **<dr.reliability.kind>** column are:

- "BEST_EFFORT_RELIABILITY_QOS" (default value if the column is NULL)

  This value means that the DataWriters will send their data to the DataReader using best efforts. Data may be lost if the system is too busy.
- "RELIABLE_RELIABILITY_QOS"

This value means that the DataWriters will send their data to the DataReader using a reliable protocol. The exact semantics of the reliable connection is controlled by the DDS_HistoryQosPolicy of both the DataWriter and DataReader.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.11 dw.durability.kind on page 55, 4.5.1.1.14 dw.history.kind, dw.history.depth on page 56, and 4.5.2.1.21 dr.history.kind, dr.history.depth on page 75.

4.5.2.1.18 dr.destination_order.kind

This column sets the DestinationOrderQosPolicy for the DataReader created to subscribe to Topic data that is stored in the table. The value in this column determines how the DataReader treats data received for the same instance of the Topic from different DataWriters.

When a data instance is received, a timestamp associated with the data is compared to the timestamp of the last value of the data instance. If the time of the new data is older than the time of the last data received (for that instance), then the new data is dropped.

What this column does is to set which timestamp (the one associated with the source of the data when it was sent or the one associated with the data when it was received) the DataReader will use.

This column has no practical effect unless the value of the **<dr.ownership.kind>** column is "SHARED_OWNERSHIP_QOS".

The possible values for the **<dr.destinaton_order.kind>** column are:

- "BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS"
  (default value if the column is NULL)

  This configures the DataReader to use the timestamp of when the data was received to determine whether or not to drop the data. In practice, this setting means all data received from all DataWriters will be accepted since the timestamp will always be newer for the new data.
- "BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS"

  This value means that the DataReader will use the timestamp that was sent with the data in determining whether or not to accept the data. This timestamp was added by the DataWriter when the data was published. Because different DataWriters may run in applications on different machines, it is likely that the clocks on the different machines are only synchronized to a certain resolution or not synchronized at all.

  Thus the DataReader may receive data with timestamps older than the last data that received and thus drop those data. However if all DataReaders of the same Topic used the source timestamp to filter the data, then all DataReaders will end up with the same final value for a data instance.

If DataReaders used the reception timestamp, the DataReaders may end up with different final values because data from different DataWriters may be received in a different order by different DataReaders.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.6 resolution_column on page 52.

4.5.2.1.19 dr.liveliness.lease_dur

These columns specify the lease duration for the DDS_LivelinessQosPolicy for the DataReader created to subscribe to Topic data that is stored in the table. This value is useful when there are redundant DataWriters that publish values for the same data instance for the Topic and the value set for the **<dr.ownership.kind>** column is "EXCLUSIVE_OWNERSHIP_QOS".

The liveliness of a DataWriter is monitored by the DataReader. These columns control how quickly the DataReader can determine that the DataWriter with the highest ownership strength has lost liveliness because heartbeat packets or data were not received within the liveliness lease duration. When liveliness is lost, the DataReader will then receive the data instance from the DataWriter with the next highest ownership strength that is still alive.

The possible values of the **<dr.liveliness.lease_dur.sec>** (seconds) and **<dr.liveliness.lease_dur.nsec>** (nanoseconds) columns are:

- An infinite lease duration is specified if both columns are NULL or contain the value 2147483647 (2^{31} - 1). This is the DDS default value.
- A non-zero value representing the number of seconds and nanoseconds for the lease duration.

Note: The DDS_LivelinessQosPolicy.kind is always set to DDS_AUTOMATIC_LIVELINESS_QOS.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.12 dw.liveliness.lease_dur on page 55, 4.5.2.1.22 dr.ownership.kind on the next page, and 4.5.1.1.15 dw.ownership.kind, dw.ownership_strength.value on page 57.

4.5.2.1.20 dr.deadline.period

These columns specify the deadline period for the DDS_DeadlineQosPolicy for the DataReader created to subscribe to Topic data that is stored in the table. By setting the values in this column, the user is setting an expectation that DataWriters will publish new values for data instances at least as fast as the deadline period.

The possible values of the **<dr.deadline.period.sec>** (seconds) and **<dr.deadline.period.nsec>** (nanoseconds) columns are:

- An infinite deadline period is specified if both columns are NULL or contain the value 2147483647 (2^{31} - 1). This is the DDS default value.
- A non-zero value representing the number of seconds and nanoseconds for the deadline period.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.1.1.13 dw.deadline.period on page 56.

4.5.2.1.21 dr.history.kind, dr.history.depth

These columns directly map to the DDS_HistoryQosPolicy for the DataReader created to subscribe to Topic data that is stored in the table. The values set for this QosPolicy affect the DDS_ReliabilityQosPolicy.

Using a "KEEP_ALL_HISTORY_QOS" will ensure that reliable DataReaders will receive every change to the table reliably. With a "KEEP_LAST_HISTORY_QOS", the *Database Integration Service* daemon will only guarantee that the last **<dr.history.depth>** changes for each data instance are received reliably.

The possible values of the **<dr.history.kind>** and **<dr.history.depth>** columns are:

- "KEEP_LAST_HISTORY_QOS"

  For this setting, the column **<dr.history.depth>** determines the maximum number of values for each data instance that be buffered in the DataReader before the *Database Integration Service* daemon stores the received values into the table.

  <dr.history.depth> should be set to an integer greater than 0. The default value for history depth is 1 if this column is NULL.
- "KEEP_ALL_HISTORY_QOS" (default value if the column is NULL)

  This setting implies that the DataReader created to subscribe to Topic data has an unlimited queue in which to save received data before the data is stored in the table. The actual size of the queue is limited by the value in **<dr.res_limits.max_samples>** column.

  For this setting, the value in **<dr.history.depth>** is ignored.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.2.1.24 dr.res_limits.max_samples, dr.res_limits.max_instances on the next page.

4.5.2.1.22 dr.ownership.kind

These columns directly map to the DDS_OwnershipQosPolicy and DDS_Ownership-StrengthQosPolicy for the DataReader created to subscribe to Topic data that is stored in the table. These policies control whether or not the DataReader is allowed to receive changes to an instance of a Topic from multiple DataWriters simultaneously.

The possible values of the **<dr.ownership.kind>** column are:

- "SHARED_OWNERSHIP_QOS" (default value if the column is NULL)

  This setting allows the DataReader to receive updates for an instance of a Topic from multiple DataWriters at the same time.
- "EXCLUSIVE_OWNERSHIP_QOS"

  This setting prevents the DataReader from receiving changes from more than a single DataWriter for an instance of a Topic at the same time.

  The DataReader will receive changes for a topic instance from the DataWriter with the greatest value of ownership strength. If the liveliness of the DataWriter fails or if the DataWriter fails to write within a deadline period, then the DataReader will receive published changes to the topic instance from the DataWriter with the next highest ownership strength.

See the *Connext DDS* documentation for more details on how this QoS policy may be used. See also 4.5.2.1.19 dr.liveliness.lease_dur on page 74, 4.5.2.1.20 dr.deadline.period on page 74, and 4.5.1.1.15 dw.ownership.kind, dw.ownership_strength.value on page 57.

4.5.2.1.23 dr.time_filter.min_sep

This column specifies the minimum separation duration between subsequent samples for the DDS_TimeBasedFilterQosPolicy for the DataReader created to subscribe to Topic data that is stored in the table. By setting the values in these columns, the user configures the DataReader to see at most one change every the minimum_separation period.

The possible values of the **<dr.time_filter.min_sep.sec>** (seconds) and **<dr.time_filter.min_sep.nsec>** (nanoseconds) columns are:

- A 0 minimum separation duration is specified if both columns are NULL or contain the value 0. This is the DDS default value. With this value, the DataReader is potentially interested in all the samples.
- A non-zero value representing the number of seconds and nanoseconds for the minimum separation duration. That value must be smaller than the deadline period and contained in the interval [0, 1 year].

See the *Connext DDS* documentation for more details on how this QoS policy may be used.

4.5.2.1.24 dr.res_limits.max_samples, dr.res_limits.max_instances

These columns set some parameters for the DDS_ResourceLimits QosPolicy for the DataReader created to subscribe to Topic data that is stored in the table. In particular, they control the amount of memory that the system is allowed to allocate for storing published data values as well as the total number of data instances (different primary keys) that can be handled by the DataReader.

A value of -1 for either of these columns means infinite. This is also the default value for these columns if they are NULL. An infinite setting means that the DataReader is allowed to allocate memory as needed to store received table changes and manage new keys.

The number of keys that the DataReader is allowed to manage places an upper limit on the number of rows that the related table in the database can have.

See the *Connext DDS* documentation for more details on how this QoS policy may be used.

4.5.2.1.25 dr.unicast.receive_port

This column is used to configure the unicast port on which the DataReader will receive data. When the default value (NULL or 0) is used, the actual port number is determined by a formula as a function of the domain ID.

4.5.2.1.26 dr.multicast.receive_address

This column is used to set a multicast address for the DataReader to receive values for the Topic. The column maps to the DDS_TransportMulticastQosPolicy of the DataReader.

The possible values for the **<dr.multicast.receive_address>** column are:

- NULL

  A NULL column means that the DataReader will receive Topic data using unicast.
- A string that contains a valid multicast address in the form "xxx.xxx.xxx.xxx".

  The DataReader for the table will subscribe to the Topic on the multicast address provided.

See the *Connext DDS* documentation for more details on how this QoS policy may be used.

4.5.2.1.27 dr.multicast.receive_port

This column configures the multicast port on which the DataReader will receive data. When the default value (NULL or 0) is used, the actual port number is determined by a formula as a function of the domain ID.

Note that the value of this field is ignored when **dr.multicast.receive_address** is NULL.

4.5.2.1.28 metadata.timestamp_type

This column specifies the SQL type used to store timestamps. Possible values are "BIGINT" and "TIMESTAMP".

The BIGINT type is stored by composing two 4-byte integers for sec and nanosec:

bigintTimestamp = (sec << 32) | nanosec

Thus it can represent the full range of times.

The TIMESTAMP type uses a SQL timestamp. Dates outside the range supported by the database type will be set to zero. The resolution may also be limited by the underlying database. Thus, this timestamp and the DDS timestamps as seen by applications may not exactly match.

4.5.2.1.29 metadata.include_fields, metadata.exclude_fields

These lists determine the columns used to store sample metadata within user tables. Each is a comma-separated list of metadata fields. Spaces are not allowed. The excluded fields have priority, so a field that is in both lists will be excluded. Possible fields are:

- ALL (only allowed for include_fields) - store all available fields
- SOURCE_TIMESTAMP - store SampleInfo source_timestamp. The name of the column in the user topic table is RTIDDS_SOURCE_TIMESTAMP
- RECEPTION_TIMESTAMP - store SampleInfo reception_timestamp. The name of the column in the user topic table is RTIDDS_RECEPTION_TIMESTAMP

4.5.2.1.30 table_schema

The **table_schema** column determines how DDS samples are stored in a database table. *Database Integration Service* supports three schemas:

- **FLATTEN:** The SQL table schema is derived from the typecode following the IDL type to SQL type mapping described in 5.2 Flatten Data Representation Mapping on page 90.
- *[Only for MySQL and PostgreSQL]* **JSON**: The DDS sample content is stored in a JSON column called SAMPLE following the format described in 5.3 JSON Data Representation Mapping on page 99. Publication of DDS samples with JSON schema is not supported.
- *[Only for PostgreSQL]* **JSONB**: The DDS sample content will be stored in a JSONB column (binary JSON representation). Publication of DDS samples with JSON schema is not supported.

The default value is FLATTEN.

4.5.2.1.31 RTIRTC_SCN

The System Change Number (SCN) column is available only for connections to a MySQL database. The value of this column is automatically maintained by *Database Integration Service* and is usually of no interest to the application. For more information about the RTIRTC_SCN column see 4.6 User-Table Creation on page 82.

4.5.3 Table Info

The meta-table **RTIRTC_TBL_INFO** stores meta information associated with the user tables.

When a table is automatically created by the *Database Integration Service* daemon (see 4.6 User-Table Creation on page 82), its TypeCode is stored in **RTIRTC_TBL_INFO** as a sequence of octets. When the

Database Integration Service daemon is restarted, the persisted TypeCodes corresponding to existing publications and subscriptions will be made available to other *Connext DDS* applications.

4.5.3.1 Table Info Schema

The **RTIRTC_TBL_INFO** table is created with the following SQL statement:

MySQL:

```
Create Table RTIRTC_TBL_INFO (
    table_owner VARCHAR(128) NOT NULL,
    table_name VARCHAR(128) NOT NULL,
    type_code VARBINARY(65000),
    RTIRTC_SCN BIGINT DEFAULT 0,
    PRIMARY KEY(table_owner,table_name)
)
```

PostgreSQL:

```
Create Table RTIRTC_TBL_INFO (
    table_owner VARCHAR(128) NOT NULL,
    table_name VARCHAR(128) NOT NULL,
    type_code BYTEA,
    PRIMARY KEY(table_owner,table_name)
)
```

Table 4.22 RTIRTC_TBL_INFO Table Schema describes the meta-table columns. Detailed column descriptions follow below.

Table 4.22 RTIRTC_TBL_INFO Table Schema

Column Name	SQL Type	Nullable	Default if NULL
table_owner	VARCHAR(128)	No	N/A
table_name	VARCHAR(128)	No	N/A
type_code	VARCHAR(65000)	Yes	NULL

4.5.3.1.1 table_owner, table_name

These columns specify the user table associated with the meta information described in the other columns.

Because a DBMS uses a combination of **<table_owner>.<table_name>** to identify a table, both of these columns must have valid values.

Notes:

- In MySQL, the value of the **table_owner** column corresponds to the table schema or database name.

- In PostgreSQL, the value of the **table_owner** column corresponds to <database name>[.<schema_name>]. If you do not provide a schema, *Database Integration Service* will use “public”.

4.5.3.1.2 type_code

This column contains the TypeCode information associated to the user table identified by **<table_owner>.<table_name>**.

The TypeCode information stored in this table is used when publications and subscriptions are created after the *Database Integration Service* daemon is restarted.

4.5.4 Log Table

A meta-table named **RTIRTC_LOG** is used to store log messages generated by the daemon. Whether or not this table is created and used depends on the **-loglevel** option (see 4.2 Command-Line Parameters on page 22) and the LOGTODB and LOGHISTORY *Database Integration Service* daemon connection attributes (see 4.4.4.2 Database Mapping Options on page 34).

You should treat the contents of this table as *read-only.* There is no reason for users to modify this table. The number of rows in the Log table is controlled by the LOGHISTORY connection attribute. If set to -1, the table will hold as many log messages as generated by the *Database Integration Service* daemon. Otherwise, the daemon will only store the last *n* log messages as specified by LOGHISTORY, using the table as a circular buffer.

You may use the “id” column to determine the last log message that was generated by the daemon (see 4.5.4.1.1 id on the next page).

4.5.4.1 Log Table Schema

The **RTIRTC_LOG** table is created with the following SQL statement.

MySQL:

```
Create Table RTIRTC_LOG (
    id INTEGER NOT NULL,
    ts TIMESTAMP NOT NULL,
    type VARCHAR(7) NOT NULL,
    function VARCHAR(64) NOT NULL,
    line INTEGER,
    code INTEGER,
    native_code INTEGER,
    message VARCHAR(2048) NOT NULL
)
```

PostgreSQL:

```
Create Table RTIRTC_LOG (
    id INTEGER NOT NULL,
    ts TIMESTAMP NOT NULL,
    type VARCHAR(7) NOT NULL,
```

```
    function VARCHAR(64) NOT NULL,
    line INTEGER,
    code INTEGER,
    native_code INTEGER,
    message VARCHAR(2048) NOT NULL
)
```

Each column of the Log meta-table stores a different portion of a log message generated by the *Database Integration Service* daemon. Table 4.23 RTIRTC_LOG Table Schema describes these columns. Detailed column descriptions follow below the table.

Table 4.23 RTIRTC_LOG Table Schema

Column Name	SQL Type	Nullable	Default if NULL
id	INTEGER	NO	N/A
ts	TIMESTAMP	NO	N/A
type	VARCHAR(7)	NO	N/A
function	VARCHAR(64)	NO	N/A
line	INTEGER	NO	None
code	INTEGER	YES	None
native_code	INTEGER	YES	None
message	VARCHAR(1024)	NO	N/A

4.5.4.1.1 id

This column stores a strictly incrementing integer for each log message that is generated by the daemon. The largest value in the **id** column is the last message that was produced.

4.5.4.1.2 ts

This column stores the system timestamp of when the log message was generated.

4.5.4.1.3 type

This column stores the kind of log message. Possible values are: “ERROR”, “WARNING”, “STATUS”, and “SPECIAL”. “SPECIAL” messages are ones that are always printed independently of the log level.

4.5.4.1.4 function, line

These two columns contain the function name and line number of the *Database Integration Service* daemon code where the message was generated. It is useful only to support engineers at RTI.

4.5.4.1.5 code, native_code, message

The **code** column contains the *Database Integration Service* error code that correspond to the message. This column will have NULL entries for messages of type "STATUS".

The **native_code** column will contain the error code of any external APIs, e.g., ODBC, OS, *Connext DDS*, that the daemon has called and returned an error. This column may have NULL entries.

Finally, the **message** column will contain a statement with details on why the message was generated.

For a complete list of possible error codes and messages that can be generated by the *Database Integration Service* daemon, please see Appendix A Error Codes on page 102.

4.6 User-Table Creation

The *Database Integration Service* daemon may create tables automatically for user applications in the database when entries are made in the **RTIDDS_PUBLICATIONS** or **RTIDDS_SUBSCRIPTIONS** metatables (see 4.5.1 Publications Table on page 46 and 4.5.2 Subscriptions Table on page 59). The daemon will create the table with the table owner and table name specified in an entry in one of those tables if:

1. There is no existing table in the database with the same **<table_owner>.<table_name>** identifier.

   and

2. A type corresponding to the **<type_name>** column for the entry has been defined in the XML configuration file (see 4.4.3 Top-Level XML Tags on page 27).

   or

   A typecode corresponding to the **<topic_name>** column for the entry has been discovered.

If either condition above is not satisfied, the daemon will not create the user table. If the user table already exists, the daemon will attempt to use that table when publishing or subscribing to Topics. It is up to the user to create the table with the right schema.

When subscribing to data there are three possible user table schemas configurable using the column table_schema in the RTIDDS_SUBSCRIPTION table (see 4.5.2.1.30 table_schema on page 78):

- **FLATTEN**: The user table schema is derived from the typecode following the IDL type to SQL type mapping described in 5.2 Flatten Data Representation Mapping on page 90.
- [Only for MySQL and PostgreSQL] **JSON**: The DDS sample content is stored in a JSON column called SAMPLE following the format described in 5.3 JSON Data Representation Mapping on page 99. Publication of DDS samples with JSON schema is not supported.
- [Only for PostgreSQL] **JSONB**: The DDS sample content will be stored in a JSONB column (binary JSON representation). Publication of DDS samples with JSON schema is not supported.

When publishing data, Database Integration Service only supports FLATTEN schema.

If the table does not exist and there is no XML definition for the type and the typecode for the IDL type specified by the entry is unknown, the *Database Integration Service* daemon will defer creation of the table until the typecode has been discovered from other applications on the network that are using *Connext DDS*. See 4.1.4 Typecode and TypeObject on page 21 for more details on how the daemon uses typecodes.

If the table is created by the *Database Integration Service* daemon, the daemon may add up to 8 additional columns (9 in MySQL) that store meta-data used by the daemon when storing data received via *Connext DDS* or sending table changes via *Connext DDS*. Although optional, there are specific operating scenarios where these meta-columns are required for the proper operation of the daemon. We suggest that the user understands the purpose of the meta-columns, and if the user applications create the tables used by the *Database Integration Service* daemon, the user code itself should add the meta-columns to the table schema when appropriate. If needed, RTIDDS_SOURCE_TIMESTAMP and RTIDDS_RECEPTION_TIMESTAMP must directly follow user data columns and the source column must precede the reception column. The other meta-columns may be in any order.

The meta-columns that may be created are:

- **RTIDDS_DOMAIN_ID** and **RTIRTC_REMOTE**

  These two SQL INTEGER columns are always added to the tables created by the daemon. These additional columns are used by the daemon when user has created entries in both the **RTIDDS_PUBLICATIONS** and **RTIDDS_SUBSCRIPTIONS** meta-tables for the same user table. In that situation, changes to the table made by local user applications will be published via *Connext DDS* at the same time that the daemon itself may store data into the table received via *Connext DDS*.

  Database Integration Service daemon uses these meta-columns in order to prevent the republishing of tables values that were changed because they were received via *Connext DDS*. User applications that create the table do not need to add these columns if the daemon is configured only to publish data from the table or to store data into the table.

  However, it is essential that these columns do exist for the situation where both publications and subscriptions are tied to the same table. If the meta-columns are omitted, then when *Database Integration Service* daemon receives data via *Connext DDS*, it will be echoed (republished) as a change to the table.
- **RTIRTC_KEY**

  This SQL INTEGER column is added by the daemon if the IDL type that is used to create the table does not contain any fields marked as a topic key (i.e., non-keyed IDL types). In such cases, the <**RTIRTC_KEY**> column will be added to the table as the primary key column. The value in that column will always be 0. Thus, there is only a single instance of the Topic which means the table will only ever have a single row (subject to whether or not the user wants the table to store historical

value of data instances, see the details for the <**RTIRTC_-HISTORY_SLOT**> and <**RTIRTC_HISTORY_ORDER**> meta-columns below).

If the IDL type does have key fields, then the fields will be mapped into columns that are marked as primary keys. This meta-column is not added, and the table can contain as many rows as there are different instance keys (primary keys).

- **RTIRTC_HISTORY_SLOT** and **RTIRTC_HISTORY_ORDER**

  These SQL INTEGER columns are used to implement the ability of the *Database Integration Service* daemon to store multiple values (historical) of the same data instance into a table. Usually, a single data instance maps to a single row of a table. As new values for the instance is received by the daemon, the value of the same row is changed.

  However, users may use the <table_history_depth> columns (see 4.5.1.1.5 table_history_depth on page 51 and 4.5.2.1.5 table_history_depth on page 65) of the **RTIDDS_PUBLICATIONS** and **RTIDDS_SUBSCRIPTIONS** meta-tables to direct the daemon to store multiple past values of a data instance. These values are be stored in multiple rows of a table. To support table history, the daemon must add the meta-columns <**RTIRTC_HISTORY_SLOT**> and <**RTIRTC_HISTORY_ORDER**> to a table. They will only be added if the <table_history_depth> column for an entry is non-NULL and has a non-0 value.

  The <**RTIRTC_HISTORY_SLOT>** is an auto-increment column that will also be added as a primary key column of the table.

  The **<RTIRTC_HISTORY_ORDER>** is a column that will contain a number that is incremented as data is stored into the table. The oldest row of an instance will have the lowest value for this column whereas the most recent row of an instance will have the highest value.

- **RTIDDS_SOURCE_TIMESTAMP** and **RTIDDS_RECEPTION_TIMESTAMP**

  These columns are used to store the source and reception timestamp metadata fields. Columns are only created if needed, based on the configuration in the RTIDDS_PUBLICATIONS or RTIDDS_SUBSCRIPTIONS meta-tables (see 4.5.2.1.29 metadata.include_fields, metadata.exclude_fields on page 78).

  If present, they will directly follow the user data columns, source then reception. They are not needed if timestamps are not being stored by a subscription or used by a publication as a source timestamp.

- **RTIRTC_SCN**

  The System Change Number (SCN) meta-column (SQL_BIGINT) is only required for connections to a MySQL database. The SCN meta-column is used to detect committed changes in a table. Its value is automatically assigned by the MySQL server.

Each time there is a change in a table row or a new row is inserted, the MySQL server assigns a new SCN value to the column RTIRTC_SCN. The assignment is done during the execution of the BEFORE UPDATE/INSERT trigger installed by the *Database Integration Service* daemon.

4.7 Support for Extensible Types

Database Integration Service includes partial support for the OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3. This section assumes that you are familiar with Extensible Types and you have read the RTI Connext DDS Core Libraries Extensible Types Guide.

- The *Database Integration Service* daemon can publish and subscribe to topics associated with final and extensible types. Optional members are not supported with final and extensible types, but are supported with mutable types.
- The *Database Integration Service* daemon can subscribe to topics associated with mutable types when using JSON table_schema mode.
- When using FLATTEN table schema mode, the *Database Integration Service* daemon can publish and/or subscribe to only one version of the types associated with a topic within a domain
- By default, *DataReaders* created when a new entry is inserted in the RTIDDS_SUBSCRIPTIONS table are configured with the TypeConsistencyEnforcementQosPolicy's **kind** set to DISALLOW_TYPE_COERCION. You can override this setting only when using JSON table_schema mode.
- You can select the type version for a given topic within a domain in two different ways:
  - By providing the type description via XML and then referring to that description using the column **type_name** in the RTIDDS_SUBSCRIPTIONS and/or RTIDDS_PUBLICATIONS tables.
  - By pre-creating the database table used to store/publish the data with the right schema.

Since by default the *DataReaders* created by the *Database Integration Service* daemon have their TypeConsistencyEnforcementQosPolicy's **kind** set to **DISALLOW_TYPE_COERCION**, they will not match with DataWriters whose types are not equal to the *DataReader*'s type. You can modify this default behavior only when using JSON table_schema mode. You can learn more here: *Type Consistency Enforcement*, in the *Type Safety and System Evolution* chapter of the RTI Connext DDS Core Libraries Extensible Types Guide.

4.8 Enabling RTI Distributed Logger in Database Integration Service

Database Integration Service provides integrated support for *RTI Distributed Logger*.

When you enable *Distributed Logger, Database Integration Service* will publish its log messages to *Connext DDS*. Then you can use *RTI Monitor*[1] to visualize the log message data. Since the data is provided in a *Connext DDS* topic, you can also use *rtiddsspy* or even write your own visualization tool.

To enable *Distributed Logger*, modify the *Database Integration Service* XML configuration file. In the <general_options><administration> section, add the <distributed_logger> tag as shown in the example below.

```
<real_time_connect name="default">
    <general_options>
        <administration>
            <domain_id>0</domain_id>
 <distributed_logger>
                <enabled>true</enabled>
            </distributed_logger>
        </administration>
    </general_options>
    ...
</real_time_connect>
```

Replace the value of <domain_id> with the domain ID that *Database Integration Service* will use to send log messages when *Distributed Logger i*s enabled.

There are more configuration tags that you can use to control *Distributed Logger's* behavior. For example, you can specify a filter so that only certain types of log messages are published. For details, see the *Distributed Logger* part of the RTI Connext DDS Core Libraries User's Manual.

4.9 Enabling RTI Monitoring Library in Database Integration Service

To enable monitoring of the Entities that are created by *Database Integration Service*, you must specify the property **rti.monitor.library** in the QoS of the participants that you want to monitor. For example:

```
<domain_participant_qos>
    <property>
        <value>
            <element>
                <name>rti.monitor.library</name>
                <value>rtimonitoring</value>
                <propagate>false</propagate>
            </element>
        </value>
    </property>
</domain_participant_qos>
```

[1] *RTI Monitor* is a separate GUI application that can run on the same host as your application or on a different host.

The QoS associated with the DomainParticipants that are created by *Database Integration Service* can be configured in three different ways:

- By setting the attribute **is_default_qos** in the tag <qos_profile> containing the <domain_participant_qos> to true. In this case, that profile is the default configuration for all the Entities created by the *Database Integration Service* daemon.

  For a list of XML files where you can declare the QoS Profile, see 4.4.1 How to Load the XML Configuration on page 25
- By referring to a profile using the XML tag <profile_name> within <publication> and <subscription> (see 4.4.4.4 Initial Subscriptions and Publications on page 38).
- By referring to a profile in the **profile_name** column of the tables RTIDDS_PUBLICATIONS or RTIDDS_SUBSCRIPTIONS (see 4.5.1 Publications Table on page 46 and 4.5.2 Subscriptions Table on page 59).

Notice that since *Database Integration Service* is statically linked with *RTI Monitoring Library*, you do not need to have it in your Path (on Windows systems) or LD_LIBRARY_PATH (on Linux systems).

For details on how to configure the monitoring process, see the *Monitoring Library* part of the RTI Connext DDS Core Libraries User's Manual.

Chapter 5 IDL/SQL Semantic and Data Mapping

This chapter describes the semantic and data representation mapping that *RTI Database Integration Service* uses to connect DDS-based applications such as *Connext DDS* to MySQL databases.

Connext DDS provides an API to send and receive data between networked applications following a publish/subscribe paradigm. The corresponding standard API in the database world is SQL. Both the *Connext DDS* and SQL APIs have various language bindings in C/C++ and Java.

How *Database Integration Service* maps actions (semantics) and data types (data representation) from *Connext DDS* to SQL and vice versa is described in the following sections.

5.1 Semantic Mapping

Connext DDS applications publish and subscribe to topics which are named data structures using functions like **DDSDataWriter::write()** and **DDSDataReader::read()**. Relational databases contain tables that applications access data using SQL operations such as **INSERT**, **UPDATE**, **DELETE** and **SELECT**. Table 5.1 Connext DDS-DBMS Semantic Models describes the mapping between *Connext DDS* and relational database semantic models.

Table 5.1 Connext DDS-DBMS Semantic Models

Connext DDS	Relational Database	Details
Accessed via *Connext DDS* API Various language bindings (e.g. C/C++, Java)	Accessed via SQL Various language bindings (e.g. C/C++ and ODBC, Java and JDBC)	
Data structures Defined by IDL (Interface Description Language).	Tables Defined by table schema.	Fields in data structures are mapped to columns of a table. Each row of a table represents a different value for a data structure. The exact mapping of IDL data structures to table schemas is described in 5.2 Flatten Data Representation Mapping on the next page.
Topic Identified by a name string. DataWriter can publish values for Topics and DataReaders can subscribe	Table Identified by a name string. Applications can write values or read values from tables using SQL.	Topic names and table names do not have to be the same when making a correspondence between a Topic and a database table.
Data values	Rows in table No history: A single row in a table. History: Multiple rows in a table.	When the *Database Integration Service* daemon table history option is turned OFF (see Sections 4.5.1.1.5 table_history_depth on page 51 and 4.5.2.1.5 table_history_depth on page 65), only the last value of a topic instance is stored in the table. So a non-keyed topic will be stored in a single row whereas for keyed topics, there will be as many rows as there are topic instances. When the *Database Integration Service* daemon table history option is turned ON, each instance will occupy up to a user-settable maximum number of rows so that the last N values received for the Topic are stored in the table. When N values have been stored, the N rows are used as a circular buffer so that new values received will overwrite the oldest values stored.
Key IDL data types may contain one or more fields that are used to distinguish different instances of the Topic.	Primary key Most relational databases require table schemes to identify one or more columns to act as the primary key for the table.	Keys are mapped to the primary keys of a table. When a table is created by the *Database Integration Service* daemon, the columns corresponding to the IDL key fields will be created as primary key columns. For tables created by user code, the correspondence of IDL key fields to table primary key columns must be set correctly.
DDSDataWriter::write()	SQL **INSERT** or **UPDATE**	Values published for Topics will be stored into a database table by the *Database Integration Service* daemon. Table rows modified by SQL **INSERT** or **UPDATE** commands will be published by the *Database Integration Service* daemon as values of Topics.
DDSDataReader::take() DDSDataReader::read()	SQL **SELECT**	
DDSDataWriter::dispose()	SQL **DELETE**	When SQL **DELETE** is used to delete a row from a table, the *Database Integration Service* daemon will call **DDSDataWriter::dispose()** to dispose the instance corresponding to the row. If a user application calls **DDSDataWriter::dispose()** to dispose an instance, the *Database Integration Service* daemon may be configured to delete or keep the corresponding rows.

5.2 Flatten Data Representation Mapping

In *Connext DDS*, data is stored in data structures or classes defined using the Interface Definition Language (IDL). In relational databases, data is stored in tables defined using SQL table schemas. While there is a good correspondence of IDL primitive data types to SQL data types, this mapping is not one-to-one. Both IDL and SQL have data types that the other does not define nor has an unambiguous mapping. In addition, many complex data structures in IDL such as unions and data structures that contain embedded data structures do not have equivalents in SQL.

This section describes the FLATTEN mapping used by the *Database Integration Service* daemon when taking data received with DDS and storing it in tables, or taking data from tables and publishing it with *Connext DDS*.

5.2.1 IDL to SQL Mapping

Identifiers are used for the names of table columns in SQL and names of fields within an IDL structure. SQL identifiers are a superset of IDL identifiers. Because of that, an IDL identifier can always be used as a SQL identifier. However, there are some SQL identifiers that cannot be used as IDL identifiers. For example, SQL allows special characters like '#' to be part of an identifier, whereas IDL does not.

There are two kinds of SQL identifiers: *quoted identifiers* and *basic identifiers*:

- Quoted identifiers can use any combination of characters. These identifiers need to be surrounded by double quotes when referenced.
- The definition of a basic identifier changes depending on the database vendor:
    - In MySQL, a basic identifier can consist of any letters (A to Z), decimal digits (0 to 9), $, or underscore (_).

If the daemon creates the user table, it will use quoted strings for the identifiers of the table and column names only if they cannot be considered as basic identifiers according to the previous definitions. Thus, user applications should also use quoted strings when referring to those column and table names in their SQL statements.

Ordinarily, the name of a field in an IDL data structure can just be used as the name of a column in a table. In fact, for those data types with clear and obvious mappings, the column name can be independent of the field name used in the IDL type. However, because there is no one-to-one mapping of all IDL data types to all SQL data types, for certain types, the column names used in SQL table schemas *must* follow certain conventions that tie them to the names of the fields of IDL types from which they are mapped. This is true for only the small subset of primitive IDL data types and for the complex IDL data types that would otherwise have ambiguous mappings, i.e., multiple ways to map IDL to SQL or vice versa.

The *Database Integration Service* daemon scans for, identifies and uses special mappings of column names when serializing and deserializing IDL data to and from a table in a database. There are two types of special mappings, hierarchical naming and suffixes.

Hierarchical Naming

Complex IDL types may have fields that are actually embedded structures, so a field may actually contain multiple values. In SQL, columns usually contain a single value for each column element, although there are a few types like BINARY(x) CHAR(x), VARBINARY(x) and VARCHAR(x) that can store multiple values of the same type in a single column element. To map complex IDL types to SQL table schemas, embedded data structures are unfolded so that elements of an embedded structure are stored individually in separate columns.

When the *Database Integration Service* daemon creates a table schema from a Topic, it will automatically flatten hierarchical data structures into tables. In doing so, the names of columns that store the fields of embedded structures will have hierarchical names. For example, given this IDL definition:

```
struct bar {          struct foo {
   long one;             bar element;
   long two;          };
};
```

The table constructed from a Topic which uses the foo type would have the following schema by default:

```
CREATE table foo ( INTEGER element.one, INTEGER element.two )
```

The *Database Integration Service* daemon allows the configuration of the separator character (‘.’) using the attribute *IdentifierSeparatorChar* defined in the general options of the configuration file (described in 4.4.4.1 General Options on page 30).

While for most embedded structures, the hierarchical naming of columns is not needed for the *Database Integration Service* daemon to handle type translation correctly, the proper hierarchical naming of columns is essential for the daemon to serialize and deserialize IDL unions and sequences. These types are variable in length, however the table must have enough columns to hold the maximum size of the IDL data type.

Hierarchical naming allows the *Database Integration Service* daemon to identify columns that form a embedded, complex element.

For variable-length types (other than sequences of “char”, “wchar” or “octet”), an extra column with the suffix “#length” is also added to the table to hold the current length of the type. Also, each column that represents a field in an element of the variable-length type must have a suffix “[x]” in its name that identifies the index of the element, where

x = 0 to (max_length - 1)

During the serialization and deserialization process, the daemon will usually be working with less than the maximum length of data, and thus, will need to use the hierarchical naming along with the suffix to determine which columns belong to unused elements that should be skipped.

This hierarchical flattening operation of member names may lead to very long column names in the generated table and can easily exceed the maximum number of characters supported by the database (some databases limit the column names to 30 characters).

To reduce the length of the generated names, you can instruct *Database Integration Service* to consider only the first *n* and the last *m* characters of the flattened name, and eventually resolve any conflict by using a progressive number between the prefix and the suffix. The two tags **<idl_member_prefix_max_length>** and **<idl_member_suffix_max_length>** (see Table 4.8 Database Mapping Options), defined in the configuration file (described in 4.4 Configuration File on page 25) and the columns **idl_member_prefix_max_length** and **idl_member_suffix_max_length** in the meta-tables (described in 4.5.1.1.7 idl_member_prefix_max_length, idl_member_suffix_max_length on page 53) tell the daemon the values to use. (The values defined in the meta-table have precedence over the values defined in the configuration file.)

Suffixes

Suffixes are also needed for column names when multiple IDL primitive types map into the same SQL type. Because there are more IDL primitive types than SQL primitive types, a full mapping will result in the use of the same SQL type to hold more than one IDL type. For example, an IDL “long double” has no equivalent in SQL. Thus, a SQL BINARY(16) does double duty and is used to store both an IDL “long double” as well as an IDL “octet[16]”.

If a “long double” could be treated the same as an “octet[16]” by the *Database Integration Service* daemon, then there would be no issue and no special name mapping would be needed. However, because the representation of a “long double” is Endianess-dependent while an “octet[16]” is not, the *Database Integration Service* daemon *must* use the column name to decide whether or not a SQL BINARY(16) value needs to be byte swapped or not when converting to an IDL data type. Since “long double” has no equivalent SQL type, a “.ld” must be appended to the name of a SQL BINARY(16) column that is used to store one.

Similarly, a suffix of “.str” is used to indicate that a SQL VARCHAR(x) stores IDL “string”, which is a NULL-terminated sequence of the primitive type “char”. Without the suffix in the column name, a SQL VARCHAR(x) naturally stores a sequence of chars-the IDL type “sequence <char,x>.

Note: Because of the use of suffixes in the mapping of identifiers of certain IDL datatypes, the identifiers "**str**", "**ld**", and "**bin**" are reserved keywords that should not be used as the name of fields in IDL structures. For example, the following IDL definitions have the same SQL mapping which would in result in the incorrect treatment of the type "**Foo2**" by the daemon. Each would result in a table schema that would have the ten columns named "**my_field[0].str**", "**my_field[1].str**", ..., "**my_field[2].str**".

```
struct Foo1 {                    struct Bar {
    string<10> my_field;  and        sequence<char,10> str;
};                               };
                                 struct Foo2 {
                                     struct Bar my_field;
                                 }
```

5.2.2 Primitive Types Mapping

The following tables show the mapping between basic types in IDL and SQL:

- Table 5.2 Basic Types in IDL and SQL (MySQL)
- Table 5.3 Basic Types in IDL and SQL (PostgreSQL)

Table 5.2 Basic Types in IDL and SQL (MySQL)

IDL Type	IDL Field Name	SQL Type	Table Column Name
char [a]	my_field	CHAR(1)	"my_field"
char[x] [b]	my_field	CHAR(x)	"my_field"
sequence<char,x>	my_field	VARCHAR(x)	"my_field"
wchar [c]	my_field	NCHAR(1)	"my_field"
wchar[x] [d]	my_field	NCHAR(x)	"my_field"
sequence<wchar,x>	my_field	NVARCHAR(x)	"my_field"
octet [e]	my_field	BINARY(1)	"my_field"

[a]The format on the wire of "char" and "char[x]" is the same.

[b]The format on the wire of "char" and "char[x]" is the same.

[c]The format on the wire of "wchar" and "wchar[x]" is the same.

[d]The format on the wire of "wchar" and "wchar[x]" is the same.

[e]The format on the wire of "octet"and "octet[x]" is the same.

Table 5.2 Basic Types in IDL and SQL (MySQL)

IDL Type	IDL Field Name	SQL Type	Table Column Name
octet[x][a]	my_field	BINARY(x)	“my_field”
sequence<octet,x>[b]	my_field	VARBINARY(x)	“my_field”
boolean	my_field	TINYINT	“my_field”
short	my_field	SMALLINT	“my_field”
unsigned short	my_field	SMALLINT	“my_field”
long	my_field	INTEGER	“my_field”
unsigned long	my_field	INTEGER	“my_field”
double	my_field	DOUBLE	“my_field”
float	my_field	FLOAT	“my_field”
string<x>	my_field	VARCHAR(x)	“my_field.str”[c]
wstring<x>	my_field	NVARCHAR(x)	“my_field.str”[d]
long long	my_field	BIGINT	“my_field”
unsigned long long	my_field	BIGINT	“my_field”
long double	my_field	BINARY(16)	“my_field.ld”[e]
unsigned long long	my_field	DATETIME	“my_field”

Table 5.3 Basic Types in IDL and SQL (PostgreSQL)

IDL Type	IDL Field Name	SQL Type	Table Column Name
char[f]	my_field	CHAR(1)	“my_field”

[a]The format on the wire of “octet”and “octet[x]” is the same.

[b]The format on the wire of “octet”and “octet[x]” is the same.

[c]The “.str” suffix is used to distinguish between “(w)string<x>” and “sequence<(w)char,x>”.

[d]The “.str” suffix is used to distinguish between “(w)string<x>” and “sequence<(w)char,x>”.

[e]The “.ld” suffix is used to distinguish between “octet[x]” and “long double”.

[f]The format on the wire of “char” and “char[x]” is the same.

Table 5.3 Basic Types in IDL and SQL (PostgreSQL)

IDL Type	IDL Field Name	SQL Type	Table Column Name
char[x][a]	my_field	CHAR(x)	“my_field”
sequence<char,x>	my_field	VARCHAR(x)	“my_field”
wchar	Not supported		
wchar[x]	Not supported		
sequence<wchar,x>	Not supported		
octet[b]	my_field	BYTEA	“my_field.1.bin”[c]
octet[x][d]	my_field	BYTEA	“my_field.x.bin”[e]
sequence<octet,x>	my_field	BYTEA	“my_field.x”[f]
boolean	my_field	BOOLEAN	“my_field”
short	my_field	SMALLINT	“my_field”
unsigned short	my_field	SMALLINT	“my_field”
long	my_field	INTEGER	“my_field”
unsigned long	my_field	INTEGER	“my_field”
double	my_field	DOUBLE PRECISION	“my_field”
float	my_field	REAL	“my_field”
string<x>	my_field	VARCHAR(x)	“my_field.str”[g]

[a] The format on the wire of “char” and “char[x]” is the same.

[b] The format on the wire of “octet” and “octet[x]” is the same.

[c] The “bin” part of the “x.bin” suffix is used to distinguish octet[x] from sequence<octet,x> as PostgreSQL does not support SQL BINARY or VARBINARY. The “x” part of the suffix encodes the size of the octet array as the SQL type BYTEA does not have a bound.

[d] The format on the wire of “octet” and “octet[x]” is the same.

[e] The “bin” part of the “x.bin” suffix is used to distinguish octet[x] from sequence<octet,x> as PostgreSQL does not support SQL BINARY or VARBINARY. The “x” part of the suffix encodes the size of the octet array as the SQL type BYTEA does not have a bound.

[f] The “x” part of the suffix encodes the maximum size of the octet sequence as the SQL type BYTEA does not have a bound.

[g] The “.str” suffix is used to distinguish between “string<x>” and “sequence<char,x>”.

Table 5.3 Basic Types in IDL and SQL (PostgreSQL)

IDL Type	IDL Field Name	SQL Type	Table Column Name
wstring<x>	Not supported		
long long	my_field	BIGINT	“my_field”
unsigned long long	my_field	BIGINT	my_field
long double	Not supported		

5.2.3 Enum Types Mapping

IDL enumeration fields are mapped to columns of type SQL INTEGER. No special naming is required.

5.2.4 Simple IDL Structures

Simple IDL structures containing only basic or primitive types directly map to SQL schemas with fields in the structure becoming columns in the table. Table 5.4 Simple Structures in IDL and SQL shows the mapping of a simple structure between IDL and SQL

Table 5.4 Simple Structures in IDL and SQL

IDL Types	SQL Table Schema

`struct MyStruct {`
`  long my_key_field; //@key[a]`
`  short my_short_field;`
`};`
	`CREATE TABLE "MyStructContainer" (`
	`"my_key_field" INTEGER NOT NULL,`
	`"my_short_field" SMALLINT NOT NULL,`
	`  PRIMARY KEY(my_key_field)`
	`);`

5.2.5 Complex IDL Structures

IDL structures that contain more complex fields, fields that are structures, unions, or sequences and arrays of types other than “octet”, “char” or “wchar” are mapped to SQL tables by flattening the embedded structures so that their fields are all at the top (and only) level.

Structure fields

Elements of embedded structures map into individual table columns with names that are hierarchically composed from the name of the field in the embedded structure and the name of the embedded structure field itself. This naming convention is not required for serialization to work properly. Just as long as the column types map to the types of the embedded structure, then the *Database Integration Service* daemon will properly handle the data irrelevant of the actual column name.

[a] IDL fields marked as keys are mapped to the primary keys of SQL tables.

Table 5.5 Nested Structures in IDL and SQL shows the mapping of a complex structure between IDL and SQL.

Table 5.5 Nested Structures in IDL and SQL

IDL Type	SQL Table Schema

`struct MyStruct {`
`  short my_short_field;`
`  long  my_long_field;`
`};`
`struct MyStructContainer {`
`  long my_key_field; //@key`
`  MyStruct my_struct_field;`
`};`
	`CREATE TABLE "MyStructContainer" (`
	`"my_key_field" INTEGER NOT NULL,`
	`"my_struct_field.my_short_field" SMALLINT NOT NULL,`
	`"my_struct_field.my_long_field" INTEGER NOT NULL,`
	`PRIMARY KEY(my_key_field)`
	`);`

Union fields

IDL unions are mapped by adding an extra column with the name "**_d**" to represent the discriminator that is used to indicate which type is actually stored by the union. These unions are also known as "switched unions". All of the individual union fields are mapped to corresponding columns in a table. However, only one of these columns will contain valid data as indicated by the discriminator column, "**_d**".

If the *Database Integration Service* daemon creates the table from an IDL containing an union, it will generate the data columns with hierarchical names from the name of the union field and the name of the union itself. In addition, the values of the switch/case statement in the IDL union are encoded into the names of the data columns as well, e.g., "**.c(0,1).**", "**.c(2).**", "**.(def).**".

This naming convention **is required** for the proper serialization and deserialization of unions. The *Database Integration Service* daemon uses the name of the fields when processing an IDL union to know which column(s) correspond to the value of the discriminator.

Table 5.6 Union Fields in IDL and SQL shows the mapping of an union between IDL and SQL.

Table 5.6 Union Fields in IDL and SQL

IDL Type	SQL Table Schema

`union MyUnion switch(long) {`
`  case 0:`
`  case 1:`
`  long my_long_field;`
`  case 2:`
`  double my_double_field;`
`  default:`
`  short my_short_field;`
`};`
`struct MyUnionContainer {`
`  long my_key_field; //@key`
`  MyUnion my_union_field;`
`};`
	`CREATE TABLE "MyUnionContainer" (`
	`"my_key_field" INTEGER NOT NULL,`
	`"my_union_field._d" INTEGER NOT NULL,`
	`"my_union_field.c(0,1).my_long_field" INTEGER,`
	`"my_union_field.c(2).my_double_field" DOUBLE,`
	`"my_union_field.c(def).my_short_field" SMALLINT,`
	`PRIMARY KEY(my_key_field)`
	`);`

5.2.6 Array Fields

For array fields where the array type is different from “octet”, “char” and “wchar”, an IDL array type is stored as consecutive columns of the same type in a SQL table. If the *Database Integration Service* daemon creates a table from an IDL type that contains an array, it will create the column names using a naming convention that prevents name collisions. By default, the daemon simply adds the suffix “**[i]**”, where “i” is the array index of that element (beginning at 0 for the first index). The open bracket and close bracket characters can be configured using the tags in the configuration file **<open_bracket_char>** and **<close_bracket_char>** (see Table 4.8 Database Mapping Options*)*. Note, this naming convention is not required for the *Database Integration Service* daemon to serialize/deserialize IDL array fields.

Note that array fields of type “octet”, “char” and “wchar” are mapped into a single column element of the corresponding SQL types BINARY(x), CHAR(x) and WCHAR(x), respectively. Table 5.7 Array Fields in IDL and SQL shows a mapping of an array field between IDL and SQL.

Table 5.7 Array Fields in IDL and SQL

IDL Type	SQL Table Schema

`struct MyArrayContainer {`
`  long my_key_field; //@key`
`  short my_arr_field[2];`
`};`
	`CREATE TABLE "MyArrayContainer" (`
	`"my_key_field" INTEGER NOT NULL,`
	`"my_arr_field[0]" SMALLINT NOT NULL,`
	`"my_arr_field[1]" SMALLINT NOT NULL,`
	`  PRIMARY KEY("my_key_field")`
	`);`

5.2.7 Sequence Fields

Sequences are basically variable-sized arrays that have a maximum length and carry an additional integer that indicates the current size. The mapping of IDL sequences to a table schema is similar to the array mapping, with the following differences:

- An extra column is added with the suffix “**#length**”, used to store the current length of the sequence.
- The total number of columns created is equal to the maximum number of elements that the sequence can hold, although the number of columns containing valid data at a given time is stored in the “**#length**” column.
- The naming convention of adding the suffix “**[i]**” to each column is required for the *Database Integration Service* daemon to handle the mapping between IDL and SQL correctly. The open bracket and close bracket characters can be configured using the tags **<open_bracket_char>** and **<close_bracket_char>** (see Table 4.8 Database Mapping Options*)*.
- Sequence elements can contain the NULL value since not all elements may be used at a given time.

Note: Sequences of the IDL types “char”, “wchar” or “octet” map directly into the variable-length SQL types VARCHAR, VARWCHAR, and VARBINARY, respectively. Table 5.8 Sequence Fields in IDL and SQL shows the mapping of a sequence field between IDL and SQL.

Table 5.8 Sequence Fields in IDL and SQL

IDL Type	SQL Table Schema

`struct MySequenceContainer {`
`  long my_key_field; //@key`
`  sequence<short,4> my_seq_field;`
`};`
	`CREATE TABLE "MySequenceContainer" (`
	`"my_key_field" INTEGER NOT NULL,`
	`"my_seq_field#length" INTEGER NOT NULL,`
	`"my_seq_field[0]" SMALLINT,`
	`"my_seq_field[1]" SMALLINT,`
	`"my_seq_field[2]" SMALLINT,`
	`"my_seq_field[3]" SMALLINT,`
	`  PRIMARY KEY("my_key_field")`
	`);`

5.2.8 NULL Values

The *Database Integration Service* daemon converts NULL values into '0'-values when publishing from a SQL table, in the following way:

- numerical types: 0
- fixed-length string types (CHAR, NCHAR): ""
- variable-length types (VARCHAR, NVARCHAR, VARBINARY): length 0
- binary: every byte is set to 0
- timestamp: 0

5.3 JSON Data Representation Mapping

Database Integration Service can be configured to store a DDS sample's content in a JSON column. This is only supported for MySQL and PostgreSQL databases. In MySQL, this is supported as of version 5.7.8. In PostgreSQL, this is supported as of version 9.2.

Table 5.9 Mapping of DDS Sample to JSON Representation describes how to map the content of a DDS Sample Representation to JSON representation.

Table 5.9 Mapping of DDS Sample to JSON Representation

IDL Construct	IDL Type Example	Sample JSON Representation Example

Structs (top level)	`struct AStruct {`
	`  long aLong;`
	`  bolean aBool;`
	`};`
		`{`
		`  "aLong": 1,`
		`  "aBool":  true`
		`}`

Structs (nested)	`struct AnotherStruct {`
	`  long aLong;`
	`  boolean aBool;`
	`};`
	`struct AStruct {`
	`  AnotherStruct anotherStruct;`
	`};`
		`{`
		`  "anotherStruct" : {`
		`  "aLong": 1,`
		`  "aBool":  true`
		`  }`
		`}`

Table 5.9 Mapping of DDS Sample to JSON Representation

IDL Construct	IDL Type Example	Sample JSON Representation Example

Union (top level)	`union AUnion switch (long) {`
	`  case 0:`
	`  long aLong;`
	`  default:`
	`  boolean aBool;`
	`};`
		`{`
		`  “aLong”: 1`
		`}`

Union (nested)	`union AUnion switch (long) {`
	`  case 0:`
	`  long aLong;`
	`  default:`
	`  boolean aBool;`
	`};`
	`struct AStruct {`
	`  AUnion aUnion;`
	`};`
		`{`
		`  "aUnion":{`
		`  "aLong": 1`
		`  }`
		`}`

Arrays
(simple members)
	`struct AStruct {`
	`  long anArrayOfLongs[2];`
	`};`
		`{`
		`  "anArrayOfLongs" : [`
		`  1,`
		`  2`
		`  ]`
		`}`

Arrays
(complex members)
	`struct AnotherStruct {`
	`  float aFloat;`
	`};`
	`struct AStruct {`
	`  AnotherStruct`
	`  anArrayOfStructs[2];`
	`};`
		`{`
		`  "anArrayOfStructs": [`
		`  {`
		`  "aFloat": 0.0`
		`  },`
		`  {`
		`  "aFloat": 1.0`
		`  }`
		`  ]`
		`}`

Sequences
(simple members)
	`struct AStruct {`
	`  sequence<double,2>`
	`  aSequenceOfDoubles;`
	`};`
		`{`
		`  "aSequenceOfDoubles" : [`
		`  1.1,`
		`  1.2`
		`  ]`
		`}`

Sequences
(complex members)
	`struct AnotherStruct {`
	`  float aFloat;`
	`};`
	`struct AStruct {`
	`sequence<AnotherStruct,2>`
	`aSequenceOfStructs;`
	`};`
		`{`
		`  "aSequenceOfStructs": [`
		`  {`
		`  "aFloat": 0.0`
		`  },`
		`  {`
		`  "aFloat": 1.0`
		`  }`
		`  ]`
		`}`

Shorts	`struct AStruct {`
	`  short aShort;`
	`};`
		`{`
		`  "aShort": 3`
		`}`

Unsigned Shorts	`struct AStruct {`
	`  unsigned short aUShort;`
	`};`
		`{`
		`  "aUShort": 2`
		`}`

Enums	`enum AEnum {`
	`  ENUMERATOR_1 = 1,`
	`  ENUMERATOR_2 = 2`
	`};`
	`struct AStruct {`
	`  AEnum anEnum;`
	`};`
		`{`
		`  "anEnum": “ENUMERATOR_1”`
		`}`

Table 5.9 Mapping of DDS Sample to JSON Representation

IDL Construct	IDL Type Example	Sample JSON Representation Example

Longs	`struct AStruct {`
	`  long aLong;`
	`};`
		`{`
		`  "aLong": 2324`
		`}`

Unsigned Longs	`struct AStruct {`
	`  unsigned long aULong;`
	`};`
		`{`
		`  "aULong": 2326`
		`}`

Long Longs	`struct AStruct {`
	`  long long aLongLong;`
	`};`
		`{`
		`  "aLongLong": 23245`
		`}`

Unsigned Long Longs	`struct AStruct {`
	`  unsigned long aULongLong;`
	`};`
		`{`
		`  "aULongLong": 232457`
		`}`

Floats	`struct AStruct {`
	`  float aFloat;`
	`};`
		`{`
		`  "aFloat": 2.3`
		`}`

Doubles	`struct AStruct {`
	`  double aDouble;`
	`};`
		`{`
		`  "aDouble": 3.14`
		`}`

Booleans	`struct AStruct {`
	`  boolean aBool;`
	`};`
		`{`
		`  "aBool": false`
		`}`

Octets	`struct AStruct {`
	`  octet aOctet;`
	`}`
		`{`
		`  "aOctet": "0x00"`
		`}`

Chars	`struct AStruct {`
	`  char aChar;`
	`}`
		`{`
		`  "aChar": "a"`
		`}`

WChars	Not supported	

Strings	`struct AStruct {`
	`  string aString;`
	`}`
		`{`
		`  "aString": "A string!"`
		`}`

WStrings	Not supported	
Long Doubles	Not supported	

Appendix A Error Codes

Table A.1 Database Integration Service Errors and Warnings lists the native error and warning messages that may be logged by the *Database Integration Service* daemon. While some of these messages may actually provide enough information by themselves to help users fix the problem, many have to be used along with other data to help with debugging the issue.

Often several of these messages will be logged for a single problem. A failure at a lower layer will cause log messages to be printed at various levels of the *Database Integration Service* daemon logic. These messages will be valuable to you and to RTI support engineers in debugging issues with *Database Integration Service*.

Table A.1 Database Integration Service Errors and Warnings

Code	Message	Details
0 - 1023 Database Integration Service daemon errors These messages are produced by the logic of the *Database Integration Service* daemon itself.		
0	Unexpected error	Should never occur. Contact support@rti.com if seen.
1	<message>	General error.
2	Error storing RTI DDS sample in table '<table>'	There was an error when storing value received with *Connext DDS* into the database.
3	Error creating <entity>	
4	Error creating <entity> associated to the table '<table>'	
5	Error getting <entity>	
6	<meta-table> entry not valid	There was an entry in a meta-table (RTIDDS_PUBLICATIONS or RTIRTCSUBSCRIPTIONS) that was not valid.

Table A.1 Database Integration Service Errors and Warnings

Code	Message	Details
7	Error creating '<type>' SQL statement	There was an error when creating or preparing a SQL statement.
8	Error creating table '<table>'	
9	Error opening RTI DDS connection	There was a problem initializing *Connext DDS*.
10	The type of the column '<column>' is not valid	The meta-columns RTIDDS_DOMAIN_ID and RTIRTC_REMOTE must be added to tables that the user creates and wished to connect to via *Database Integration Service*. They will be automatically if the *Database Integration Service* daemon creates the table. If the user creates the table and adds the two columns, they must be of type INTEGER. This message is produced if these columns exist and are of the wrong type.
11	Error publishing record/instance	The *Database Integration Service* daemon had a problem publishing a table change as a Topic.
12	Error disposing record/instance	The *Database Integration Service* daemon had a problem disposing of an instance of Topic when the user deleted a row in a table.
13	Error initializing <module> module	The *Database Integration Service* daemon had problems initializing an internal code module.
14	The definition of environment variable '%s' is required	
15	Error opening the database connection associated to the DSN '<DSN>'	
16	Error enabling database log	
17	<string> too long (maximum length: <length>)	
18	Error creating connection to database log	
19	The value of the column 'column' in the table '<meta-table>' is not valid	
20	Error generating '<type>' SQL statement string Error generating SQL statement string	The *Database Integration Service* daemon had a problem in generating the string for preparing or executing a SQL statement.
21	Error skipping parameter for the field '<column>'	
22	Error binding parameter for the column '<column>' Error binding parameters	

Table A.1 Database Integration Service Errors and Warnings

Code	Message	Details
23	The column '<column>' has an unexpected SQL Type	Supported SQL types are: SQL_CHAR SQL_WCHAR SQL_VARCHAR SQL_WVARCHAR SQL_BINARY SQL_VARBINARY SQL_INTEGER SQL_SMALLINT SQL_TINYINT SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE SQL_TIMESTAMP
24	Error opening configuration file 'filename'	
25	Error reading configuration file 'filename'	
26	Error parsing configuration file 'filename'	
27	The maximum length for a <type> field is <length>	
28	Error serializing record	A problem occurred when serializing a table row for publishing as a Topic.
29	Error deserializing RTI DDS sample	A problem occurred when deserializing data received via *Connext DDS* for storing into a table.
30	Error creating key cache	Could not create cache of known instance keys see 4.5.2.1.7 cache_maximum_size, cache_initial_size on page 69.
31	Error inserting key in cache	Problem occurred while storing instance key in cache see 4.5.2.1.7 cache_maximum_size, cache_initial_size on page 69.
32	Null pointer argument	A precondition failed in which a NULL pointer was passed to an internal daemon function.
33	Error reading table '<table>'	
34	The resolution column '<column>' does not exist in the table '<table>'	The RTIDDS_PUBLICATIONS table contained an entry in the 'resolution_column' which does not match the name of an existing column in the corresponding table. See 4.5.1.1.6 resolution_column on page 52.
35	The type of the resolution column '<column>' in the table '<table>' is not valid. The type of the resolution column can be: SQL_INTEGER, SQL_SMALLINT, SQL_BIGINT and SQL_TIMESTAMP	A column specified in the column 'resolution_column' in the RTIDDS_PUBLICATIONS table is not of an acceptable type. See 4.5.1.1.6 resolution_column on page 52.

Table A.1 Database Integration Service Errors and Warnings

Code	Message	Details
36	Error gathering instance information	Error gathering *Connext DDS* instance information through the execution of the associated SELECT statement
37	Invalid metatable schema	The schema of the metatables is not valid. It is possible that those tables were created with a previous version of *Database Integration Service*.
38	Error deleting key from cache	
39	Error deleting a row from '<table>'	There was a problem deleting a row from a user data table.
41	Error creating publication/subscription for the '<table>' without primary key.	The user tried to create a publication/subscription for a table without a primary key.

43	Key not supported	Non-primitive IDL keys are not supported. When *Database Integration Service* tries to create a table with complex keys, it will report this error message.
		Example with supported keys:
		`struct SupportedKeysSt {`
		`  string id_str; //@key`
		`  long id_long; //@key`
		`  short id_short; //@key`
		`};`
		Example with unsupported keys:
		`struct KeySt {`
		`  long id_long;`
		`}`
		`struct NonSupportedKeysSt {`
		`  KeySt id_st; //@key`
		`}`

44	Error creating subscriber state queue	
45	Error updating subscription state	
46	Error creating '<object>'	
47	Path too long	The path to the configuration file is too long.
48	Error creating database publication cache	The *Database Integration Service* daemon had a problem creating the publication database cache.
49	Error adding record to publication cache	The *Database Integration Service* daemon had a problem adding a new record to the publication database cache.
50	Column name length exceeds maximum length	The maximum length of a column name in *Database Integration Service* is 30 characters. To control the length of a column name, use the configuration tags <idl_member_prefix_max_length> and <idl_member_suffix_max_length> under <database_mapping_options>. See 4.4.4.2 Database Mapping Options on page 34.

Table A.1 Database Integration Service Errors and Warnings

Code	Message	Details
1024 - 2047 *Connext DDS*-related errors These messages are produced through the interaction of the *Database Integration Service* daemon with *Connext DDS*. More information on each error can be found by examining the native *Connext DDS* errors codes that will be logged with these messages.		
1024	<message>	General *Connext DDS* error message.
1025	Error getting <entity> default QoS	
1026	Error getting <entity> QoS	
1027	Error setting <entity> QoS	
1028	Error creating <entity>	
1029	Error getting <entity>	
1030	Error enabling <entity>	
1031	Error cloning type code	
1032	Error reading RTI DDS samples	
1033	Error setting <entity> user data	
1034	Error disposing RTI DDS instance	
1035	Error unregistering RTI DDS instance	
1036	Error writing RTI DDS sample	
1037	Error ignoring <entity>	
1038	Error creating <waitset type> waitset	
1039	Error waiting in <waitset type> waitset	
1040	Error getting builtin transport property	
1041	Error setting builtin transport property	
1042	Error creating <waitset type> guard condition	
1043	Error attaching condition	
1044	Error registering type '<type>'	
1045	Error taking REDA buffer	
1046	Error creating mutex	
1047	Error creating <thread> thread	

Table A.1 Database Integration Service Errors and Warnings

Code	Message	Details
1048	Error creating REDA fast buffer	
1049	Error taking semaphore	
1050	Error giving semaphore	
1051	Error creating worker factory	
1052	Error creating worker	
1053	Error creating clock	
1054	Error creating event manager	
1055	Error creating timer	
1056	Error posting event	
1057	Error getting time	
1058	Error creating semaphore	
1059	Error loading DDS XML Profile	
1060	Error getting TypeCode	
1061	Error cloning TypeCode	
1062	Error parsing TypeCode	
1063	Error creating TypeCode	
2048 - 4095 ODBC-related errors These message are produced through the interaction of the *Database Integration Service* daemon with the database through the ODBC driver. More information on each error can be found by examining the native ODBC errors codes that will be logged with these messages.		
2048	<message> <message>: <ODBC driver error message>	General ODBC error message.
4096 - 8191 DBMS Log Connection-related errors These messages are produced through the interaction of the *Database Integration Service* daemon with the *Connext DDS*. More information on each error can be found by examining the native *Connext DDS* errors codes that will be logged with these messages.		
4096	<message>	General DBMS log connection error message.

Table A.1 Database Integration Service Errors and Warnings

Code	Message	Details
8192 - 16383 OS-related errors These messages are produced through the interaction of the *Database Integration Service* daemon with the operating system. More information on each error can be found by examining the native OS errors codes that will be logged with these messages.		
8192	<message>	General OS error message.
8193	Error handling OS signals	
8194	Unable to set signal handler for <signal>	
8195	Error getting the host name	
8196	Error allocating memory for <object>	
16384+ Warning messages These are warning messages that may be logged by the *Database Integration Service* daemon.		
16384	Timestamps prior to '1970-01-01 00:00:00.00' cannot be used for conflict resolution. The *Database Integration Service* daemon will always use '1970-01-01 00:00:00,00' as the timestamp for those cases	
16385	The Timestamp value of the resolution column is NULL. The *Database Integration Service* daemon will use the value '1970-01-01 00:00:00,00'.	
16387	IDL member identifier collision	The prefix/suffix-based name associated with member *A* in IDL type *T* collides with the name of another member inside the same type. *Database Integration Service* will resolve the conflict using an index.
16388	Invalid configuration parameter	
16389	Ignored QoS value	A QoS value has been ignored by *Database Integration Service*.
16392	Dynamic loading of monitoring library is not supported	*RTI Monitoring Library* is statically linked. In *Database Integration Service* there is no need to load this library dynamically.
32769	Requested incompatible QoS	The QoS of a *Database Integration Service* subscription is incompatible with the QoS of a *Connext DDS* publication. The name of the policy that is incompatible is shown in this warning message.

Table A.1 Database Integration Service Errors and Warnings

Code	Message	Details
32770	Offered incompatible QoS	The QoS of a *Database Integration Service* publication is incompatible with the QoS of a *Connext DDS* subscription. The name of the policy that is incompatible is shown in this warning message.
32771	Sample lost message	A *Database Integration Service* subscription lost a sample.
32772	Sample rejected message	A *Database Integration Service* subscription rejected a sample.

Appendix B Database Limits

The maximum number of columns is limited by the underlying database product. The maximum length of a column is independent of the database and it is limited to 30 characters. Table B.1 Real-Time Connect Database Limits notes the database limits of *Database Integration Service* (referred to as DIS in the table).

Table B.1 Real-Time Connect Database Limits

	MySQL 5.7	PostgreSQL
Maximum number of columns	4096, although the effective maximum may be considerably smaller, see B.1 Maximum Columns for MySQL on the next page	250-1600, depending on column types
Maximum column-name length (characters)	30 [a]	30 (bytes)[b]
CLOB/BLOB support	Not supported by DIS Maximum record size is 65535 bytes	Not supported by DIS Maximum record size is 1.6 TB
CHAR maximum size (bytes)	255 (See Note 1 below)	1GB
VARCHAR maximum size (bytes)	65535 (See Note 1 below)	1GB
BINARY maximum size (bytes)	255 (See Note 1 below)	1GB
VARBINARY maximum size (bytes)	65535 (See Note 1 below)	1GB

[a]Limit is imposed by *Database Integration Service* (not MySQL, which allows column names of up to 64 characters).

[b]Limit is imposed by *Database Integration Service* (not PostgreSQL, which allows column names up to 63 bytes)

Table B.1 Real-Time Connect Database Limits

	MySQL 5.7	PostgreSQL
JSON maximum size (bytes)	4194304 by default (JSON support was added in MySQL 5.7)	1 GB
JSONB maximum size (bytes)	Not supported by DIS	256 MB

Note 1: The maximum size of a row in MySQL 5.7 is limited to 65535. For example, you cannot have two fields of type VARCHAR(40000) because the total width of the columns would exceed 65535 bytes. For additional information on this restriction, see http://dev.mysql.com/doc/refman/5.7/en/column-count-limit.html.

Note 2: [varbinary](max) and [varchar](max) are not supported by *Database Integration Service*.

B.1 Maximum Columns for MySQL

The exact limit for the number of columns is driven by two factors:

- The maximum row size for a table, not counting BLOBs, is 65535.
- The maximum size of the meta information (schema) associated with a table is 64 KB. The meta information includes the column names. The longer the column names, the smaller the maximum number of columns.

Table B.2 Max. Number of Columns Used by RTI for Different Column Types and Name Lengths provides information about the maximum number of columns that RTI was able to use for different column-name lengths and column types.

For more information about MySQL restrictions on the number of columns, see http://dev.mysql.com/doc/refman/5.7/en/column-count-limit.html.

Table B.2 Max. Number of Columns Used by RTI for Different Column Types and Name Lengths

Column Type	Column-Name Length (characters)	Maximum Number of Columns
INTEGER	30	1283[a]
DOUBLE		
CHAR (50)		
VARCHAR (50)		
INTEGER	15	1823[b]
DOUBLE		
CHAR (50)	15	1308[c]
VARCHAR (50)	15	1283[d]

[a]Limited by schema size

[b]Limited by schema size

[c]Limited by row size.

[d]Limited by row size.