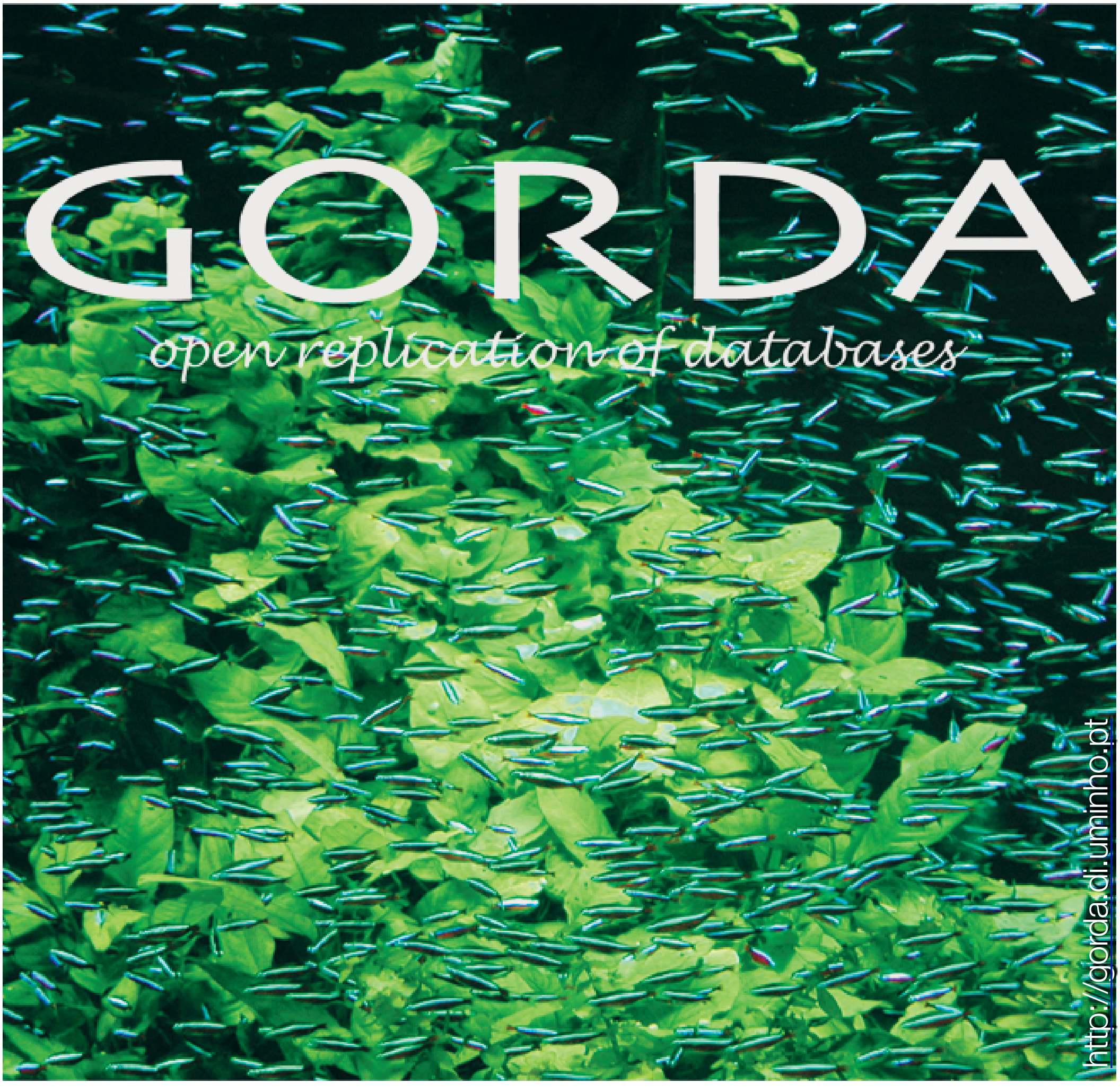


DATABASE SUPPORT FOR REPLICATION

Information Society Technologies

Universidade do Minho

UNIVERSITÀ DELLA SVIZZERA ITALIANA

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

INRIA

continuent
Open. Always Available.

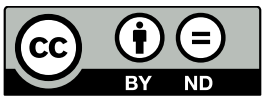
MySQL

Database Support for Replication

Revision 0.2

The GORDA Consortium

June 2007



©2006-2007 The GORDA Consortium. Some rights reserved.

Distribution is allowed according to Creative Commons Attribution-NoDerivs 3.0 license. See Appendix A for details or visit:

http://creativecommons.org/licenses/by-nd/3.0/

Contents

1 Introduction and Background **7**

1.1 Introduction 7

1.2 The GORDA Project 7

1.3 Document Conventions 8

1.3.1 Definitions 8

1.3.2 Formatting Conventions 8

1.4 Open Issues 9

1.5 Contributors 9

1.6 Feedback 9

2 Scope and Requirements **10**

2.1 Goals 10

2.2 Non-Goals 10

2.3 Requirements 10

3 Design **11**

3.1 Approach and Terminology 11

3.2 Overview 11

3.3 Event Handling 12

3.4 Event Concurrency 13

3.5 Commit Order 13

3.6 Context Attachments 14

3.7 Base-level and Meta-level Calls 14

3.8 Notification-Disabled Contexts 15

3.9 Transaction Priority 16

3.10 Exception Handling 16

3.11 Configuration and Bootstrap 16

4 API Description **18**

4.1 Overview 18

4.2 Package gorda.db 23

4.2.1 Interface ConnectionConstant 23

4.2.2 Interface ConnectionContext 24

4.2.3 Interface ConnectionMetaInfo 25

4.2.4 Interface ConnectionProcessor 26

4.2.5 Interface ConnectionShutdownListener 27
4.2.6 Interface ConnectionStartupListener 28
4.2.7 Interface Context 28
4.2.8 Interface ContextReference 30
4.2.9 Interface Database 31
4.2.10 Interface DatabaseConstant 33
4.2.11 Interface DatabaseMetaInfo 35
4.2.12 Interface DatabaseProcessor 35
4.2.13 Interface DatabaseShutdownListener 36
4.2.14 Interface DatabaseStartupListener 37
4.2.15 Interface Dbms 37
4.2.16 Interface DbmsConstant 38
4.2.17 Interface DbmsMetaInfo 39
4.2.18 Interface DbmsProcessor 42
4.2.19 Interface DbmsShutdownListener 43
4.2.20 Interface DbmsStartupListener 43
4.2.21 Interface ExecutionControl 44
4.2.22 Interface PipelineConstant 45
4.2.23 Interface PreparedExecution 46
4.2.24 Interface Request 62
4.2.25 Interface RequestBeginListener 63
4.2.26 Interface RequestCompletionListener 63
4.2.27 Interface RequestConstant 64
4.2.28 Interface RequestProcessor 65
4.2.29 Interface Transaction 66
4.2.30 Interface TransactionBeginListener 68
4.2.31 Interface TransactionCompletionListener 69
4.2.32 Interface TransactionConstant 69
4.2.33 Interface TransactionPrepareListener 72
4.2.34 Interface TransactionProcessor 73
4.2.35 Interface TransactionUpdateListener 75
4.3 Package gorda.db.executor 77
4.3.1 Interface ExecutorStage 77
4.3.2 Interface ObjectSet 78
4.3.3 Interface ObjectSetConstant 79
4.3.4 Interface ObjectSetReadListener 79
4.3.5 Interface ObjectSetWriteListener 80
4.4 Package gorda.db.logminer 81
4.4.1 Interface LoggerObjectSet 81
4.4.2 Interface LoggerObjectSetExecutionListener 81
4.4.3 Interface LogMinerStage 82
4.5 Package gorda.db.parser 83
4.5.1 Interface ParsedStatement 83
4.5.2 Interface ParsedStatementExecutionListener 86
4.5.3 Interface ParserStage 86
4.6 Package gorda.db.receiver 88
4.6.1 Interface ReceiverStage 88
4.6.2 Interface Statement 88
4.6.3 Interface StatementExecutionListener 89

5 Samples **90**

5.1 Query Caching 90

5.2 Streaming 92

5.3 Replication 94

A License **99**

Preface

This document, *Database Support for Replication*, specifies the programming interface to enable pluggable replication protocols and tools in relational database management systems.

This specification is presented in the context of the Java platform. It is however possible to map it to other languages and platforms as it relies on standard concepts and interfaces. For clarity, we make references only to Java transcriptions of such standards.

Revision History

Date	Version	Description
2007-03-31	0.1	Initial public draft
2007-06-18	0.2	Added exception handling section

Who Should Use This Specification

The audience for this document is:

- developers of relational database management systems;
- developers of database replication protocols.

How This Specification Is Organized

Section 1 introduces the interface in the context of the GORDA project as well as document conventions used. Section 2 describes the goals, scope, and requirements of the proposed interface. Section 3 presents the abstract model of transaction processing underlying the interface as well as key design patterns. Section 4 discusses the interface in detail. Finally, Section 5 is a guide to sample code distributed with the interface.

Related Literature

- *On the use of a reflective architecture to augment DBMS by N. Carvalho et al. Technical report FCUL/UMinho, 2007.*
- *The Java™ Language Specification by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1*
- *JSR-54: JDBC™ 3.0 Specification by Jon Ellis and Linda Ho with Maydene Fisher. Sun Microsystems, 2001.*

1 Introduction and Background

1.1 Introduction

This document specifies a programming interface that allows the processing of SQL statements in a relational database management system to be inspected, intercepted, and altered in order to enable replication.

Replication is understood as providing multiple copies of a database, including partial copies, addressing multiple consistency criteria, fault tolerance, and scalability goals. This includes mechanisms sometimes also referred as *clustering* and *synchronization*.

1.2 The GORDA Project

The goal of the GORDA project is to foster database replication as a means to address the challenges of trust, integration, performance, and cost in current database systems underlying the information society. This is to be achieved by standardizing architecture and interfaces, and by sparking their usage with a comprehensive set of components ready to be deployed.

GORDA is supported by the European Community under the Sixth European Union Framework Programme for Research and Technological Development, thematic priority Information Society Technologies, contract number 004758. The consortium is composed by U. Minho, U. della Svizzera Italiana, U. Lisboa, INRIA Rhône-Alpes, Continuent, and MySQL.

More information is available at:

- http://gorda.di.uminho.pt

1.3 Document Conventions

1.3.1 Definitions

This document uses definitions based upon those specified in RFC-2119 (See http://www.ietf.org/). For a better reading experience these terms are written in lowercase.

Table 1: Specification terms.

Term	Definition
MUST	The associated definition is an absolute requirement of this specification.
MUST NOT	The definition is an absolute prohibition of this specification.
SHOULD	Indicates a recommended practice. There may exist valid reasons in particular circumstances to ignore this recommendation, but the full implications must be understood and carefully weighed before choosing a different course.
SHOULD NOT	Indicates a non-recommended practice. There may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
MAY	Indicates that an item is truly optional.

1.3.2 Formatting Conventions

This specification uses the following formatting conventions.

Table 2: Formatting conventions.

Convention	Description
`fixed`	Used in all Java code including keywords, data types, constants, method names, variables, class names, and interface names.
italic	Used for emphasis and to signify the first use of a term.

1.4 Open Issues

- Additional pipeline stages should be considered. Namely, support for federated and distributed databases with a rewriter stage and low level observation of disk I/O with a physical stage.
- The specification of the optimizer stage referred in Figure 1 (see Section 3) has been omitted from the current revision, as existing prototypes have shown limitations in the current proposal.
- Adequation to the version 4.0 of the JDBC specification, namely regarding the exception hierarchy.

1.5 Contributors

- Alfranio Correia Jr., U. Minho
- Nuno Carvalho, U. Lisboa
- Nuno A. Carvalho, U. Minho
- Emmanuel Cecchet, Continuent
- Susana Guedes, U. Lisboa
- Rui Oliveira, U. Minho
- José Pereira, U. Minho
- Luís Rodrigues, U. Lisboa
- Luís Soares, U. Minho
- Ricardo Vilaça, U. Minho

1.6 Feedback

Please send any comments and questions concerning this specification to:

community@gorda.di.uminho.pt

2 Scope and Requirements

2.1 Goals

Support for multiple replication techniques. The specification aims at enabling the use of the multiple replication techniques found in the literature, encompassing asynchronous and synchronous replication, conservative and optimistic execution, total and partial replication.

Portability of replication protocols. The major goal of the specification is to allow replication protocols to be reused with multiple database management systems.

Multiple implementation strategies. The specification aims at allowing multiple implementation strategies, namely, within the database server itself or as a middleware wrapper.

Performance. Although shielding the developer from database server internals, the interface must allow efficient implementations. For instance, by not forcing multiple data conversion steps or by imposing overly restrictive concurrency models.

Compatibility with existing interfaces and idioms. The specification builds on existing interfaces and idioms, thus making it immediately familiar to database developers.

2.2 Non-Goals

Replication protocols. The specification does not include any specific replication protocol, thus omitting all issues related to consistency criteria and update mechanisms.

Communication protocols. The specification does not specify interfaces for communication protocols to disseminate updates.

Configuration and management. The specification does not specify interfaces to bootstrap replication protocols or to manage them while running.

2.3 Requirements

Java Standard Edition platform. All interfaces use the Java language and make use of the standard `java.sql` and `javax.sql` packages. A Java runtime is thus required to deploy replicated database management systems based on the specification.

Server-side JDBC. To support direct base-level to meta-level calls and transparent modification of base-level requests as described in Section 3.7, the database server must provide a Server-side JDBC interface. This is widely available in database management systems supporting Java stored procedures.

3 Design

3.1 Approach and Terminology

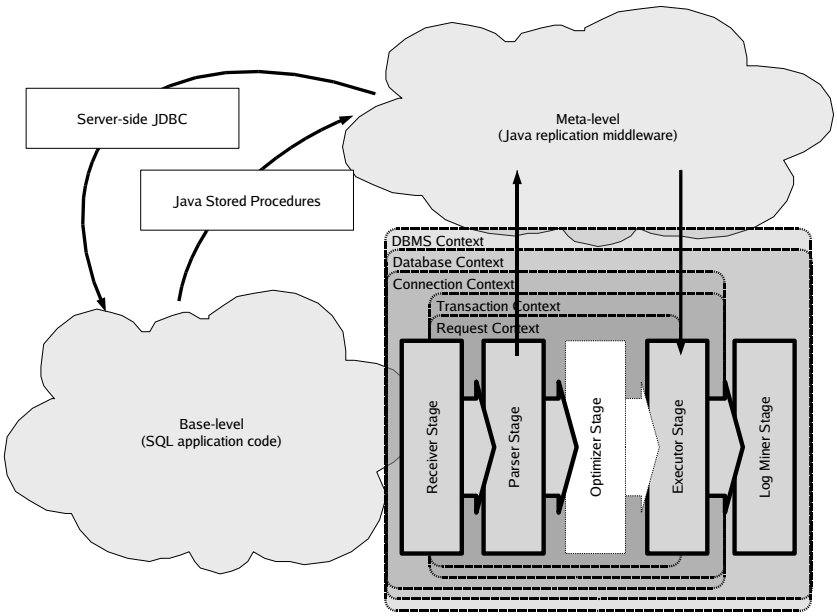
This specification is based on a reflective model of transaction processing. The execution of SQL code by the *database server* is abstracted as a pipeline that performs a number of processing steps.

According to the usual naming in reflective systems, the *base-level* denotes SQL code, as issued by application programs. The *meta-level* denotes add-on middleware that observes and modifies the processing of base-level code.

Each *stage* of the pipeline produces an intermediate data structure that can be inspected and modified. Meta level code can register *event handlers* to be notified when a stage has completed. It can also control when the next stage is started. Related event notifications reference a common *context* object, describing their relation.

3.2 Overview

Figure 1: Abstract transaction processing model.



As shown in Figure 1, the specification abstracts SQL processing by the database server as receiving, parsing, optimizing, and executing statements. The resulting transactional log can then be observed asynchronously.

Multiple statements can be provided in a single request. Therefore, resulting event notifications will reference a common request context. Likewise, several requests can be issued in the context of a transaction. Transactions execute in the context of a client connection. Client connections are established to a specific database in a database management system.

Base-level (*i.e.*, SQL application code) calls into meta-level implicitly, as events are triggered when it traverses the pipeline. Meta-level (*i.e.*, Java replication middleware) influences base-level by interacting with data structures within the pipeline as depicted in Figure 1.

It is however possible that SQL code calls directly into meta-level by means of custom Java stored procedures. It is also possible that meta-level issues SQL statements by means of Server-side JDBC connections.

3.3 Event Handling

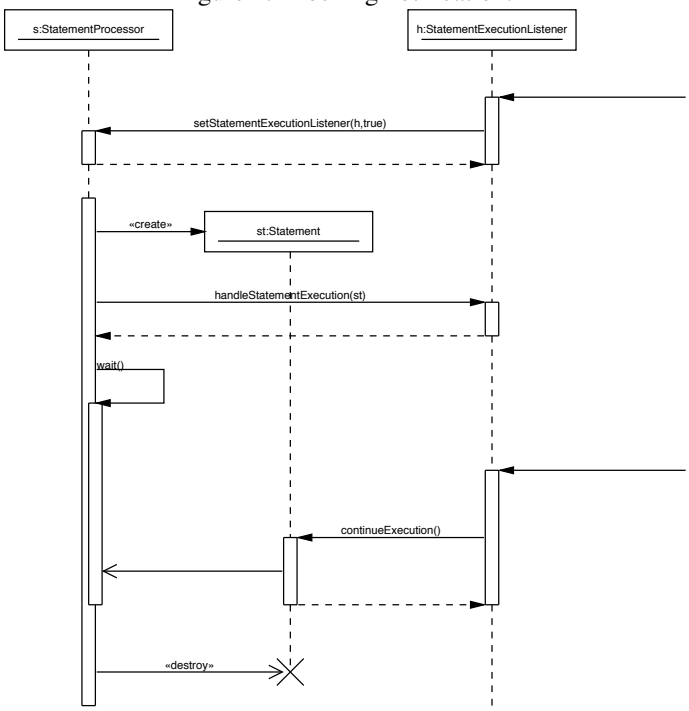
Meta-level code must register event-handlers to intercept the flow of data-structures within the pipeline. An event handler can be set in two different modes: *blocking* and *non-blocking*. This is chosen at run time by specifying a boolean parameter when setting the handler.

Blocking mode

When a handler is set in blocking mode, the database server must suspend the current activity until both the event handler has returned and the continue or cancel methods have been invoked in the event object. The meta-level code can do it in any order.

Figure 2 shows an example of a statement handler being set in blocking mode. Execution is suspended until continue is invoked after waiting for an external event.

Figure 2: Blocking notification.



Non-blocking mode

When a handler is set in non-blocking mode, the database server may suspend the current activity until the event handler has returned. Meta-level code must not invoke continue or cancel methods on the event.
Figure 3 shows an example of a statement handler being set in non-blocking mode. Execution is suspended while calling the handler and resumes right after the handler returns.

Figure 3: Non-blocking notification.

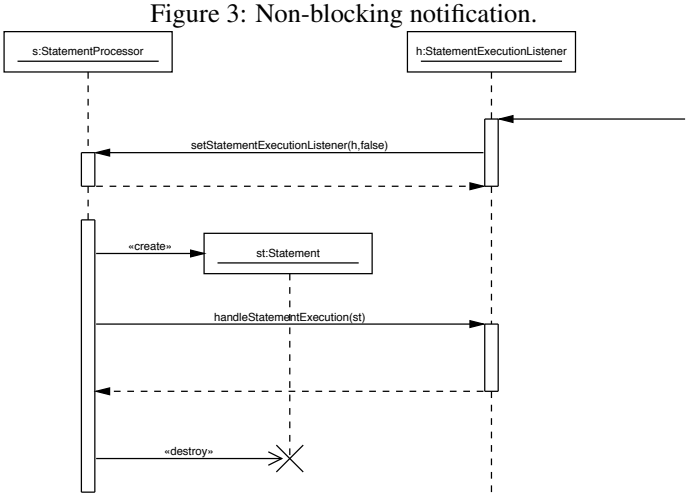


Figure 4 shows an example of a database server that spawns a separate thread for asynchronous notifications. This means that the meta-level handler must not influence the base-level and assume any synchronization between separate asynchronous events. The former point can be achieved by either creating copies of the objects on which the event occurs, thus ensuring that one can access them, without any concerning on object life cycles; or only enabling access to static information or identification data (*e.g.*, transaction identification, request identification).
Methods to continue and cancel execution are available in interface `ExecutionControl`, as shown in Section 4.2.21, and all its sub-interfaces.

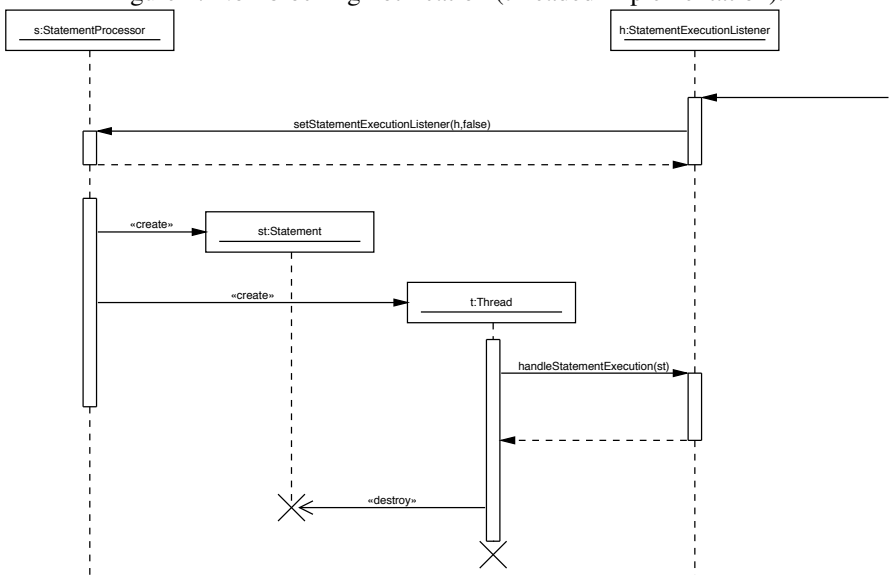
3.4 Event Concurrency

The implementation may invoke multiple event-handlers concurrently, even if registered in blocking mode, unless they depend on each other. It is up to the meta-level code to handle synchronization where required.
Dependency relations exist between events in nested contexts. Namely, events are triggered in a context only when all outer contexts are in the `UP` or `ACTIVE` state. This means that `STARTING` events are notified first for outer contexts and `CLOSED` events first for inner contexts.

3.5 Commit Order

Commit order is determined by the order by which meta-level code invokes `Transaction.continueExecution()` upon a

Figure 4: Non-blocking notification (threaded implementation).



`TransactionConstant.TRANSACTION_COMMITTING` event. The meta-level code must ensure that no concurrent invocations of such method exist within the same database context.

When no blocking event-handler is registered (*i.e.*, no event-handler at all or only a non-blocking event-handler), the commit order is unspecified.

3.6 Context Attachments

Context interfaces allow application specific state to be attached and recovered later when handling different events. The database server must therefore associate the reference with the context and return it in future invocations on the same context. This is similar to what can be done with `java.nio.SelectionKey` in the standard Java library.

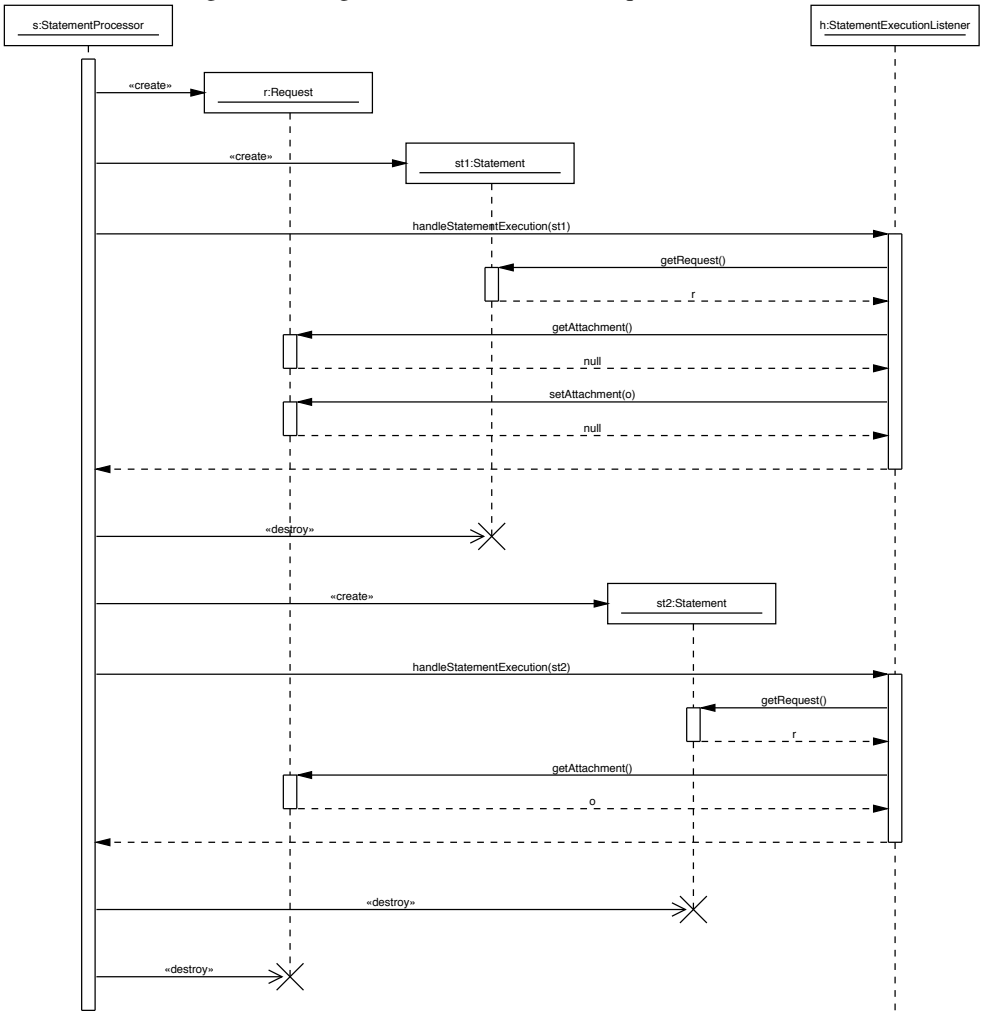
Figure 5 shows two different statements being handled within the same request context. Upon the first invocation, the database server must return `null`. After an object has been attached, it must be kept and returned later. An attachment is removed by setting it to `null`. See interface `Context` in Section 4.2.7.

3.7 Base-level and Meta-level Calls

A direct call to meta-level code may be forced by the application programmer by registering it as a native procedure and then using the `CALL` SQL statement. This causes a call to the meta-level code to be issued from the base-level code within the *Executor Stage*. The target procedure can then retrieve a reference to the enclosing *Request Context* and thus to all relevant meta-interfaces (see Figure 6).

Meta-level code can callback into base-level in two different situations. The first is within a direct call from base-level to issue statements in an existing enclosing request

Figure 5: Using an attachment to store request state.



context. This can be achieved using the Server-side JDBC interface by looking up the `jdbc:default:connection` driver, as is usually done in Java procedures. The second option is to use the enclosing *Database Context* to open a new base-level connection to the database.

Transactions issued at the meta-level using client interfaces must be signaled by invoking the following SQL command:

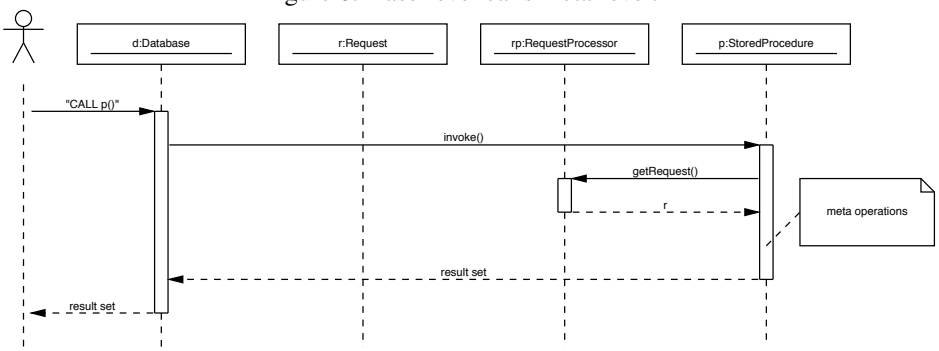
```
SET TRANSACTION AS MASTER
```

The database server must not reflect any further events within the corresponding *Connection Context* to avoid recursion.

3.8 Notification-Disabled Contexts

Reflection can also be disabled on a case-by-case basis by invoking an operation on context meta-objects. Therefore, meta-level code can disable reflection for a given

Figure 6: Base-level calls meta-level.



request, a transaction, a specific connection, a database or even an entire database management system. See interface `Context` in Section 4.2.7.

3.9 Transaction Priority

Base-level calls issued by meta-level code interact with regular transaction processing regarding concurrency control, namely, how are conflicts that require rollback are resolved. This happens in multi-version concurrency control where the first committer wins or, regardless of concurrency control strategy, whenever resolving deadlocks.

A compliant implementation must ensure that transactions issued at meta-level do not abort in face of conflicts with regular base-level transactions.

3.10 Exception Handling

The handling of base-level exceptions within the DBMS is not changed, other than issuing notifications to the meta-level when transactions are aborted or connections closed. The DBMS must react to unhandled exceptions at the meta-level within a transaction context by aborting the encolsing transactions. Other exceptions within the scope of a connection context may close the client connection. Other exceptions must be handled by leaving the database in a panic mode, thus requiring external intervention to repair the system.

Exceptions during meta-level to base-level calls need additional should be handled as meta-level errors to avoid disseminating errors inside the database while executing the base-level code. For instance, while a transaction is committing, meta-level code might need to execute additional statements to keep track of custom meta-information on the transaction before proceeding, and this action might cause errors due to deadlock problems or low amount of resources.

3.11 Configuration and Bootstrap

Configuration of meta-level is out of the scope of this specification. It is thus implementation dependent how such code is loaded. The implementation dependent loader

must however provide references to singleton objects that provide access to the interface. In detail, it must provide references to context singletons, *i.e.*, interfaces named with a `Processor` suffix.

For each implemented and active stage of the pipeline, the implementation must provide its respective singleton object, *i.e.*, interfaces named with the `Stage` suffix.

4 API Description

4.1 Overview

The main part of the specification is contained in package `gorda.db` described in Section 4.2, including all contexts and related interfaces. A diagram outlining the relations between individual interfaces is shown in Figure 7. The database server must fully implement all these interfaces. The life-cycle of each context is shown in Figure 8. Events triggered upon state change are the main entry point to observing the database server.

The rest of the specification corresponds to the pipeline stages, described in Sections 4.3 to 4.6. The database server may implement only some of them. For each implemented stage, the database server must however be complete. Diagrams outlining the required interfaces for each stage are shown in Figures 9 and 10. All data elements share the same life-cycle, shown in Figure 11.

Figure 7: Context interfaces.

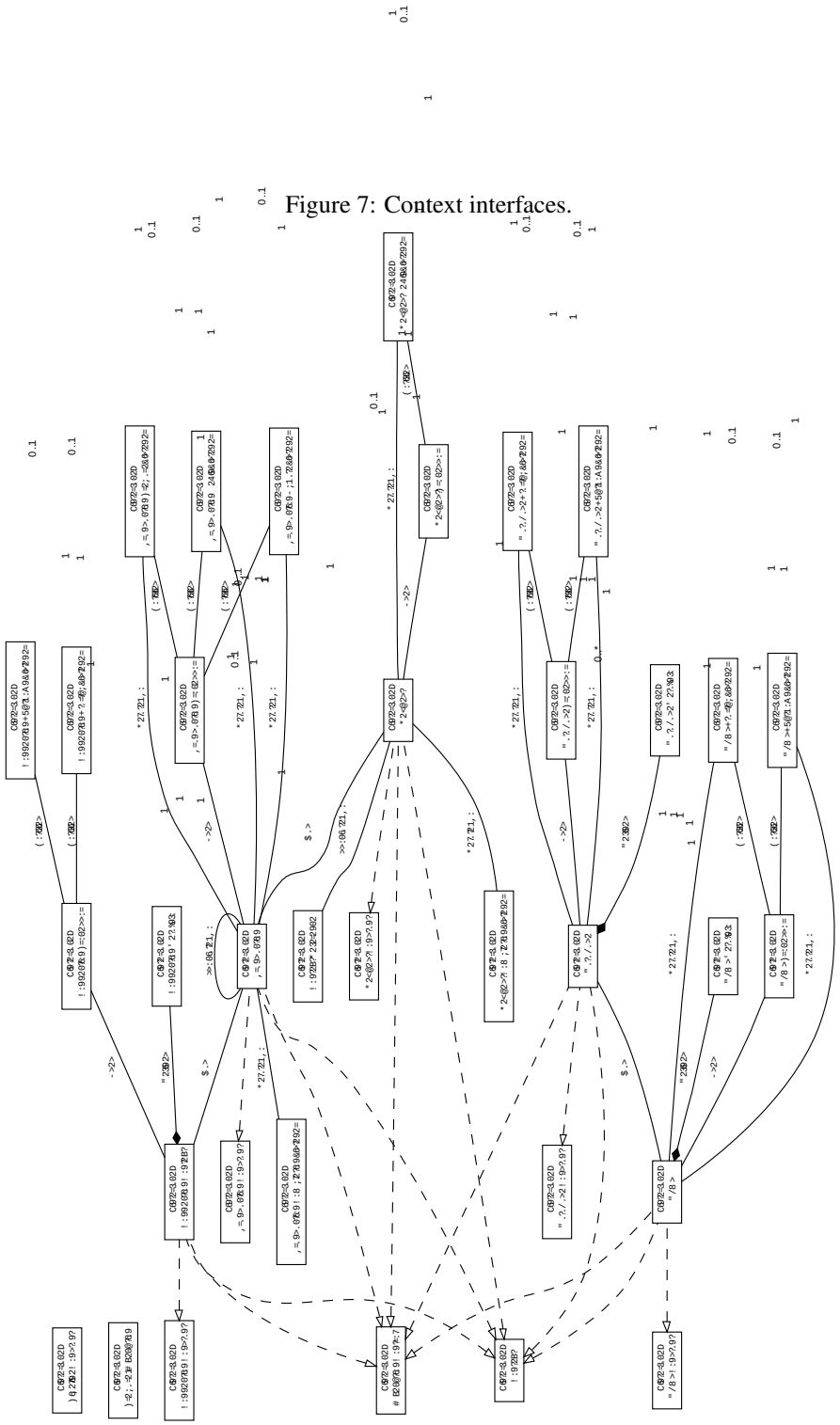
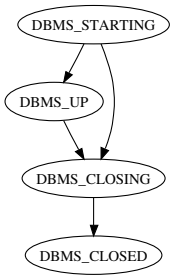
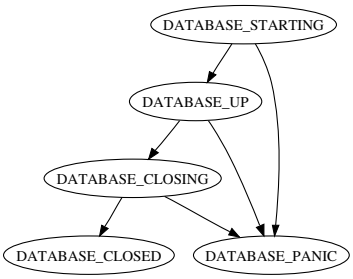


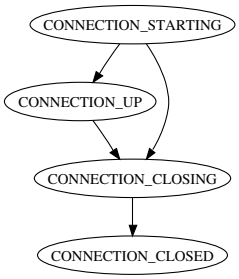
Figure 8: Context life-cycles.



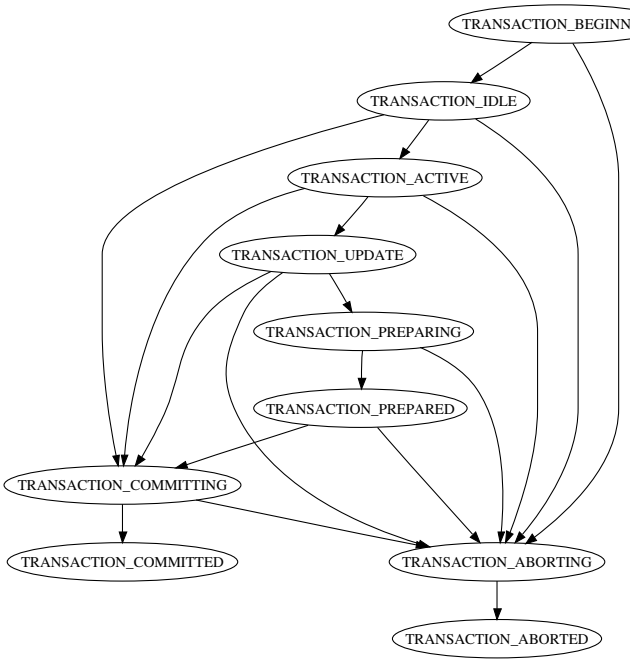
(a) DBMS.



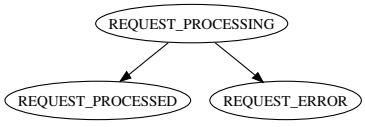
(b) Database.



(c) Connection.

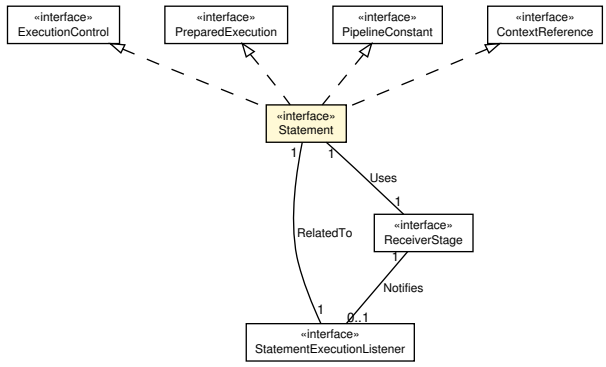


(d) Transaction.

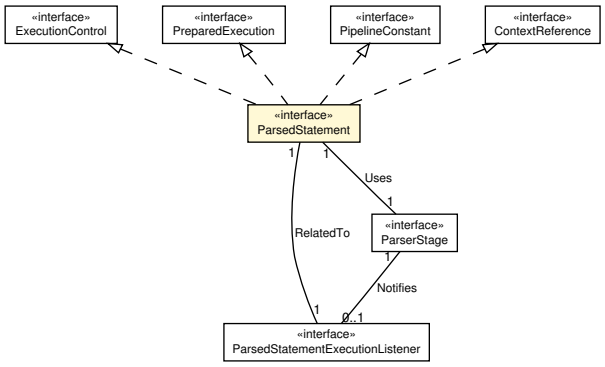


(e) Request.

Figure 9: Stage interfaces.

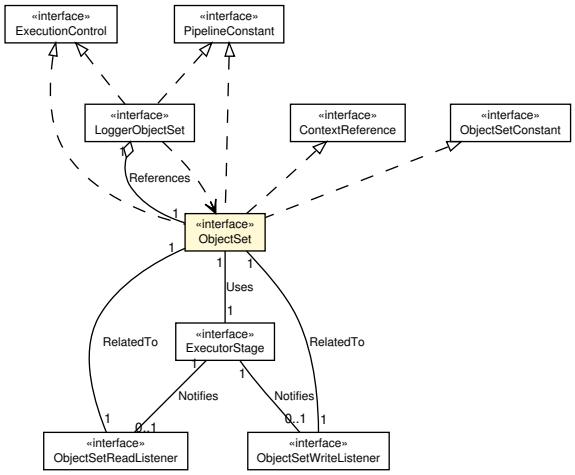


(a) Receiver stage.

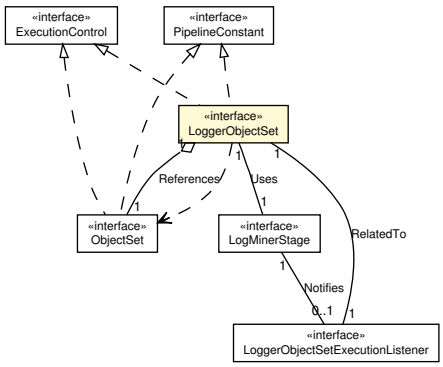


(b) Parser stage.

Figure 10: Stage interfaces (cont).

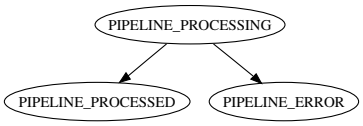


(a) Executor stage.



(b) Log miner stage.

Figure 11: Stage life-cycle.



4.2 Package gorda.db

This package provides access to the DBMS, database, connection, transaction and request contexts.

The DBMS context aims at providing access to server startup and shutdown events, as well as to system configuration information.

A database context holds shared state between multiple connections and provides access to a Server-side JDBC driver and full image backup and restore. In addition to that, it provides events related with database startup and shutdown, as well as enumerating active client connections.

A connection context holds shared state between multiple transactions and provides access to events related with connection establishment and tear down, as well as meta information associated with a client.

A transaction context holds shared state between multiple requests issued on behalf of a transaction. Its main goal is to allow events related to transaction startup, commit and rollback to be observed and validated. This is key to synchronous replication protocols as propagation may be performed before allowing commit to be confirmed back to clients. It is also key to certification based replication protocols, by allowing transactions to be aborted after failing certification.

A request holds shared state between multiple statements contained in a single client request and thus allows grouping of multiple statements in a single client interaction. In addition to that, it identifies the boundaries of a client request by notifying events related with its beginning and completion.

4.2.1 Interface ConnectionConstant

Defines states and constants used by `Connection` (in 4.2.2, page 24).

Declaration `public interface ConnectionConstant`

All known subinterfaces ConnectionContext (in 4.2.2, page 24)

Fields

- `int CONNECTION_STARTING`
  - Defines that a connection is starting up.
    This is the first state and identifies that a connection is being made.
    It must be notified and one must guarantee that access to `Connection`'s meta information and methods is possible.
    In this state, a connection may already be established but the control is not returned to the client, which means that requests cannot be sent.
- `int CONNECTION_UP`

  - Defines that a connection is up.

    This is the second state and identifies that a connection is already established.

    It must be notified and one must guarantee that access to `Connection`'s meta information and methods is possible.

- `int CONNECTION_CLOSING`

  - Defines that a connection is closing.

    This is the third state and identifies that a connection is being closed. Any problem during or after startup must bring a connection to this state.

    It must be notified and one must guarantee that access to `Connection`'s meta information and methods is possible.

    It is worth noticing that it is not possible to cancel this event as it is a transition to the final state.

- `int CONNECTION_CLOSED`

  - Defines that a connection is closed.

    This is the fourth and final state of a connection and identifies that it is closed.

    There is no obligation of notifying this information. However, if one decides to do so, one must guarantee that access to at least a connection identification is possible. Every meta information and method that are not available must throw an exception.

    It is worth noticing that it is not possible to cancel this event as it is the final state.

4.2.2 Interface ConnectionContext

Reflects a client connection. This interface is named `ConnectionContext` and not simply `Connection`, in order to avoid misunderstands between a **reflected connection** and a **JDBC connection**.

Declaration `public interface ConnectionContext`
extends `ConnectionConstant, Context, ExecutionControl`

Methods

- `ConnectionMetaInfo getConnectionMetaInfo()`
  `throws java.sql.SQLException`

  - **Description**

    Returns the connection meta information.

  - **Returns** – Connection meta information.
  - **Throws**

    * `java.sql.SQLException` – If a database access error occurs.

- `ConnectionProcessor getConnectionProcessor()`
  - **Description**
    Returns a reference to the connection processor.
  - **Returns** – A reference to the connection processor.
- `Database getDatabase()`
  - **Description**
    Returns a reference to the database object.
    There is no need of returning a copy of this object as one must do when handling the method `getTransaction` (in 4.2.2, page 25). Assuming a blocking notification, the database context must be accessible by means of a connection.
  - **Returns** – A reference to the database object.
- `Transaction getTransaction()`
  - **Description**
    Returns a copy of the active transaction object.
    To avoid synchronization problems, one must do exactly what follows:
    * Returning a copy of the object and throwing an exception if any method that attempts to change its state is called.
  - **Returns** – A copy of the active transaction object, if there is any, otherwise `null`.

4.2.3 Interface ConnectionMetaInfo

Defines connection meta information.

Declaration `public interface ConnectionMetaInfo`

Methods

- `java.lang.Object getCharacterSetInformation()`
  `throws java.sql.SQLException`
  - **Description**
    Returns character set information.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.
- `java.lang.String getUserId()`
  `throws java.sql.SQLException`
  - **Description**
    Returns user identification.
  - **Returns** – User identification.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.

4.2.4 Interface ConnectionProcessor

Handles listener registration for connection events and has a connection repository.

Declaration `public interface ConnectionProcessor`

Methods

- `ConnectionContext getConnection(java.lang.String connectionId)`
  - **Description**
    Returns a copy of the reflected connection object with the given id.
    To avoid synchronization problems, one must do exactly what follows:
    * Returning a copy of the object and throwing an exception if any method that attempts to change its state is called.
  - **Parameters**
    * `connectionId` – The connection identification.
  - **Returns** – A copy of the reflected connection object with the given id, if there is any, `null` otherwise.
- `void setConnectionShutdownListener(ConnectionShutdownListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified upon connection shutdown.
    Subsequent notifications, with respect to the connection and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30) or `Connection.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    * `listener` – The listener that handles connection shutdown events.
    * `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    * `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    * `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)
- `void setConnectionStartupListener(ConnectionStartupListener listener, boolean wait)`

– **Description**

Registers a listener that must be notified upon connection startup.

Subsequent notifications, with respect to the connection and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30).

– **Parameters**

* `listener` – The listener that handles connection startup events.
* `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.

– **See also**

* `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
* `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.2.5 Interface ConnectionShutdownListener

Defines the listener that will be notified whenever a connection is shutting down.

Declaration `public interface ConnectionShutdownListener`

Methods

- `void handleConnectionShutdown(ConnectionContext connection)`

  – **Description**

  Is called whenever the listener is registered to receive connection shutdown events.

  If the wait flag is set to `true` at registration time (see `setConnectionShutdownListener` (in 4.2.4, page 26)), then this method implementation must call `continueExecution` (in 4.2.21, page 45).

  If the wait flag is set to `false` at registration time, then this method must be run in parallel with the connection shutdown.

  If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.

  A connection must have one of the following states when receiving a notification: `CONNECTION_CLOSING` (in 4.2.1, page 24) or `CONNECTION_CLOSED` (in 4.2.1, page 24).

  – **Parameters**

  * `connection` – The connection on which the event occurs.

4.2.6 Interface ConnectionStartupListener

Defines the listener that will be notified whenever a connection is made.

Declaration `public interface ConnectionStartupListener`

Methods

- `void handleConnectionStartup(ConnectionContext connection)`
  - **Description**

    Is called whenever the listener is registered to receive connection startup events.

    If the wait flag is set to `true` at registration time (see `setConnectionStartupListener` (in 4.2.4, page 26)), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).

    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the connection startup.

    If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.

    A connection must have one of the following states when receiving a notification: `CONNECTION_STARTING` (in 4.2.1, page 23) or `CONNECTION_UP` (in 4.2.1, page 23).
  - **Parameters**
    - `connection` – The connection on which the event occurs.

4.2.7 Interface Context

This interface defines information common to every context in the GORDA API.

Each context must be capable of maintaining a reference to an object, also named attachment. The attachment may be set using the `setAttachment` (in 4.2.7, page 29) method and retrieved using the `getAttachment` (in 4.2.7, page 29) method.

There are several sub-interfaces that define the API for the GORDA API defined contexts:

- `Dbms` (in 4.2.15, page 37);
- `Database` (in 4.2.9, page 31);
- `Connection` (in 4.2.2, page 24);
- `Transaction` (in 4.2.29, page 66);
- `Request` (in 4.2.24, page 62);

Additionally, associated with each context there are several processing stages. These stages are based on the classic stages already proposed by previous published works on the subject. The result of each stage is mapped into a set of interfaces that represent the outcome of the processing stage.

The result of each processing stage must be one instance of the following interfaces:

- `Statement` (in 4.6.2, page 88)
- `ObjectSet` (in 4.3.2, page 78)
- `ParsedStatement` (in 4.5.1, page 83)
- `LoggerObjectSet` (in 4.4.1, page 81)
- `ExecutionPlan`

.

Declaration `public interface Context`

All known subinterfaces ConnectionContext (in 4.2.2, page 24), Database (in 4.2.9, page 31), Dbms (in 4.2.15, page 37), Request (in 4.2.24, page 62), Transaction (in 4.2.29, page 66)

Methods

- `java.lang.Object getAttachment()`
  - **Description**
    Returns the current attachment. This method must not remove the attachment.
  - **Returns** – The current attachment, or `null`, if there is no attachment.
- `int getContextState()`
  - **Description**
    Retrieves the current context state. Every context has an associated state:
    * `DbmsConstant` (in 4.2.16, page 38);
    * `DatabaseConstant` (in 4.2.10, page 33);
    * `ConnectionConstant` (in 4.2.1, page 23);
    * `TransactionConstant` (in 4.2.32, page 69);
    * `RequestConstant` (in 4.2.27, page 64);
  - **Returns** – The current context state.
- `java.lang.String getId()`
  - **Description**
    Returns a context identification.
  - **Returns** – Context identification.
- `java.lang.Object setAttachment(java.lang.Object obj)`

- **Description**
  Attaches a reference to the given object to a context. This method provides access to a placeholder in which a programmer may set any kind of object.
  A reference that has been attached may be retrieved later via the `getAttachment` (in 4.2.7, page 29) method. There must only be one reference attached at a time.
  Calling this method must discard the current attached reference. In order to discard the current reference, one must call this method with `null` as the parameter.
- **Parameters**
  * `obj` – The object whose reference must be attached, which may be `null`.
- **Returns** – The previously attached reference, if any, otherwise `null`.

- `void setNotificationIgnored(boolean isIgnored)`
  - **Description**
    Enables or disables notifications regarding a context.
    This method has a "Cascade Effect", meaning that a listener must not receive any notification regarding a context and inner contexts.
  - **Parameters**
    * `isIgnored` – `true` if notifications must be ignored, `false` otherwise.

4.2.8 Interface ContextReference

Retrieves a reference to the enclosing context.

Declaration `public interface ContextReference`

All known subinterfaces ObjectSet (in 4.3.2, page 78), ParsedStatement (in 4.5.1, page 83), Statement (in 4.6.2, page 88)

Methods

- `Request getRequest()`
  - **Description**
    Returns a reference to the request context.
  - **Returns** – A reference to the request context.

- `int getState()`
  - **Description**
    Retrieves the current state of a stage of the pipeline. Every stage has a common or specific set of constants based on which their states are defined:
    * `PipelineConstant` (in 4.2.22, page 45);
    * `ObjectSetConstant` (in 4.3.3, page 79);
  - **Returns** – The current state of a stage.

4.2.9 Interface Database

Reflects a database or a logical entity that is reflected.

Declaration `public interface Database`
extends `DatabaseConstant, Context, ExecutionControl`

Methods

- `void freeze()`
  - **Description**
    Sets the database in panic mode.
    For instance, this method must be used to freeze a database when it somehow aborts a transaction sent by a metalevel-application and such transaction was not supposed to abort.
    When the database is set to panic mode, then it freezes and only an administrator is able to manually change its state.
- `java.util.Iterator getConnections()`
  - **Description**
    Returns an iterator with a copy of all reflected connection objects opened to access this database.
    To avoid synchronization problems, one must do exactly what follows:
    * Returning a copy of the objects and throwing an exception if any method that attempts to change their state is called.
  - **Returns** – An iterator with a copy of all reflected connection objects.
- `long getCurrentVersion()`
  - **Description**
    Returns the current database version number.
    Every time an update transaction is committed, the implementation increments a counter to register such event.
  - **Returns** – The current database version number.
- `java.io.InputStream getDatabaseImage(java.lang.String tableName)`
  - **Description**
    Retrieves a database image or a table image as an `InputStream`.
    The structure of the image is application dependent.
    If `tableName` is not null, an image is taken from that table. Otherwise an image from the database is taken. If the table does not exist, an exception must be thrown.
  - **Parameters**
    * `tableName` – Defines from which table an image must be taken.

  - **Returns** – A database image as an `InputStream`.

- `DatabaseMetaInfo getDatabaseMetaInfo()`
  `throws java.sql.SQLException`

  - **Description**
    Returns the database meta information.
  - **Returns** – The database meta information.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.

- `DatabaseProcessor getDatabaseProcessor()`

  - **Description**
    Returns a reference to the database processor.
  - **Returns** – A reference to the database processor.

- `int getDatabaseSize()`

  - **Description**
    Calculates the size of the database.
    This provides a mechanism to estimate the size of an image in a unit that is application dependent.
  - **Returns** – The size of the database.

- `javax.sql.DataSource getDataSource()`

  - **Description**
    Returns the datasource.
  - **Returns** – The datasource.

- `Dbms getDbms()`

  - **Description**
    Returns a reference to the DBMS.
    In contrast to `getConnections` (in 4.2.9, page 31), a reference to the DBMS must be returned. In this case, there is no problem as the DBMS context is finished only after stopping all active databases.
  - **Returns** – A reference to the DBMS.

- `long getMinimumVersion()`

  - **Description**
    Returns the version of the oldest active transaction in the database.
    The implementation must check all active transactions in order to find the oldest.
  - **Returns** – The oldest active transaction.

- `java.lang.String getUrl()`

  - **Description**
    Returns the database *Uniform Resource Locator*.
  - **Returns** – The database URL.

- `void increaseVersion(long inc)`
  - **Description**
    Increments the database version by a positive value passed as parameter.
    Every time an update transaction commits, the database version is incremented by one. Sometimes however a transaction is executed on behalf of several other transactions. In such cases, incrementing by one does not reflect the number of implicit transactions committed. For that reason, this method enables developers to define which is the increment.
  - **Parameters**
    * `inc` – A positive value used as increment.

- `void installDatabaseImage(java.io.InputStream databaseImage)`
  - **Description**
    Installs an image as an `InputStream`.
    The database must be in the state `DATABASE_STARTING` or `DATABASE_IN_PANIC`, to be allowed to install images. This requirement ensures that a database is in recovering mode. If a database is in another state, it is not supposed to do so as it may have internal active (`DATABASE_BOOTED` or `DATABASE_CLOSING`) or or may be shutting down (`DATABASE_CLOSED`).
    It is also worth noticing that is possible to install images from tables, but the metalevel-application needs to guarantee that such images are coherent among them in order to bring the database to a consistent state.
  - **Parameters**
    * `databaseImage` – The database or table image as an `InputStream`.

4.2.10 Interface DatabaseConstant

Defines states and constants used by `Database` (in 4.2.9, page 31).

Declaration `public interface DatabaseConstant`

All known subinterfaces Database (in 4.2.9, page 31)

Fields

- `int DATABASE_STARTING`

  - Defines that a database is starting up.
    This is the first state and identifies that a database is starting up.
    It must be notified and one must guarantee that access to `Database`'s meta information and methods is possible.
    In this state, a database allows to carry out recovery routines.

- `int DATABASE_UP`
  - Defines that a database is up.
    This is the second state and identifies that a database executed its recovery routines and is ready to receive client connections.
    It must be notified and one must guarantee that access to `Database`'s meta information and methods is possible.

- `int DATABASE_CLOSING`
  - Defines that a database is shutting down.
    This is the third state and identifies that a database is shutting down.
    It must be notified and one must guarantee that access to at least a database identification is possible. Every meta information and methods that are not available must throw an exception.
    It is worth noticing that it is not possible to cancel this event as it is a transition to a final state.

- `int DATABASE_CLOSED`
  - Defines that a database is shutdown.
    This is the fourth and a final state of a database and identifies that it is shutdown.
    It must be notified and one must guarantee that access to at least a database identification is possible. Every meta information and methods that are not available must throw an exception.
    It is worth noticing that it is not possible to cancel this event as it is a final state.

- `int DATABASE_IN_PANIC`
  - A database is put in this state by calling the method `freeze` (in 4.2.9, page 31) or automatically when something goes wrong. In other words, when something unrecoverable happens.
    For instance, the database is put in this state when:
    * It somehow aborts a transaction sent by a metalevel-application and such transaction was not supposed to abort.
    * During startup it is not brought to a consistent state.
    * Or during shutdown it is not possible for some reason (e.g. log is corrupted) to bring it to a consistent state.

    If the database is set to panic mode, then it freezes and only an administrator is able to manually change its state.
    It must be notified and one must guarantee that access to at least a database identification is possible. Every meta information and methods that are not available must throw an exception.
    It is worth noticing that it is not possible to cancel this event as it is a final state.

4.2.11 Interface DatabaseMetaInfo

Provides access to database meta information. This builds on the `DatabaseMetaData` (*i.e.*, a Server-side JDBC driver) and provides additional methods important to the metalevel development.

Declaration `public interface DatabaseMetaInfo`
extends `java.sql.DatabaseMetaData`

Methods

- `boolean isFrozen()`
  `throws java.sql.SQLException`
  - **Description**
    Returns whether this database is frozen or not.
    If the database is frozen, then only the administrator is able to apply changes and manually redefine its state.
    If the `freeze` (in 4.2.9, page 31) method is called in some point in time, then subsequent calls to the `isFrozen` method must return `true`.
  - **Returns** – `true` if the database is frozen, `false` otherwise.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.
- `boolean isReadOnly()`
  `throws java.sql.SQLException`
  - **Description**
    Returns whether this database is in read-only mode or not.
  - **Returns** – `true` if so; `false` otherwise.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.

4.2.12 Interface DatabaseProcessor

Handles listener registration for database events and has a database repository.

Declaration `public interface DatabaseProcessor`

Methods

- `Database getDatabase(java.lang.String databaseId)`
  - **Description**
    Returns a copy of the database object with the given id.
    To avoid synchronization problems, one must do exactly what follows:

    * Returning a copy of the object and throwing an exception if any method that attempts to change its state is called.
  - **Parameters**
    * `databaseId` – The database identification.
  - **Returns** – A copy of the database object with the given id, if there is any, `null` otherwise.

- `void setDatabaseShutdownListener(DatabaseShutdownListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified upon database shutdown.
    Subsequent notifications, with respect to the database and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30) or `Database.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    * `listener` – The listener that handles database shutdown events.
    * `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    * `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    * `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

- `void setDatabaseStartupListener(DatabaseStartupListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified upon database startup.
    Subsequent notifications, with respect to the database and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30) or `Database.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    * `listener` – The listener that handles database startup events.
    * `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    * `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    * `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.2.13 Interface DatabaseShutdownListener

Defines the listener that will be notified whenever the database is shutting down.

Declaration `public interface DatabaseShutdownListener`

Methods

- `void handleDatabaseShutdown(Database database)`
  - **Description**
    Is to be called whenever there is a listener registered to receive database shutdown events.
    If the wait flag is set to `true` at registration time (see `setDatabaseShutdownListener` (in 4.2.12, page 36)), then this method implementation must call `continueExecution` (in 4.2.21, page 45).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the database shutdown.
    If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.
  - **Parameters**
    * `database` – The database on which the event occurs.

4.2.14 Interface DatabaseStartupListener

Defines the listener that will be notified whenever the database is starting up.

Declaration `public interface DatabaseStartupListener`

Methods

- `void handleDatabaseStartup(Database database)`
  - **Description**
    Is to be called whenever the listener is registered to receive database startup events.
    If the wait flag is set to `true` at registration time (see `setDatabaseStartupListener` (in 4.2.12, page 36)), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the database startup.
    If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.
  - **Parameters**
    * `database` – The database on which the event occurs.

4.2.15 Interface Dbms

Reflects a DBMS.

Declaration `public interface Dbms`
extends `DbmsConstant, Context, ExecutionControl`

Methods

- `java.util.Iterator getDatabases()`
  - **Description**
    Returns an iterator with a copy of all database objects.
    To avoid synchronization problems, one must do exactly what follows:
    * Returning a copy of the objects and throwing an exception if any method that attempts to change their state is called.
  - **Returns** – An iterator with a copy of all database objects.
- `DbmsMetaInfo getDbmsMetaInfo()`
  - **Description**
    Returns the DBMS meta information.
  - **Returns** – The DBMS meta information.
- `DbmsProcessor getDbmsProcessor()`
  - **Description**
    Returns a reference to the DBMS processor.
  - **Returns** – The DBMS processor.

4.2.16 Interface DbmsConstant

Defines states and constants used by `Dbms` (in 4.2.15, page 37).

Declaration `public interface DbmsConstant`

All known subinterfaces Dbms (in 4.2.15, page 37)

Fields

- `int DBMS_STARTING`
  - Defines that a DBMS is starting up.
    This is the first state and identifies that a DBMS is starting up.
    It must be notified and one must guarantee that access to `Dbms`'s meta information and methods is possible.
- `int DBMS_UP`

  - Defines that a DBMS is up.

    This is the second state and identifies that a DBMS is ready to manage databases.

    It must be notified and one must guarantee that access to `Dbms`'s meta information and methods is possible.

- `int DBMS_CLOSING`

  - Defines that a DBMS is shutting down. Any unforeseen event must bring the DBMS to this state.

    This is the third state and identifies that a DBMS is shutting down.

    It must be notified and one must guarantee that access to at least a database identification is possible. Every meta information and methods that are not available must throw an exception.

    It is worth noticing that it is not possible to cancel this event as it is a transition to the final state.

- `int DBMS_CLOSED`

  - Defines that a DBMS is shutdown.

    This is the fourth and final state of a DBMS and identifies that it is shutdown.

    It must be notified and one must guarantee that access to at least a database identification is possible. Every meta information and methods that are not available must throw an exception.

    It is worth noticing that it is not possible to cancel this event as it is a transition to the final state.

4.2.17 Interface DbmsMetaInfo

Provides access to DBMS meta information. This builds on the `DatabaseMetaData` (*i.e.*, a Server-side JDBC driver) and provides additional methods important to the reflection mechanism.

Declaration `public interface DbmsMetaInfo`

Methods

- `int getDbmsMajorVersion()`
  `throws java.sql.SQLException`

  - **Description**

    Returns the DBMS major version number.

  - **Returns** – The DBMS major version number.

  - **Throws**

    * `java.sql.SQLException` – If a database access error occurs.

- `int getDbmsMinorVersion()`
  `throws java.sql.SQLException`
  - **Description**
    Returns the DBMS minor version number.
  - **Returns** – The DBMS minor version number.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.
- `java.lang.String getDbmsProductName()`
  `throws java.sql.SQLException`
  - **Description**
    Returns the name of the Database Management System (DBMS) product.
  - **Returns** – The DBMS product name.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.
- `java.lang.String getDbmsProductVersion()`
  `throws java.sql.SQLException`
  - **Description**
    Returns the DBMS version number.
  - **Returns** – The DBMS version number.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.
- `int getJDBCDriverMajorVersion()`
  - **Description**
    Returns an integer identifying the JDBC (*i.e.*, a Server-side JDBC driver) driver major version.
  - **Returns** – The driver major version.
- `int getJDBCDriverMinorVersion()`
  - **Description**
    Returns the JDBC (*i.e.*, a Server-side JDBC driver) driver minor version.
  - **Returns** – The driver minor version.
- `java.lang.String getJDBCDriverName()`
  `throws java.sql.SQLException`
  - **Description**
    Returns the JDBC (*i.e.*, a Server-side JDBC driver) driver name.
  - **Returns** – The driver name.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.

- `java.lang.String getJDBCDriverVersion()`
  `throws java.sql.SQLException`
  - **Description**
    Returns the JDBC (*i.e.*, a Server-side JDBC driver) driver version.
  - **Returns** – The driver version.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.
- `java.lang.String getURL()`
  `throws java.sql.SQLException`
  - **Description**
    Returns the DBMS *Uniform Resource Locator*.
  - **Returns** – The DBMS Uniform Resource Locator.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs.
- `boolean isExecutorStageImplemented()`
  - **Description**
    Returns the implementation status of the executor stage.
  - **Returns** – `true` if it is implemented, `false` otherwise.
- `boolean isExtractingReadSets()`
  - **Description**
    Returns the implementation status on read sets.
  - **Returns** – `true` if it is implemented, `false` otherwise.
- `boolean isExtractingWriteSets()`
  - **Description**
    Returns the implementation status on write sets.
  - **Returns** – `true` if it is implemented, `false` otherwise.
- `boolean isLogMinerStageImplemented()`
  - **Description**
    Returns the implementation status of the log miner stage.
  - **Returns** – `true` if it is implemented, `false` otherwise.
- `boolean isOptimizerStageImplemented()`
  - **Description**
    Returns the implementation status of the optimizer stage.
  - **Returns** – `true` if it is implemented, `false` otherwise.
- `boolean isParserStageImplemented()`

  - **Description**
    Returns the implementation status of the parser stage.
  - **Returns** – `true` if it is implemented, `false` otherwise.

- `boolean isReceiverStageImplemented()`
  - **Description**
    Returns the implementation status of the receiver stage.
  - **Returns** – `true` if it is implemented, `false` otherwise.

4.2.18 Interface DbmsProcessor

Handles listener registration for DBMS events.

Declaration `public interface DbmsProcessor`

Methods

- `void setDbmsShutdownListener(DbmsShutdownListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified upon DBMS shutdown.
    Subsequent notifications, with respect to the DBMS and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    * `listener` – The listener that handles the DBMS shutdown events.
    * `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    * `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    * `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)
- `void setDbmsStartupListener(DbmsStartupListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified upon DBMS startup.
    Subsequent notifications, with respect to the DBMS and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    * `listener` – The listener that handles the DBMS startup events.

    * `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.

  – **See also**

    * `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    * `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.2.19 Interface DbmsShutdownListener

Defines the listener that will be notified whenever the database is shutting down.

Declaration `public interface DbmsShutdownListener`

Methods

- `void handleDbmsShutdown(Dbms dbms)`

  – **Description**

  Is called whenever the listener is registered to receive DBMS shutdown events.

  If the wait flag is set to `true` at registration time (see `setDbmsShutdownListener` (in 4.2.18, page 42)), then this method implementation must call `continueExecution` (in 4.2.21, page 45).

  If the wait flag is set to `false` at registration time, then this method must be run in parallel with the DBMS shutdown.

  If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.

  – **Parameters**

    * `dbms` – The DBMS on which the event occurs.

4.2.20 Interface DbmsStartupListener

Defines the listener that will be notified whenever the database is starting up.

Declaration `public interface DbmsStartupListener`

Methods

- `void handleDbmsStartup(Dbms dbms)`

  – **Description**

  Is called whenever the listener is registered to receive DBMS startup events.

If the wait flag is set to `true` at registration time (see `setDbmsStartupListener` (in 4.2.18, page 42), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).
If the wait flag is set to `false` at registration time, then this method must be run in parallel with the database startup.
If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.

– **Parameters**
  * `dbms` – The DBMS on which the event occurs.

4.2.21 Interface ExecutionControl

Resumes or cancels execution.
One of the methods provided by this interface must be invoked by a listener when it is registered with the waiting flag set to `true`. Any invalid call characterized by the set of cases that follows must either throw an exception or simply be ignored:

- Calling such methods when a notifier is not blocked. This means that there is no pending notification waiting for a call to proceed.
- As a cololllary of the previous case one has: calling such methods when a listener is registered with the waiting flag set to `false`.
- Some states define a final step of a state machine (e.g., `CONNECTION_CLOSED` (in 4.2.1, page 24)) and do not allow calls to the cancel method to be made.

The choice between ignoring invalid calls or throwing an exception is presented to preserve compatibility with meta-level applications already developed. This was the common practice in the early versisons of this interface.
The descriptions presented in this document however only uses exceptions and it is thoroughly recommend its adoption.

Declaration `public interface ExecutionControl`

All known subinterfaces ConnectionContext (in 4.2.2, page 24), Database (in 4.2.9, page 31), Dbms (in 4.2.15, page 37), Request (in 4.2.24, page 62), Transaction (in 4.2.29, page 66), ObjectSet (in 4.3.2, page 78), LoggerObjectSet (in 4.4.1, page 81), ParsedStatement (in 4.5.1, page 83), Statement (in 4.6.2, page 88)

Methods

- `void cancelExecution()`
  `throws java.sql.SQLException`

  – **Description**
  Cancels execution.
  Some events does not support calling this method and an exception must be thrown.
  Should the listener be registered with the waiting flag set to `false`, then the execution must proceed without waiting for this method to be called.

- `void continueExecution()`
  `throws java.sql.SQLException`

  - **Description**

    Resumes execution.

    If the listener is registered with the waiting flag set to `false`, then the execution must proceed without waiting for this method to be called.

4.2.22 Interface PipelineConstant

Defines states and constants used by the `Statement` (in 4.6.2, page 88), `ParsedStatement` (in 4.5.1, page 83), `ExecutionPlan` , `ObjectSet` (in 4.3.2, page 78) and `LoggerObjectSet` (in 4.4.1, page 81).

The processing state indicates that an object was processed by their respective counterpart in the pipeline and it is about to enter in the next stage:

- The `Statement.PIPELINE_PROCESSING` means that a request was split in different statements and is about to entering into the parser. Right after parsing a statement, a notification may be sent to indicate that it was processed and its state is `Statement.PIPELINE_PROCESSED`.
- The `ParsedStatement.PIPELINE_PROCESSING` means that a parsed statement was produced by the parser and is about to entering into the optimizer. Right after optimizing a parsed statement, a notification may be sent to indicate that it was processed and its state is `ParsedStatement.PIPELINE_PROCESSED`.
- The `ExecutionPlan.PIPELINE_PROCESSING` means that an execution plan was produced by the optimizer and is about to entering into the executor. Right after processing an execution plan, a notification may be sent to indicate that it was processed and its state is `ExecutionPlan.PIPELINE_PROCESSED`.
- The `ObjectSet.PIPELINE_PROCESSING` means that an object set was produced by the executor and is about to be logged in memory (write-ahead logging). Right after logging an object set a notification may be sent to indicate that it was processed and its state is `ObjectSet.PIPELINE_PROCESSED`.
- The `LoggerObjectSet.PIPELINE_PROCESSING` means that a logger object set was created in memory and is about to be written to disk (write-ahead logging). Right after writing it to disk, a notification may be sent to indicate that it was processed and its state is `LoggerObjectSet.PIPELINE_PROCESSED`.

Declaration `public interface PipelineConstant`

All known subinterfaces ObjectSet (in 4.3.2, page 78), LoggerObjectSet (in 4.4.1, page 81), ParsedStatement (in 4.5.1, page 83), Statement (in 4.6.2, page 88)

Fields

- `int PIPELINE_PROCESSING`
  - Defines that a stage of the pipeline is being processed.
    It must be notified and one must guarantee that access to meta information and methods is possible.
- `int PIPELINE_PROCESSED`
  - Defines that a stage of the pipeline was processed.
    There is no obligation of notifying this information. This is optional as it is always possible to detect completion of any stage of the pipeline by checking a request completion.
    If one decides to do so, one must guarantee that access to at least an object identification (e.g. `Statement`'s identification or `ExecutionPlan`'s identification) is possible. Every meta information and methods that are not available must throw an exception. Any attempt to change the object must throw an exception.
    It is worth noticing that it is not possible to cancel this event as it is a final state. Thus, any attempt to cancel this notification must throw an exception.
- `int PIPELINE_ERROR`
  - Defines that a stage of the pipeline did not ended correctly or was canceled.
    There is no obligation of notifying this information. This is optional as it is always possible to detect recoverable errors in any stage of the pipeline by a transaction abort. Other errors are detected when a database is put in panic mode.
    However, if one decides to do so, one must guarantee that access to at least an object identification (e.g. `Statement`'s identification or `ExecutionPlan`'s identification) is possible. Every meta information and methods that are not available must throw an exception. Any attempt to change the object must throw an exception.
    It is worth noticing that it is not possible to cancel this event as it is a final state. Thus, any attempt to cancel this notification must throw an exception.

4.2.23 Interface PreparedExecution

Provides methods to manipulate a prepared object.

This should be used to change and access information in any stage on requests to be prepared or executed.

The interface is built upon the `java.sql.PreparedStatement` and `java.sql.CallableStatement` and allows to redefine parameter values. However, one should do so by carefully checking parameter types through the `java.sql.ParameterMetaData`. If types do not match or an implicit conversation is not possible an exception is thrown.

Declaration `public interface PreparedExecution`

All known subinterfaces ParsedStatement (in 4.5.1, page 83), Statement (in 4.6.2, page 88)

Methods

- `java.sql.Array getArray(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `Array`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `Array`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.
- `java.math.BigDecimal getBigDecimal(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `BigDecimal`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `BigDecimal`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.
- `java.sql.Blob getBlob(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `Blob`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `Blob`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.
- `boolean getBoolean(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `boolean`.
  - **Parameters**
    * `index` – The index of the parameter to return.

  - **Returns** – The parameter value as a `boolean`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `byte getByte(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `byte`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `byte`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `byte[] getBytes(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java byte array.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a byte array
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.sql.Clob getClob(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `Clob`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `Clob`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.sql.Date getDate(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `java.sql.Date`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `java.sql.Date`.

  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.sql.Date getDate(int index, java.util.Calendar cal)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `java.sql.Date`.
  - **Parameters**
    * `index` – The index of the parameter to return.
    * `cal` – The `Calendar` to use for timezone and locale.
  - **Returns** – The parameter value as a `java.sql.Date`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `double getDouble(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `double`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `double`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `float getFloat(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `float`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `float`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `int getInt(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `int`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `int`.

  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `long getLong(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `long`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `long`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.lang.Object getObject(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `Object`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as an `Object`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.lang.Object getObject(int index, java.util.Map map)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `Object`.
  - **Parameters**
    * `index` – The index of the parameter to return.
    * `map` – The mapping to use for conversion from SQL to Java types.
  - **Returns** – The parameter value as an `Object`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.sql.ParameterMetaData getParameterMetaData()`
  `throws java.sql.SQLException`
  - **Description**
    Retrieves the number, types and properties of this `PreparedExecution` object's parameters.
  - **Returns** – a `ParameterMetaData` object that contains information about the number, types and properties of this `PreparedExecution` object's parameters

  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs
  - **See also**
    * `java.sql.ParameterMetaData`

- `java.sql.Ref getRef(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `Ref`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `Ref`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `short getShort(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `short`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `short`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.lang.String getString(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `String`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `String`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.sql.Time getTime(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `java.sql.Time`.
  - **Parameters**
    * `index` – The index of the parameter to return.

  - **Returns** – The parameter value as a `java.sql.Time`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.sql.Time getTime(int index, java.util.Calendar cal)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `java.sql.Time`.
  - **Parameters**
    * `index` – The index of the parameter to return.
    * `cal` – The `Calendar` to use for timezone and locale.
  - **Returns** – The parameter value as a `java.sql.Time`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.sql.Timestamp getTimestamp(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `java.sql.Timestamp`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `java.sql.Timestamp`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.sql.Timestamp getTimestamp(int index, java.util.Calendar cal)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `java.sql.Timestamp`.
  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `java.sql.Timestamp`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `java.net.URL getURL(int index)`
  `throws java.sql.SQLException`
  - **Description**
    Returns the value of the specified parameter as a Java `java.net.URL`.

  - **Parameters**
    * `index` – The index of the parameter to return.
  - **Returns** – The parameter value as a `URL`.
  - **Throws**
    * `java.sql.SQLException` – If an error occurs.

- `boolean isExecute()`
  - **Description**

    Returns `true` if this is related to executing prepared statements.

    For instance, the following commands should be classified in this category: EXECUTE [(params, ...)] | CREATE TABLE AS EXECUTE [(params, ...)]
  - **Returns** – a `boolean` value.

- `boolean isPrepare()`
  - **Description**

    Returns `true` if this is related to preparing statements.

    For instance, the following commands should be classified in this category: PREPARE
  - **Returns** – a `boolean` value.

- `void setArray(int i, java.sql.Array x)`
  `throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given `Array` object.

    The metalevel-application converts this to an SQL `ARRAY` value when it sends it to the database.
  - **Parameters**
    * `i` – the first parameter is 1, the second is 2, ...
    * `x` – an `Array` object that maps an SQL `ARRAY` value
  - **Throws**
    * `java.sql.SQLException` – if a database access error occurs

- `void setBigDecimal(int parameterIndex,`
  `java.math.BigDecimal x)`
  `throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given `java.math.BigDecimal` value.

    The metalevel-application converts this to an SQL `NUMERIC` value when it sends it to the database.
  - **Parameters**
    * `parameterIndex` – The index of the parameter, starting at position 1.

    * `x` – The parameter value.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs

- `void setBlob(int i, java.sql.Blob x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given `Blob` object.

    The metalevel-application converts this to an SQL `BLOB` value when it sends it to the database.
  - **Parameters**
    * `i` – the first parameter is 1, the second is 2, ...
    * `x` – a `Blob` object that maps an SQL `BLOB` value
  - **Throws**
    * `java.sql.SQLException` – if a database access error occurs

- `void setBoolean(int parameterIndex, boolean x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given Java `boolean` value.

    The metalevel-application converts this to an SQL `BIT` value when it sends it to the database.
  - **Parameters**
    * `parameterIndex` – The index of the parameter, starting at position 1.
    * `x` – The parameter value.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs

- `void setByte(int parameterIndex, byte x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given Java `byte` value.

    The metalevel-application converts this to an SQL `TINYINT` value when it sends it to the database.
  - **Parameters**
    * `parameterIndex` – The index of the parameter, starting at position 1.
    * `x` – The parameter value.
  - **Throws**
    * `java.sql.SQLException` – If a database access error occurs

- `void setBytes(int parameterIndex, byte[] x) throws java.sql.SQLException`

- **Description**

  Sets the designated parameter to the given Java array of bytes.

  The metalevel-application converts this to an SQL `VARBINARY` or `LONGVARBINARY` (depending on the argument's size relative to the metalevel-application's limits on `VARBINARY` values) when it sends it to the database.

- **Parameters**

  * `parameterIndex` – The index of the parameter, starting at position 1.
  * `x` – The parameter value.

- **Throws**

  * `java.sql.SQLException` – If a database access error occurs

- `void setClob(int i, java.sql.Clob x)`
  `throws java.sql.SQLException`

  - **Description**

    Sets the designated parameter to the given `Clob` object.

    The metalevel-application converts this to an SQL `CLOB` value when it sends it to the database.

  - **Parameters**

    * `i` – the first parameter is 1, the second is 2, ...
    * `x` – a `Clob` object that maps an SQL `CLOB` value

  - **Throws**

    * `java.sql.SQLException` – if a database access error occurs

- `void setDate(int parameterIndex, java.sql.Date x)`
  `throws java.sql.SQLException`

  - **Description**

    Sets the designated parameter to the given `java.sql.Date` value.

    The metalevel-application converts this to an SQL `DATE` value when it sends it to the database.

  - **Parameters**

    * `parameterIndex` – The index of the parameter, starting at position 1.
    * `x` – The parameter value.

  - **Throws**

    * `java.sql.SQLException` – If a database access error occurs

- `void setDate(int parameterIndex, java.sql.Date x,`
  `java.util.Calendar cal)`
  `throws java.sql.SQLException`

- **Description**

  Sets the designated parameter to the given `java.sql.Date` value, using the given `Calendar` object.

  The metalevel-application uses the `Calendar` object to construct an SQL `DATE` value, which the metalevel-application then sends to the database. With a `Calendar` object, the metalevel-application can calculate the date taking into account a custom timezone. If no `Calendar` object is specified, the metalevel-application uses the default timezone, which is that of the virtual machine running the metalvel-application.

- **Parameters**
  - `parameterIndex` – the first parameter is 1, the second is 2, ...
  - `x` – the parameter value
  - `cal` – the `Calendar` object the metalevel-application will use to construct the date
- **Throws**
  - `java.sql.SQLException` – if a database access error occurs

- `void setDouble(int parameterIndex, double x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given Java `double` value.

    The metalevel-application converts this to an SQL `DOUBLE` value when it sends it to the database.
  - **Parameters**
    - `parameterIndex` – The index of the parameter, starting at position 1.
    - `x` – The parameter value.
  - **Throws**
    - `java.sql.SQLException` – If a database access error occurs

- `void setFloat(int parameterIndex, float x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given Java `float` value.

    The metalevel-application converts this to an SQL `FLOAT` value when it sends it to the database.
  - **Parameters**
    - `parameterIndex` – The index of the parameter, starting at position 1.
    - `x` – The parameter value.
  - **Throws**
    - `java.sql.SQLException` – If a database access error occurs

- `void setInt(int parameterIndex, int x) throws java.sql.SQLException`

  - **Description**

    Sets the designated parameter to the given Java `int` value.

    The metalevel-application converts this to an SQL `INTEGER` value when it sends it to the database.

  - **Parameters**

    * `parameterIndex` – The index of the parameter, starting at position 1.
    * `x` – The parameter value.

  - **Throws**

    * `java.sql.SQLException` – If a database access error occurs

- `void setLong(int parameterIndex, long x)`
  `throws java.sql.SQLException`

  - **Description**

    Sets the designated parameter to the given Java `long` value.

    The metalevel-application converts this to an SQL `BIGINT` value when it sends it to the database.

  - **Parameters**

    * `parameterIndex` – The index of the parameter, starting at position 1.
    * `x` – The parameter value.

  - **Throws**

    * `java.sql.SQLException` – If a database access error occurs

- `void setNull(int parameterIndex, int sqlType)`
  `throws java.sql.SQLException`

  - **Description**

    Sets the designated parameter to SQL `NULL`

    Note that the SQL parameter type must be specified.

  - **Parameters**

    * `parameterIndex` – The index of the parameter, starting at position 1.
    * `sqlType` – The SQL type code defined in `Types` .

  - **Throws**

    * `java.sql.SQLException` – If a database access error occurs.

- `void setNull(int paramIndex, int sqlType,`
  `java.lang.String typeName)`
  `throws java.sql.SQLException`

  - **Description**

    Sets the designated parameter to SQL `NULL`.

    This version of the method `setNull` should be used for user-defined types and REF type parameters. Examples of user-defined types include: STRUCT, DISTINCT, JAVA_OBJECT, and named array types.

Note: To be portable, metalevel-applications must give the SQL type code and the fully-qualified SQL type name when specifying a NULL user-defined or REF parameter. In the case of a user-defined type the name is the type name of the parameter itself. For a REF parameter, the name is the type name of the referenced type. If a metalevel-application does not need the type code or type name information, it may ignore it.Although it is intended for user-defined and Ref parameters, this method may be used to set a null parameter of any JDBC type. If the parameter does not have a user-defined or REF type, the given typeName is ignored.

- **Parameters**
  - `paramIndex` – the first parameter is 1, the second is 2, ...
  - `sqlType` – a value from `java.sql.Types`
  - `typeName` – the fully-qualified name of an SQL user-defined type; ignored if the parameter is not a user-defined type or REF
- **Throws**
  - `java.sql.SQLException` – if a database access error occurs

- `void setObject(int parameterIndex, java.lang.Object x)`
  `throws java.sql.SQLException`
  - **Description**

    Sets the value of the designated parameter using the given object. The second parameter must be of type `Object`; therefore, the `java.lang` equivalent objects should be used for built-in types.

    The JDBC specification specifies a standard mapping from Java `Object` types to SQL types. The given argument will be converted to the corresponding SQL type before being sent to the database.

    If the object is of a class implementing the interface `SQLData`, the metalevel-application should call the method `SQLData.writeSQL` to write it to the SQL data stream. If, on the other hand, the object is of a class implementing `Ref`, `Blob`, `Clob`, `Struct`, or `Array`, the metalevel-application should pass it to the database as a value of the corresponding SQL type.

    In contrast to the JDBC specification, this method throws an exception if the class of the object does not match the paramter meta-information.
  - **Parameters**
    - `parameterIndex` – the first parameter is 1, the second is 2, ...
    - `x` – the object containing the input parameter value
  - **Throws**
    - `java.sql.SQLException` – if a database access error occurs or the type of the given object is ambiguous

- `void setObject(int parameterIndex, java.lang.Object x, int targetSqlType)`
  `throws java.sql.SQLException`

  - **Description**

    Sets the value of the designated parameter with the given object. This method is like the method `setObject` above, except that it assumes a scale of zero.

  - **Parameters**

    * `parameterIndex` – the first parameter is 1, the second is 2, ...
    * `x` – the object containing the input parameter value
    * `targetSqlType` – the SQL type (as defined in java.sql.Types) to be sent to the database

  - **Throws**

    * `java.sql.SQLException` – if a database access error occurs

- `void setObject(int parameterIndex, java.lang.Object x, int targetSqlType, int scale) throws java.sql.SQLException`

  - **Description**

    Sets the value of the designated parameter with the given object. The second argument must be an object type; for integral values, the `java.lang` equivalent objects should be used.

    The given Java object will be converted to the given targetSqlType before being sent to the database.

    If the object has a custom mapping (is of a class implementing the interface `SQLData`), the metalevel-application should call the method `SQLData.writeSQL` to write it to the SQL data stream. If, on the other hand, the object is of a class implementing `Ref`, `Blob`, `Clob`, `Struct`, or `Array`, the metalevel-application should pass it to the database as a value of the corresponding SQL type.

  - **Parameters**

    * `parameterIndex` – the first parameter is 1, the second is 2, ...
    * `x` – the object containing the input parameter value
    * `targetSqlType` – the SQL type (as defined in java.sql.Types) to be sent to the database. The scale argument may further qualify this type.
    * `scale` – for java.sql.Types.DECIMAL or java.sql.Types.NUMERIC types, this is the number of digits after the decimal point. For all other types, this value will be ignored.

  - **Throws**

    * `java.sql.SQLException` – if a database access error occurs

  - **See also**

    * `java.sql.Types`

- `void setRef(int i, java.sql.Ref x) throws java.sql.SQLException`

  - **Description**

    Sets the designated parameter to the given `REF(<structured-type>)` value.

The metalevel-application converts this to an SQL `REF` value when it sends it to the database.

- **Parameters**
  - `i` – the first parameter is 1, the second is 2, ...
  - `x` – an SQL `REF` value
- **Throws**
  - `java.sql.SQLException` – if a database access error occurs

- `void setShort(int parameterIndex, short x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given Java `short` value.

    The metalevel-application converts this to an SQL `SMALLINT` value when it sends it to the database.
  - **Parameters**
    - `parameterIndex` – The index of the parameter, starting at position 1.
    - `x` – The parameter value.
  - **Throws**
    - `java.sql.SQLException` – If a database access error occurs

- `void setString(int parameterIndex, java.lang.String x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given Java `String` value.

    The metalevel-application converts this to an SQL `VARCHAR` or `LONGVARCHAR` value (depending on the argument's size relative to the metalevel-application's limits on `VARCHAR` values) when it sends it to the database.
  - **Parameters**
    - `parameterIndex` – The index of the parameter, starting at position 1.
    - `x` – The parameter value.
  - **Throws**
    - `java.sql.SQLException` – If a database access error occurs

- `void setTime(int parameterIndex, java.sql.Time x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given `java.sql.Time` value.

    The metalevel-application converts this to an SQL `TIME` value when it sends it to the database.

- **Parameters**
  - `parameterIndex` – The index of the parameter, starting at position 1.
  - `x` – The parameter value.
- **Throws**
  - `java.sql.SQLException` – If a database access error occurs

- `void setTime(int parameterIndex, java.sql.Time x, java.util.Calendar cal) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given `java.sql.Time` value, using the given `Calendar` object.

    The metalevel-application uses the `Calendar` object to construct an SQL `TIME` value, which the metalevel-application then sends to the database. With a `Calendar` object, the metalevel-application can calculate the time taking into account a custom timezone. If no `Calendar` object is specified, the metalevel-application uses the default timezone, which is that of the virtual machine running the metalevel-application.
  - **Parameters**
    - `parameterIndex` – the first parameter is 1, the second is 2, ...
    - `x` – the parameter value
    - `cal` – the `Calendar` object the metalevel-application will use to construct the time
  - **Throws**
    - `java.sql.SQLException` – if a database access error occurs

- `void setTimestamp(int parameterIndex, java.sql.Timestamp x) throws java.sql.SQLException`
  - **Description**

    Sets the designated parameter to the given `java.sql.Timestamp` value.

    The metalevel-application converts this to an SQL `TIMESTAMP` value when it sends it to the database.
  - **Parameters**
    - `parameterIndex` – The index of the parameter, starting at position 1.
    - `x` – The parameter value.
  - **Throws**
    - `java.sql.SQLException` – If a database access error occurs

- `void setTimestamp(int parameterIndex, java.sql.Timestamp x, java.util.Calendar cal) throws java.sql.SQLException`

- **Description**

  Sets the designated parameter to the given `java.sql.Timestamp` value, using the given `Calendar` object.

  The metalevel-application uses the `Calendar` object to construct an SQL `TIMESTAMP` value, which the metalevel-application then sends to the database. With a `Calendar` object, the metalevel-application can calculate the timestamp taking into account a custom timezone. If no `Calendar` object is specified, the metalevel-application uses the default timezone, which is that of the virtual machine running the metalevel-application.

- **Parameters**

  * `parameterIndex` – the first parameter is 1, the second is 2, ...
  * `x` – the parameter value
  * `cal` – the `Calendar` object the metalevel-application will use to construct the timestamp

- **Throws**

  * `java.sql.SQLException` – if a database access error occurs

- `void setURL(int parameterIndex, java.net.URL x) throws java.sql.SQLException`

  - **Description**

    Sets the designated parameter to the given `java.net.URL` value.

    The metalevel-application converts this to an SQL `DATALINK` value when it sends it to the database.

  - **Parameters**

    * `parameterIndex` – the first parameter is 1, the second is 2, ...
    * `x` – the `java.net.URL` object to be set

  - **Throws**

    * `java.sql.SQLException` – if a database access error occurs

- `boolean wasNull() throws java.sql.SQLException`

  - **Description**

    This method tests whether the value of the last parameter that was fetched was actually a SQL NULL value.

  - **Returns** – `true` if the last parameter fetched was a NULL, `false` otherwise.

  - **Throws**

    * `java.sql.SQLException` – If an error occurs.

4.2.24 Interface Request

Reflects a client request.

Declaration `public interface Request`
extends `RequestConstant, Context, ExecutionControl`

Methods

- `RequestProcessor getRequestProcessor()`
  - **Description**
    Returns a reference to the request processor.
  - **Returns** – The reference to the request processor.
- `Transaction getTransaction()`
  - **Description**
    Returns a reference of the current active transaction object.
    However a `null` value may be returned when the request is not associated to a transaction such as in a stream processing environment. Thus, whether to return a `null` value or not is application dependent.
  - **Returns** – A reference to the current active transaction object, if there is any, `null` otherwise.

4.2.25 Interface RequestBeginListener

Defines the listener that will be notified whenever a request is made.

Declaration `public interface RequestBeginListener`

Methods

- `void handleRequestBegin(Request request)`
  - **Description**
    Is called whenever the listener is registered to receive request begin events.
    If the wait flag is set to `true` at registration time (see `setRequestBeginListener` (in 4.2.28, page 65), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the request being made.
    If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.
  - **Parameters**
    * `request` – The request on which the event occurs.

4.2.26 Interface RequestCompletionListener

Defines the listener that will be notified whenever a request is made.

Declaration `public interface RequestCompletionListener`

Methods

- `void handleRequestCompletion(Request request)`
  - **Description**
    Is called whenever the listener is registered to receive request completion events.
    If the wait flag is set to `true` at registration time (see `setRequestCompletionListener` (in 4.2.28, page 66), then this method implementation must call `continueExecution` (in 4.2.21, page 45).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the request being finishing.
    If the listener has previously called the `Request.setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.
  - **Parameters**
    * `request` – The request on which the event occurs.

4.2.27 Interface RequestConstant

Defines states and constants used by `Request` (in 4.2.24, page 62).

Declaration `public interface RequestConstant`

All known subinterfaces Request (in 4.2.24, page 62)

Fields

- `int REQUEST_PROCESSING`
  - Defines that a request is being processed.
    It must be notified and one must guarantee that access to meta information and methods is possible.
- `int REQUEST_PROCESSED`
  - Defines that a request was processed.
    Right after processing the last statement issued in the context of a request, one must change the state from processing to processed. The completion of a request does not include writing to the log as such operations at this point are executed asynchronously.
    There is no obligation of notifying this information. This is optional as it is always possible to detect completion of any request by checking directly on the pipeline.

If one decides to do so, one must guarantee that access to at least a a request identification is possible. Every meta information and methods that are not available must throw an exception.
It is worth noticing that it is not possible to cancel this event as it is a final state. Thus, any attempt to cancel this notification must throw an exception.

- `int REQUEST_ERROR`
  - Defines that a request did not ended correctly or was canceled.
    There is no obligation of notifying this information. This is optional as it is always possible to detect recoverable errors by a transaction abort. Other errors are detected when a database is put in panic mode.
    However, if one decides to do so, one must guarantee that access to at least a request identification is possible. Every meta information and methods that are not available must throw an exception. Any attempt to change the request must throw an exception.
    It is worth noticing that it is not possible to cancel this event as it is a final state. Thus, any attempt to cancel this notification must throw an exception.

4.2.28 Interface RequestProcessor

Handles listener registration for request events and has a request repository.

Declaration `public interface RequestProcessor`

Methods

- `Request getRequest(java.lang.String requestId)`
  - **Description**
    Returns a copy of the request object with the given id.
    To avoid synchronization problems, one must do exactly what follows:
    * Returning a copy of the object and throwing an exception if any method that attempts to change its state is called.
  - **Parameters**
    * `requestId` – The request identification.
  - **Returns** – A copy of the request object with the given id, if there is any, `null` otherwise.
- `void setRequestBeginListener(RequestBeginListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified upon request begin.
    Subsequent notifications, with respect to the request that initiated and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30), `Transaction.setNotificationIgnored` (in 4.2.7, page 30) or `Request.setNotificationIgnored` (in 4.2.7, page 30).

- **Parameters**
  - `listener` – The listener that handles request begin events.
  - `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
- **See also**
  - `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
  - `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

- `void setRequestCompletionListener(RequestCompletionListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified upon request completion.
    Subsequent notifications, with respect to the request completion and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30), `Transaction.setNotificationIgnored` (in 4.2.7, page 30) or `Request.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    - `listener` – The listener that handles request completion events.
    - `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    - `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    - `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.2.29 Interface Transaction

Reflects a transaction.

Declaration `public interface Transaction`
extends `TransactionConstant, Context, ExecutionControl`

Methods

- `ConnectionContext getConnection()`
  - **Description**
    Returns a reference to the reflected connection object.
    There is no need of returning a copy of this object as one must do when handling the method `getRequest` (in 4.2.29, page 67). Assuming a blocking notification, the connection context must be accessible by means of transaction.

  - **Returns** – the reference to a connection object.

- `Request getRequest()`

  - **Description**

    Returns a copy of a request object.

    To avoid synchronization problems, one must do exactly what follows:

    * Returning a copy of the object and throwing an exception if any method that attempts to change its state is called.

  - **Returns** – A copy of a request object, if there is any, otherwise `null`.

- `Transaction getTransaction()`

  - **Description**

    Returns a reference to the parent transaction.

    This method must return `null` if this transaction is not a sub-transaction.

    In contrast to `getRequest` (in 4.2.29, page 67), a reference to the transaction object must be returned. In this case, there is no problem as the parent-transaction context is finished only after committing or aborting its sub-transaction.

  - **Returns** – The reference to a parent transaction, if any, `null` otherwise.

- `int getTransactionIsolation()`
  `throws java.sql.SQLException`

  - **Description**

    Returns the transaction isolation level.

    Isolation levels available:

    * `TRANSACTION_READ_UNCOMMITTED` (in 4.2.32, page 71)
    * `TRANSACTION_READ_COMMITTED` (in 4.2.32, page 72)
    * `TRANSACTION_REPEATABLE_READ` (in 4.2.32, page 72)
    * `TRANSACTION_SERIALIZABLE` (in 4.2.32, page 72)
    * `TRANSACTION_SNAPSHOT` (in 4.2.32, page 72)

  - **Returns** – The transaction current isolation level.

  - **Throws**

    * `java.sql.SQLException` – If a database access error occurs.

  - **See also**

    * `Transaction.setTransactionIsolation()` (in 4.2.29, page 68)

- `TransactionProcessor getTransactionProcessor()`

  - **Description**

    Returns a reference to the transaction processor.

  - **Returns** – A reference to the associated transaction processor.

- `long getVersion()`

- **Description**

  Returns the transaction version number.

  This information is used as a timestamp and must be assigned when a transaction starts processing its first command (e.g., select, insert, update, etc).

  The version may not be available. In this case, this method must acknowledges this situation by returning `UNKNOWN_VERSION` (in 4.2.32, page 71). For instance, this may happen due to the fact that a transaction just started and does not have a version assigned to it. See `TRANSACTION_BEGINNING` (in 4.2.32, page 70) and `TRANSACTION_IDLE` (in 4.2.32, page 70).

- **Returns** – The transaction version, if it is available, `UNKNOWN_VERSION` (in 4.2.32, page 71) otherwise.

• `void setTransactionIsolation(int level) throws java.sql.SQLException`

- **Description**

  Changes the transaction isolation level.

  Isolation levels available:

  * `TRANSACTION_READ_UNCOMMITTED` (in 4.2.32, page 71)
  * `TRANSACTION_READ_COMMITTED` (in 4.2.32, page 72)
  * `TRANSACTION_REPEATABLE_READ` (in 4.2.32, page 72)
  * `TRANSACTION_SERIALIZABLE` (in 4.2.32, page 72)
  * `TRANSACTION_SNAPSHOT` (in 4.2.32, page 72)

  If this method is called during execution (*i.e.*, after processing the first transaction's command), an exception must be thrown.

- **Throws**

  * `java.sql.SQLException` – If *a)* a database access error occurs, *b)*the given parameter is not one of the constants or *c)*, one have tried to change it after starting processing the first command (*i.e.*, in the middle of a transaction).

- **See also**

  * `DatabaseMetaInfo (in 4.2.11, page 35)`
  * `Transaction.getTransactionIsolation() (in 4.2.29, page 67)`

4.2.30 Interface TransactionBeginListener

Defines the listener that will be notified whenever a transaction is being started.

Declaration `public interface TransactionBeginListener`

Methods

• `void handleTransactionBegin(Transaction transaction)`

– **Description**

Is called whenever a listener is registered to receive transaction begin events.

If the wait flag is set to `true` at registration time (see `setTransactionBeginListener` (in 4.2.34, page 73), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).

If the wait flag is set to `false` at registration time, then this method must be run in parallel with the transaction execution.

If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.

– **Parameters**

* `transaction` – The transaction on which the event occurs.

4.2.31 Interface TransactionCompletionListener

Defines the listener that will be notified whenever a transaction is being committed or aborted.

Declaration `public interface TransactionCompletionListener`

Methods

- `void handleTransactionCompletion(Transaction transaction)`

  – **Description**

  Is called whenever a listener is registered to receive transaction finish events.

  If the wait flag is set to `true` at registration time (see `setTransactionCompletionListener` (in 4.2.34, page 74), then this method implementation must call `continueExecution` (in 4.2.21, page 45).

  If the wait flag is set to `false` at registration time, then this method must be run in parallel with the transaction execution.

  If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.

  – **Parameters**

  * `transaction` – The transaction on which the event occurs.

4.2.32 Interface TransactionConstant

Defines states and constants used by `Transaction` (in 4.2.29, page 66).

Declaration `public interface TransactionConstant`

All known subinterfaces Transaction (in 4.2.29, page 66)

Fields

- `int TRANSACTION_BEGINNING`
  - Defines that a transaction is beginning.

    This is the first state and identifies that a transaction is initiating.

    There is no obligation of notifying this information. However, if one decides to do so, one must guarantee that access to `Transaction`'s meta information and methods is possible.

    In this state, a transaction may already be established but the control is not returned to the client or other database parts, which means that requests cannot be sent or processed.
- `int TRANSACTION_IDLE`
  - Defines that a transaction has begun.

    This is the second state and identifies that a transaction started but has not done anything yet.

    There is no obligation of notifying this information. However, if one decides to do so, one must guarantee that access to `Transaction`'s meta information and methods is possible.
- `int TRANSACTION_ACTIVE`
  - Transaction is trying to execute its first command read or write. This must only happen after IDLE.

    It must be notified and one must guarantee that access to `Transaction`'s meta information and methods is possible.

    In this state, a version is assigned to the transaction and it is quite important its notification.
- `int TRANSACTION_UPDATE`
  - Transaction is trying to execute its first write. This must only happen after ACTIVE.

    There is no obligation of notifying this information. However, if one decides to do so, one must guarantee that access to `Transaction`'s meta information and methods is possible.
- `int TRANSACTION_PREPARING`
  - Transaction is starting a commitment protocol. This must only happen after UPDATE.

    It must be notified and one must guarantee that access to `Transaction`'s meta information and methods is possible.
- `int TRANSACTION_PREPARED`

  - Transaction has finished the prepare protocol successfully. This must only happened after PREPARING.

    It must be notified and one must guarantee that access to `Transaction`'s meta information and methods is possible.

- `int TRANSACTION_COMMITTING`

  - Transaction is attempting to commit. This must only happen after IDLE, ACTIVE, UPDATE or PREPARED.

    It must be notified and one must guarantee that access to `Transaction`'s meta information and methods is possible.

- `int TRANSACTION_COMMITTED`

  - Transaction has successfully committed. This must only happen after COMMITTING.

    It must be notified and one must guarantee that access to at least a transaction identification is possible. Every method and meta information that is not available must throw an exception.

    It is worth noticing that it is not possible to cancel this event as it is a final state.

- `int TRANSACTION_ABORTING`

  - Transaction is aborting.

    It must be notified and one must guarantee that access to `Transaction`'s meta information and methods is possible.

- `int TRANSACTION_ABORTED`

  - Transaction has finished. This must only happen after ABORTING or COMMITTING

    It must be notified and one must guarantee that access to at least a transaction identification is possible. Every method and meta information that is not available must throw an exception.

    It is worth noticing that it is not possible to cancel this event as it is a final state.

- `int UNKNOWN_VERSION`

  - A constant stating that the version is unknown.

    This is used when a transaction have started and have not processed any command (*i.e.*, insert, update, delete or select command) , thus not being assigned a version to its execution.

- `int TRANSACTION_READ_UNCOMMITTED`

  - A constant stating that **dirty reads**, **non-repeatable reads** and **phantom reads** may occur.

    Dirty reads are described by the following example: a transaction *t1* reads a row changed by another transaction, *t2* and before *t2* commits. If any of the changes, made by *t2* in the row read by *t1* are rolled back, *t1* will have retrieved an invalid row.

- `int TRANSACTION_READ_COMMITTED`
  - A constant stating that **dirty reads** must not happen; **non-repeatable reads** and **phantom** reads may occur.
    In this isolation level, a transaction must not be allowed to read a row with uncommitted changes in it.
- `int TRANSACTION_REPEATABLE_READ`
  - A constant stating that **dirty reads** and **non-repeatable reads** must not happen. **Phantom reads** may occur.
    In this isolation level, transactions must not: *a)* be allowed to read a row with uncommitted changes; *b)*find **non-repeatable read** issues.
    A non-repeatable read issue is described by the following situation: a transaction *t1* reads a row, afterwards, a second transaction, *t2*, updates the very same row. Finally, *t1* rereads the row, eventually getting different values from the first read operation.
- `int TRANSACTION_SERIALIZABLE`
  - A constant stating that **dirty reads**, **non-repeatable reads** and **phantom reads** must not happen.
    In this isolation level, restrictions described in `TRANSACTION_REPEATABLE_READ` (in 4.2.32, page 72) must hold, as well as there must not be any phantom rows issues.
    A phantom row is described by the following example: a transaction, *t1* reads all rows that meet a `WHERE` clause; afterwards a second transaction, *t2* inserts a row that satisfies the `WHERE` condition; finally, *t1* rereads using the same `WHERE` clause, retrieving the additional "phantom" rows, created by *t2*.
- `int TRANSACTION_SNAPSHOT`
  - A constant stating that **dirty reads**, **non-repeatable reads** and **phantom reads** must not happen.
    In this isolation level, restrictions described in `TRANSACTION_REPEATABLE_READ` (in 4.2.32, page 72) must hold, as well as there must not be any phantom rows issues. Unfortunately, write skew problems arise.
    A phantom row is described by the following example: a transaction, *t1* reads all rows that meet a `WHERE` clause; afterwards a second transaction, *t2* inserts a row that satisfies the `WHERE` condition; finally, *t1* rereads using the same `WHERE` clause, retrieving the additional "phantom" rows, created by *t2*.

4.2.33 Interface TransactionPrepareListener

Defines the listener that will be notified whenever a transaction is being prepared to be committed.

Declaration `public interface TransactionPrepareListener`

Methods

- `void handleTransactionPrepare(Transaction transaction)`
  - **Description**
    Is called whenever a listener is registered to receive transaction prepare events.
    If the wait flag is set to `true` at registration time (see `setTransactionPrepareListener` (in 4.2.34, page 74), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the transaction execution.
    If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.
  - **Parameters**
    * `transaction` – The transaction on which the event occurs.

4.2.34 Interface TransactionProcessor

Handles listener registration for transaction events and has a transaction repository.

Declaration `public interface TransactionProcessor`

Methods

- `Transaction getTransaction(java.lang.String transactionId)`
  - **Description**
    Returns a copy of the transaction object with the given id.
    To avoid synchronization problems, one must do exactly what follows:
    * Returning a copy of the object and throwing an exception if any method that attempts to change its state is called.
  - **Parameters**
    * `transactionId` – The transaction identification.
  - **Returns** – A copy of the transaction object with the given id, if there is any, `null` otherwise.
- `void setTransactionBeginListener(TransactionBeginListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified upon when a transaction is being started up.

Subsequent notifications, with respect to the transaction that is being started up and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30) or `Transaction.setNotificationIgnored` (in 4.2.7, page 30).

- **Parameters**
  - `listener` – The listener that handles transaction startup events.
  - `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
- **See also**
  - `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
  - `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

- `void setTransactionCompletionListener(TransactionCompletionListener listener, boolean wait)`
  - **Description**

    Registers a listener that must be notified when a transaction is being finished.

    Subsequent notifications, with respect to the transaction that is being finished and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30) or `Transaction.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    - `listener` – The listener that is to handle transaction finish events.
    - `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    - `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    - `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

- `void setTransactionPrepareListener(TransactionPrepareListener listener, boolean wait)`
  - **Description**

    Registers a listener that is notified when the transaction is being prepared.

    Subsequent notifications, with respect to the transaction that is being prepared and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30) or `Transaction.setNotificationIgnored` (in 4.2.7, page 30).

- **Parameters**
  - `listener` – The listener for this event
  - `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
- **See also**
  - `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
  - `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

- `void setTransactionUpdateListener(TransactionUpdateListener listener, boolean wait)`
  - **Description**

    Registers a listener that must be notified when a transaction performed its first update statement.

    Subsequent notifications, with respect to the transaction that is being receiving its first update and its inner contexts, may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30) or `Transaction.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    - `listener` – The listener that is to handle this event.
    - `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    - `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    - `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.2.35 Interface TransactionUpdateListener

Defines the listener that will be notified whenever a transaction executed its first update statement.

Declaration `public interface TransactionUpdateListener`

Methods

- `void handleTransactionUpdate(Transaction transaction)`

– **Description**

Is called whenever the listener is registered to receive information on a transaction that executed its first update statement.

If the wait flag is set to `true` at registration time (see `setTransactionUpdateListener` (in 4.2.34, page 75), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).

If the wait flag is set to `false` at registration time, then this method must be run in parallel with the transaction execution.

If the listener has previously called the `setNotificationIgnored` (in 4.2.7, page 30) method, then this notification must not happen.

– **Parameters**

* `transaction` – The transaction on which the event occurs.

4.3 Package gorda.db.executor

Events and interfaces associated with tuple sets (*i.e.*, write sets and result sets) and transaction log.

4.3.1 Interface ExecutorStage

Handles listener registration for object set events.

Declaration `public interface ExecutorStage`

Methods

- `void setObjectSetReadListener(ObjectSetReadListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified when an object set related to read information is being processed. Subsequent notifications, with respect to the logger object set and subsequent stages may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30), `Transaction#setNotificationIgnored` (in 4.2.7, page 30) or `Request#setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    * `listener` – The listener that handles object set events related to read information.
    * `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    * `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    * `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)
- `void setObjectSetWriteListener(ObjectSetWriteListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified when an object set related to written information is being processed. Subsequent notifications, with respect to the logger object set and subsequent stages may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30), `Transaction.setNotificationIgnored` (in 4.2.7, page 30) or `Request.setNotificationIgnored` (in 4.2.7, page 30).

- **Parameters**
  - `listener` – The listener that handles the object set events related to written information.
  - `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
- **See also**
  - `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
  - `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.3.2 Interface ObjectSet

Determines an object set: an object generated after processing a statement.

The interface is built upon the `java.sql.ResultSet` and is used to define which information was written and read while processing commands (e.g., update, delete, insert, etc).

For written information, the object set must be defined as follows

- a delete `TYPE_DML_DELETE` (in 4.3.3, page 79): a result set with the deleted tuples.
- `TYPE_DML_INSERT` (in 4.3.3, page 79)a result set with the inserted tuples
- an update `TYPE_DML_UPDATE` (in 4.3.3, page 79): a result set where each entry is composed by the new tuple plus the old tuple.

For read information, the object set must be defined as follows

- a delete (in 4.3.3, page 79): a result set with the read tuples.

  However, it is not a requirement to have read information if one decides to implement this stage. For further discussions on how to obtain a read set see **GORDA Documents** (at http://gorda.di.uminho.pt).

Declaration `public interface ObjectSet`
extends `java.sql.ResultSet, gorda.db.PipelineConstant, ObjectSetConstant, gorda.db.ContextReference, gorda.db.ExecutionControl`

Methods

- `ExecutorStage getExecutorStage()`
  - **Description**
    Returns a reference to the executor stage.
  - **Returns** – A reference to the executor stage.

- `int getObjectSetType()`
  - **Description**
    Returns the type of the object set.
    The type of the object is one: `TYPE_DML_DELETE` (in 4.3.3, page 79); `TYPE_DML_INSERT` (in 4.3.3, page 79); `TYPE_DML_UPDATE` (in 4.3.3, page 79) or `TYPE_NO_UPDATES` (in 4.3.3, page 79).
  - **Returns** – The object set type.

4.3.3 Interface ObjectSetConstant

Defines states and constants used by `ObjectSet` (in 4.3.2, page 78).

Declaration `public interface ObjectSetConstant`

All known subinterfaces ObjectSet (in 4.3.2, page 78)

Fields

- `int TYPE_DML_INSERT`
  - Defines that information is about to be inserted.
- `int TYPE_DML_DELETE`
  - Defines that information is about to be deleted. It is worth noticing that any changes to the object set does not make sense in this state. An exception is thrown if one tries to do so.
- `int TYPE_DML_UPDATE`
  - Defines that information is about to be updated.
- `int TYPE_NO_CHANGES`
  - Defines that information was read.

4.3.4 Interface ObjectSetReadListener

Defines the listener that will be notified whenever an object set (*i.e.*, read information) is generated.

Declaration `public interface ObjectSetReadListener`

Methods

- `void handleObjectSetRead(ObjectSet objSet)`
  - **Description**
    Is called whenever the listener is registered to receive object set (*i.e.*, read information) events.
    If the wait flag is set to `true` at registration time (see `setObjectSetReadListener` (in 4.3.1, page 77), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the object set execution.
  - **Parameters**
    * `objSet` – The object set (*i.e.*, read information) on which the event occurs.

4.3.5 Interface ObjectSetWriteListener

Defines a listener that will be notified whenever an object set (*i.e.*, written information) is generated.

Declaration `public interface ObjectSetWriteListener`

Methods

- `void handleObjectSetWrite(ObjectSet objSet)`
  - **Description**
    Is called whenever the listener is registered to receive object set (*i.e.*, written information) events.
    If the wait flag is set to `true` at registration time (see `ObjectSetWriteListener` (in 4.3.1, page 77), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the object set execution.
  - **Parameters**
    * `objSet` – The object set (*i.e.*, written information) on which the event occurs.

4.4 Package gorda.db.logminer

Events and interfaces associated with transaction log that provide transparent access to its content.

4.4.1 Interface LoggerObjectSet

Defines a logger object set provided by a log miner mechanism. At least tuples must be notified and the format must be the same defined by the `ObjectSet` (in 4.3.2, page 78).

Declaration `public interface LoggerObjectSet`
extends `gorda.db.PipelineConstant, gorda.db.ExecutionControl`

Methods

- `gorda.db.executor.ObjectSet getLoggerObjectSet()`
  - **Description**
    Returns a reference to an object set.
  - **Returns** – A reference to an object set.
- `LogMinerStage getLogMinerStage()`
  - **Description**
    Returns a reference to the log miner stage.
  - **Returns** – A reference to the log miner stage.

4.4.2 Interface LoggerObjectSetExecutionListener

Defines the listener that will be notified whenever a logger object set is being processed.

Declaration `public interface LoggerObjectSetExecutionListener`

Methods

- `void handleLoggerObjectSetExecution(LoggerObjectSet logger)`
  - **Description**
    Is called whenever the listener is registered to receive logger object set events.
    If the wait fla iss set to `true` at registration time (see `setLoggerObjectSetExecutionListener` (in 4.4.3, page 82), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the logger object set execution.
  - **Parameters**
    * `logger` – The logger object set on which the event occurs.

4.4.3 Interface LogMinerStage

Handles listener registration for logger object set events.

Declaration `public interface LogMinerStage`

Methods

- `void setLoggerObjectSetExecutionListener(LoggerObjectSetExecutionListe`
  `listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified when a logger object set is being. Subsequent notifications, with respect to the logger object set and subsequent stages may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30), `Transaction.setNotificationIgnored` (in 4.2.7, page 30) or `Request.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    * `listener` – The listener that handles logger object set events.
    * `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    * `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    * `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.5 Package gorda.db.parser

Events and interfaces associated with parse trees.

4.5.1 Interface ParsedStatement

Determines a parsed statement: an object generated by the parser stage.

Declaration `public interface ParsedStatement`
extends `gorda.db.PipelineConstant, gorda.db.ContextReference, gorda.db.PreparedExecution, gorda.db.ExecutionControl`

Methods

- `boolean altersDatabaseCatalog()`
  - **Description**
    Returns true if this request invalidates somehow the Database Catalog.
  - **Returns** – true if the database catalog is altered.
- `boolean altersDatabaseSchema()`
  - **Description**
    Returns true if this request invalidates somehow the Database Schema.
  - **Returns** – true if the database schema is altered.
- `boolean altersStoredProcedureList()`
  - **Description**
    Returns true if this request invalidates somehow the Stored Procedure List.
  - **Returns** – true if the stored procedure list is altered.
- `boolean altersUserDefinedTypes()`
  - **Description**
    Returns true if this request invalidates somehow the User Defined Types.
  - **Returns** – true if the UDTs are altered.
- `boolean altersUsers()`
  - **Description**
    Returns true if this request invalidates somehow the Users definition or rights.
  - **Returns** – true if the users are altered.
- `ParserStage getParserStage()`
  - **Description**
    Returns a reference to the parser stage.

  - **Returns** – The reference to the parser stage.

- `java.util.Set getReadLockedTables()`
  - **Description**
    Returns the list of table names that must be read locked by the execution of this request.
  - **Returns** – A set of string containing table names to be read locked by this request. This list may be null or empty if no table needs to be locked.

- `java.util.Set getReadTables()`
  - **Description**
    Returns the list of table names that should be read by the execution of this request.
  - **Returns** – Set of string containing table names to be read by this request. This list may be null or empty if no table needs to be read.

- `java.util.Set getWriteLockedTables()`
  - **Description**
    Returns the list of table names that must be write locked by the execution of this request.
  - **Returns** – A set of string containing table names to be write locked by this request. This list may be null or empty if no table needs to be locked.

- `java.util.Set getWriteTables()`
  - **Description**
    Returns the list of table names that should be written by the execution of this request.
  - **Returns** – Set of string containing table names to be write by this request. This list may be null or empty if no table needs to be written.

- `boolean isAlter()`
  - **Description**
    Returns `true` if this request in a `ALTER` statement. It is worth noticing that this method subsumes the methods: `altersUsers` (in 4.5.1, page 83), `altersUserDefinedTypes` (in 4.5.1, page 83) , `altersStoredProcedureList` (in 4.5.1, page 83), (in 4.5.1, page 83) and `altersDatabaseCatalog` (in 4.5.1, page 83).
  - **Returns** – a `boolean` value

- `boolean isCreate()`
  - **Description**
    Returns `true` if this request in a `CREATE` statement.
  - **Returns** – a `boolean` value

- `boolean isCursor()`

  - **Description**

    Returns `true` if this request is related to cursor(s).

    For instance, the following commands should be classified in this category: DECLARE CURSOR | FETCH/MOVE | CLOSE

  - **Returns** – a `boolean` value

- `boolean isDelete()`

  - **Description**

    Returns `true` if this request in a `DELETE` statement.

  - **Returns** – a `boolean` value

- `boolean isDrop()`

  - **Description**

    Returns `true` if this request in a `DROP` statement.

  - **Returns** – a `boolean` value

- `boolean isInsert()`

  - **Description**

    Returns `true` if this request in an `INSERT` statement.

  - **Returns** – a `boolean` value

- `boolean isLock()`

  - **Description**

    Returns `true` if this request has hints on locks. For instance, the following commands should be classified in this category: LOCK TABLE.

  - **Returns** – a `boolean` value

- `boolean isOther()`

  - **Description**

    Returns `true` if this an administrative request.

    For instance, the following commands must be classified as administrative commands and most likely just makes sense locally: CHECKPOINT | REINDEX | SET.

  - **Returns** – a `boolean` value

- `boolean isSelect()`

  - **Description**

    Returns `true` if this request in a `SELECT` statement.

  - **Returns** – a `boolean` value

- `boolean isTransaction()`

- **Description**

  Returns `true` if this request is related to transaction commands.

  For instance, the following commands should be classified in this category: BEGIN | COMMIT | ROLLBACK | SAVE POINT | PREPARE TRANSACTION

- **Returns** – a `boolean` value

- `boolean isUpdate()`

  - **Description**

    Returns `true` if this request in an `UPDATE` statement.

  - **Returns** – a `boolean` value

4.5.2 Interface ParsedStatementExecutionListener

Defines the listener that will be notified whenever a parsed statement event is being processed.

Declaration `public interface ParsedStatementExecutionListener`

Methods

- `void handleParsedStatementExecution(ParsedStatement parsedSt)`

  - **Description**

    Is called whenever the listener is registered to receive parsed statement events.

    If the wait flag is set to `true` at registration time (see `setParsedStatementExecutionListener` (in 4.5.3, page 87), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).

    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the parsed statement execution.

  - **Parameters**

    * `parsedSt` – The parsed statement on which the event occurs.

4.5.3 Interface ParserStage

Handles listener registration for parsed statement events.

Declaration `public interface ParserStage`

Methods

- `void setParsedStatementExecutionListener(ParsedStatementExecutionListener listener, boolean wait)`
  - **Description**

    Registers a listener that must be notified when a parsed statement is being processed.

    Subsequent notifications, with respect to the execution plan and subsequent stages may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30), `Transaction.setNotificationIgnored` (in 4.2.7, page 30) or `Request.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    - `listener` – The listener that handles parsed statement events.
    - `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    - `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    - `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.6 Package gorda.db.receiver

Events and interfaces associated with raw statements.

4.6.1 Interface ReceiverStage

Handles listener registration for statement events.

Declaration `public interface ReceiverStage`

Methods

- `void setStatementExecutionListener(StatementExecutionListener listener, boolean wait)`
  - **Description**
    Registers a listener that must be notified when an statement is being processed.
    Subsequent notifications, with respect to the statement and subsequent stages may be canceled afterwards using `Dbms.setNotificationIgnored` (in 4.2.7, page 30), `Database.setNotificationIgnored` (in 4.2.7, page 30), `Connection.setNotificationIgnored` (in 4.2.7, page 30), `Transaction.setNotificationIgnored` (in 4.2.7, page 30) or `Request.setNotificationIgnored` (in 4.2.7, page 30).
  - **Parameters**
    * `listener` – The listener that handles statement events.
    * `wait` – if `true` the notifier must wait for the listener to proceed, `false` otherwise.
  - **See also**
    * `ExecutionControl.continueExecution()` (in 4.2.21, page 45)
    * `ExecutionControl.cancelExecution()` (in 4.2.21, page 44)

4.6.2 Interface Statement

Defines an object statement: a command or a set of commands sent by a client to be processed.

Declaration `public interface Statement`
extends `gorda.db.PipelineConstant, gorda.db.ContextReference, gorda.db.PreparedExecution, gorda.db.ExecutionControl`

Methods

- `ReceiverStage getReceiverStage()`
  - **Description**
    Returns a reference to the receiver stage.
  - **Returns** – A reference to the receiver stage.
- `java.lang.String getStatement()`
  - **Description**
    Returns the statement.
  - **Returns** – The statement.
- `void setStatement(java.lang.String statement)`
  - **Description**
    Sets the statement.
  - **Parameters**
    * `statement` – The statement.

4.6.3 Interface StatementExecutionListener

Defines the listener that will be notified whenever a statement is being processed.

Declaration `public interface StatementExecutionListener`

Methods

- `void handleStatementExecution(Statement statement)`
  - **Description**
    Is called whenever a listener is registered to receive statement events.
    If the wait flag is set to `true` at registration time (see `setStatementExecutionListener` (in 4.6.1, page 88), then this method implementation must call `continueExecution` (in 4.2.21, page 45) or `cancelExecution` (in 4.2.21, page 44).
    If the wait flag is set to `false` at registration time, then this method must be run in parallel with the statement execution.
  - **Parameters**
    * `statement` – The statement on which the event occurs..

5 Samples

5.1 Query Caching

This sample shows how to implement a simple query cache. Besides being an important issue in itself for replicated databases, this is also an example of an important technique: how to replace statements in the context of a client initiated transaction while still faking result sets obtained from a different source. This is useful for query shipping and load balancing, for instance.

Note that this implementation fails to properly invalidate the cache when update operations are issued. This could be solved by using the object-set stage to inspect modified data.

```
public class QueryCache implements StatementExecutionListener,
    DatabaseStartupListener {
  private static RequestProcessor reqProc;

  private static LinkedHashMap cache = new LinkedHashMap() {
    protected boolean removeEldestEntry(Map.Entry entry) {
      return size() > 100;
    }
  };
```

As usual, the first step is to register all required event handlers. In detail, we use the statement handler.

```
  public QueryCache(DatabaseProcessor dbProc, RequestProcessor reqProc,
      ReceiverStage stmtProc) {
    QueryCache.reqProc = reqProc;
    dbProc.setDatabaseStartupListener(this, true);
    stmtProc.setStatementExecutionListener(this, true);
  }
```

The core of the query cache is the method that gets called as a Java stored procedure. It builds a result set from previously cached results and returns it.

```
  public static void cacheLookup(String reqId, String query, ResultSet[] rs1)
      throws SQLException {
    Connection c = DriverManager.getConnection("jdbc:default:connection");

    Transaction tx = reqProc.getRequest(reqId).getTransaction();
    Utils.info("looking␣up:␣txid=" + tx.getId());

    java.sql.Statement s = c.createStatement();
    String cached = (String) cache.get(query);
    if (cached == null) {
      Utils.info("not␣found,␣executing:␣" + query);
      ResultSet rs = s.executeQuery(query);
      cached = "values␣";
      boolean first = true;
      while (rs.next()) {
        cached += "(";
        if (!first)
          cached += ",␣";
        first = false;
        for (int i = 0; i < rs.getMetaData().getColumnCount(); i++) {
          if (i != 0)
            cached += ",";
          cached += "'" + rs.getString(i + 1) + "'";
        }
        cached += ")";
      }
      cache.put(query, cached);
    }
    rs1 = new ResultSet[1];
    rs1[0] = s.executeQuery(cached);
    c.close();
  }
```

The key usage of the specification is in the intercepting and replacing statements with calls to the cache lookup procedure.

```
  public void handleStatementExecution(Statement statement) {
    try {
      switch (statement.getState()) {
      case Statement.PIPELINE_PROCESSING:
        if (statement.getStatement().toLowerCase().startsWith("select"))
          statement.setStatement("CALL␣cacheLookup(’"
              + statement.getRequest().getId() + "’,␣’"
              + statement.getStatement() + "’)");
        statement.continueExecution();
        break;
      case Statement.PIPELINE_PROCESSED:
        statement.continueExecution();
        break;
      case Statement.PIPELINE_ERROR:
        statement.continueExecution();
        Utils.cleanUp(new SQLException("ObjectSet␣-␣WriteSet␣Error."));
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
```

The Java stored procedure is registered upon startup of each database, transparently to client configuration.

```
  public void handleDatabaseStartup(Database database) {
    try {
      switch (database.getContextState()) {

      case Database.DATABASE_STARTING:
        database.continueExecution();
        break;
      case Database.DATABASE_UP:

        DataSource ds = database.getDataSource();
        Connection c = ds.getConnection();

        java.sql.Statement s = c.createStatement();
        s.execute("CREATE␣PROCEDURE␣cacheLookup(reqid␣VARCHAR(10),␣query␣VARCHAR(100))"
         + "PARAMETER␣STYLE␣JAVA␣LANGUAGE␣JAVA␣READS␣SQL␣DATA␣DYNAMIC␣RESULT␣SETS␣1"
         + "EXTERNAL␣NAME␣’gorda.demo.QueryCache.cacheLookup’");
        s.close();
        c.close();

        database.continueExecution();
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
}
```

5.2 Streaming

This sample shows how to use to capture changes to the database and publish them to a JMS compliant publish-subscribe system. This allows any database server that implements the specification to achieve a similar effect to Oracle Streams.

```
public class ChangePublisher implements TransactionBeginListener,
    TransactionCompletionListener, ObjectSetWriteListener {

  private MessageProducer sender;

  private Session session;
```

The first step is to setup the meta-level code by registering all event listeners. Note that we synchronously wait for transaction begin and end events.

```
  public ChangePublisher(Session session, Destination dest,
      TransactionProcessor tranProc, ExecutorStage objProc)
      throws JMSException {
    this.session = session;
    sender = session.createProducer(dest);
    tranProc.setTransactionBeginListener(this, true);
    tranProc.setTransactionCompletionListener(this, true);
    objProc.setObjectSetWriteListener(this, false);
  }
```

When a transaction begins, either explicitly or implicitly, we are notified and initialize the attached state to hold all changes until the transaction commits.

```
  public void handleTransactionBegin(Transaction transaction) {
    try {
      switch (transaction.getContextState()) {

      case Transaction.TRANSACTION_BEGINNING:
        transaction.setAttachment(new Store());
      case Transaction.TRANSACTION_IDLE:
      case Transaction.TRANSACTION_ACTIVE:
        transaction.continueExecution();
        break;
      }
    } catch (Exception ex) {
      Utils.cleanUp(ex);
    }
  }

  public void handleTransactionCompletion(Transaction transaction) {
    Store state = (Store) transaction.getAttachment();
    TextMessage message = null;

    try {
      switch (transaction.getContextState()) {
```

We are not interested in doing anything else before the transaction commits. So we let it proceed.

```
      case Transaction.TRANSACTION_COMMITTING:
        transaction.continueExecution();
        break;
```

If the transaction has commited successfully, we wrap the changes performed as a text message and publish it using JMS. Properties are set on the message to allow filtering.

```
      case Transaction.TRANSACTION_COMMITTED:
        message = session.createTextMessage();
        message.setText("Committed␣transaction␣" + transaction.getId()
            + "\n" + state.toString());
        message.setBooleanProperty("committed", true);
        message.setStringProperty("database", transaction
```

```
            .getConnection().getDatabase().getId());
        message.setIntProperty("writes", state.writes());
        sender.send(message);
        transaction.continueExecution();
        break;

      case Transaction.TRANSACTION_ABORTING:
        transaction.continueExecution();
        break;
```

If the transaction has aborted, we nonetheless performed as a text message and publish it using JMS. By setting the commited property, we allow filtering to occur within the network.

```
      case Transaction.TRANSACTION_ABORTED:
        message = session.createTextMessage();
        message.setText("Aborted␣transaction:␣" + transaction.getId()
            + "\n" + state.toString());
        message.setBooleanProperty("committed", false);
        message.setStringProperty("database", transaction
            .getConnection().getDatabase().getId());
        sender.send(message);
        transaction.continueExecution();
        break;
      }
    } catch (Exception ex) {
      Utils.cleanUp(ex);
    }
  }
```

Upon each modification being performed, we store it in the context of the transaction for later use.

```
  public void handleObjectSetWrite(ObjectSet objSet) {
    try {
      switch (objSet.getState()) {

      case ObjectSet.PIPELINE_PROCESSING:
        Store state = (Store) objSet.getRequest().getTransaction()
            .getAttachment();

        Utils.makeUpdate(objSet, state);

        objSet.continueExecution();
        break;

      case ObjectSet.PIPELINE_PROCESSED:
        objSet.continueExecution();
        break;

      case ObjectSet.PIPELINE_ERROR:
        objSet.continueExecution();
        Utils.cleanUp(new SQLException("ObjectSet␣-␣WriteSet␣Error."));
        break;

      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
}
```

5.3 Replication

A simple asyncrhonous primary-backup replication protocol can be achived by relaying changes to a backup replica using some communication protocol. In this example, a simple stream socket is used, thus minimizing the amount of code.

```
public class NaivePrimary implements TransactionBeginListener,
    TransactionCompletionListener, ObjectSetWriteListener, Runnable {
```

The primary, or master, must collect all changes done by clients. This is similar to capturing changes in Section 5.2.

```
  public NaivePrimary(TransactionProcessor tranProc,
      ExecutorStage objProc) {
    tranProc.setTransactionBeginListener(this, true);
    tranProc.setTransactionCompletionListener(this, true);
    objProc.setObjectSetWriteListener(this, true);
  }

  public void handleTransactionBegin(Transaction transaction) {
    try {
      switch (transaction.getContextState()) {
      case Transaction.TRANSACTION_BEGINNING:
        transaction.setAttachment(new Store());
      case Transaction.TRANSACTION_IDLE:
      case Transaction.TRANSACTION_ACTIVE:
        transaction.continueExecution();
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
```

Upon each transaction commiting, we queue updates for asynchronous propagation by the separate thread.

```
  public synchronized void handleTransactionCompletion(Transaction transaction) {
    try {
      switch (transaction.getContextState()) {
      case Transaction.TRANSACTION_COMMITTING:
        queue.add(transaction);
        notifyAll();
        break;
      case Transaction.TRANSACTION_COMMITTED:
      case Transaction.TRANSACTION_ABORTING:
      case Transaction.TRANSACTION_ABORTED:
        transaction.continueExecution();
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }

  public void handleObjectSetWrite(ObjectSet objSet) {
    try {
      switch (objSet.getState()) {

      case ObjectSet.PIPELINE_PROCESSING:
        Store state = (Store) objSet.getRequest().getTransaction()
            .getAttachment();
        Utils.makeUpdate(objSet, state);

        Utils.info("write␣value␣" + state.toString());

        objSet.continueExecution();
        break;

      case ObjectSet.PIPELINE_PROCESSED:
```

```
        objSet.continueExecution();
        break;

      case ObjectSet.PIPELINE_ERROR:
        objSet.continueExecution();
        Utils.cleanUp(new SQLException("ObjectSet - WriteSet Error."));
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
```

The core of the primary replica is a separate thread that connects to a backup replica and pushes updates as they become available on the local outgoing queue.

```
  public void run() {
    try {
      ServerSocket ssock = new ServerSocket(12345);

      while (true) {
        Socket sock = ssock.accept();

        updateBackup(sock);
      }
    } catch (Exception ex) {
      Utils.cleanUp(ex);
    }
  }

  private synchronized void updateBackup(Socket sock) {
    Utils.info("Backup connected.");

    try {
      ObjectOutputStream outstr = new ObjectOutputStream(sock
          .getOutputStream());
      ObjectInputStream instr = new ObjectInputStream(sock
          .getInputStream());

      while (true) {
        while (queue.isEmpty())
          wait();
        Utils.info("Sending update.");

        Transaction evt = (Transaction) queue.removeFirst();

        Store state = (Store) evt.getAttachment();

        LinkedList concatStore = new LinkedList();

        concatStore.addAll(state.insertStore);
        concatStore.addAll(state.updateStore);
        concatStore.addAll(state.deleteStore);

        outstr.writeObject(concatStore);

        outstr.flush();

        Utils.info("Sending done, waiting acknowledgment.");

        if (!instr.readBoolean()) {
          evt.cancelExecution();
          break;
        }

        Utils.info("Acknowledgment received.");

        evt.continueExecution();
      }
      Utils.info("Backup refused update.");
    } catch (Exception e) {
      Utils.error("Backup disconnected.", e);
    }
```

```
    try {
      sock.close();
    } catch (IOException ex) {
      Utils.error("Socket error.", ex);
    }
  }

  private LinkedList queue = new LinkedList();
}
```

The backup replica waits for updates being pushed by the primary replica and applies them using the JDBC interface. Notice that the meta-level code is used to ensure that no local updates are performed to the backup, which is available for read-only transactions.

```
public class NaiveBackup implements DatabaseStartupListener,
    ConnectionStartupListener, TransactionBeginListener,
    TransactionCompletionListener, ObjectSetWriteListener, Runnable {

  private String user = "refmanager";

  private Connection conn;

  public NaiveBackup(String db, DatabaseProcessor dbProc,
      ConnectionProcessor connProc, TransactionProcessor tranProc,
      ExecutorStage objProc) throws SQLException {
    dbProc.setDatabaseStartupListener(this, true);
    connProc.setConnectionStartupListener(this, true);
    tranProc.setTransactionBeginListener(this, true);
    tranProc.setTransactionCompletionListener(this, true);
    objProc.setObjectSetWriteListener(this, true);
  }

  public void handleTransactionBegin(Transaction transaction) {
    try {
      switch (transaction.getContextState()) {
      case Transaction.TRANSACTION_BEGINNING:
        transaction.setAttachment(new Store());
      case Transaction.TRANSACTION_IDLE:
      case Transaction.TRANSACTION_ACTIVE:
        transaction.continueExecution();
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
```

Upon commit, check if updates have been performed and rollback the transaction.

```
  public synchronized void handleTransactionCompletion(Transaction transaction) {
    Store state = (Store) transaction.getAttachment();

    Utils.info("Number of tuples written is " + state.writes() + ".");

    try {
      switch (transaction.getContextState()) {
      case Transaction.TRANSACTION_COMMITTING:
        if (state.writes() != 0) {
          transaction.cancelExecution();
        }
        break;
      case Transaction.TRANSACTION_COMMITTED:
      case Transaction.TRANSACTION_ABORTING:
      case Transaction.TRANSACTION_ABORTED:
        transaction.continueExecution();
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
```

```
  }

  public void handleObjectSetWrite(ObjectSet objSet) {
    try {
      switch (objSet.getState()) {
      case ObjectSet.PIPELINE_PROCESSING:
        Store state = (Store) objSet.getRequest().getTransaction()
            .getAttachment();
        while (objSet.next()) {
          state.contWrites++;
        }
        objSet.continueExecution();
        break;
      case ObjectSet.PIPELINE_PROCESSED:
        objSet.continueExecution();
        break;
      case ObjectSet.PIPELINE_ERROR:
        objSet.continueExecution();
        Utils.cleanUp(new SQLException("ObjectSet - WriteSet Error."));
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
```

The main loop waits for a connection from the primary and then receives updates and applies them using JDBC connection. Notice that this is very naive in the sense that sequential application of updates is performed. This is done to improve the readability of the sample and can easily be done using the JDBC interface with a connection pool.

```
  public void run() {
    try {
      Utils.info("Connecting to primary.");
      Socket sock = new Socket("localhost", 12345);
      Utils.info("Connected to primary.");

      ObjectOutputStream outstr = new ObjectOutputStream(sock
          .getOutputStream());
      ObjectInputStream instr = new ObjectInputStream(sock
          .getInputStream());

      while (true) {
        Utils.info("Waiting for update.");
        LinkedList update = (LinkedList) instr.readObject();
        Utils.info("Update received, applying.");

        try {
          Statement s = conn.createStatement();
          Iterator i = update.iterator();
          while (i.hasNext()) {
            String up = (String) i.next();
            Utils.info(up);
            s.executeUpdate(up);
          }
          conn.commit();
          s.close();
        } catch (SQLException sqle) {
          Utils.error("Failed update.", sqle);
          outstr.writeBoolean(false);
          break;
        }
        Utils.info("Sending acknowledgment.");
        outstr.writeBoolean(true);
        outstr.flush();
      }
      Utils.info("Backend refused update.");
    } catch (Exception ex) {
      Utils.error("Disconnected from master.", ex);
      Utils.cleanUp(ex);
    }
  }
```

This creates a connection to inject remote updates into the backup replica.

```
  public void handleDatabaseStartup(Database database) {
    try {
      switch (database.getContextState()) {
      case Database.DATABASE_STARTING:
        database.continueExecution();
        break;
      case Database.DATABASE_UP:
        conn = database.getDataSource().getConnection();
        database.continueExecution();
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
```

This disables reflection of processing of SQL statements issued when applying updates. This ensures that only locally executed statements get reflected.

```
  public void handleConnectionStartup(gorda.db.ConnectionContext connection) {
    try {
      switch (connection.getContextState()) {
      case gorda.db.ConnectionContext.CONNECTION_STARTING:
        ConnectionMetaInfo meta = connection.getConnectionMetaInfo();
        if (meta != null && meta.getUserId() != null
            && meta.getUserId().equals(user))
          connection.setNotificationIgnored(true);
      case gorda.db.ConnectionContext.CONNECTION_UP:
        connection.continueExecution();
        break;
      }
    } catch (SQLException ex) {
      Utils.cleanUp(ex);
    }
  }
}
```

A License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. **"Distribute"** means to make available to the public the original and copies of the Work through sale or other transfer of ownership.

d. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

e. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the

public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections; and,

b. to Distribute and Publicly Perform the Work including as incorporated in Collections.

c. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   ii. **Waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

   iii. **Voluntary License Schemes.** The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

   The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Adaptations. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(b), as requested.

b. If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the

Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. The credit required by this Section 4(b) may be implemented in any reasonable manner; provided, however, that in the case of a Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

c. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

c. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

d. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

e. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.