External procedures

Eugeney Putilin, Roman Rokytskyy

Motivation

Stored procedure is powerful mechanism to process data on the server side. Selectable procedures make it even more powerful, eliminating in many cases need to store temporary data in the persistent storage. However PSQL language provides only basic computational capabilities, if more complex computations are needed, developers have to mix PSQL code with UDF invocations. However there exist situations where such approach does not work at all, or code becomes to complex to maintain. In order to overcome such limitations we propose concept of external procedure.

External procedure is a piece of code that does not belong to the engine[1] and can be accessed in the same way as ordinary stored procedure. Code of the external procedure is usually stored outside the engine, but can be also stored in some internal table. Place where code is stored should not influence the way external procedure is used.

Declaration

External procedure is defined as following

```
CREATE PROCEDURE proc_name (param_list)
      RETURNS (return_list)
LANGUAGE module_name
ENTRY POINT 'procedure_name'


proc_name   := procedure name, valid identifier.
param_list  := param_def [,param_list]
param_def   := param_name type
param_name  := valid identifier.
type        := INTEGER | CHAR | VARCHAR | ...
return_list := param_def [, param_list]
module_name := valid identifier, more about this below.
proc_name   := identifier, understandable by the specified module.
```

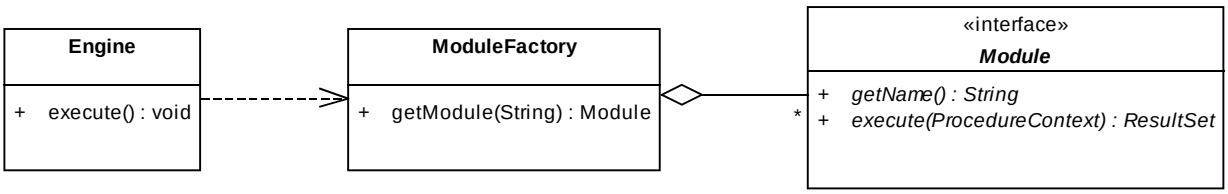
After procedure is created, user can access it using conventional operators `EXECUTE PROCEDURE` and `SELECT ... FROM`.

Interesting part is MODULE_NAME. It defines an ID of a plug-in that is responsible for executing specified procedure. Engine is no longer responsible for interpreting BLR code, rather delegates the call to the processor. Name of the module defines a processor that will continue execution.

Modules

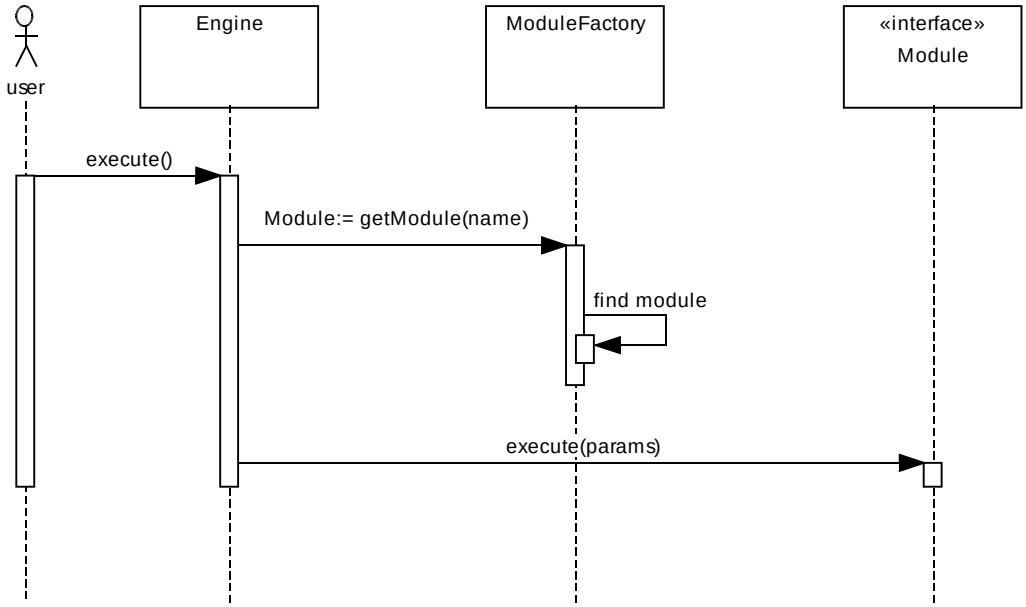
Since we want to be able to plug arbitrary procedures, we need an interface how modules can be plugged in. Simplified view of the module manager is following:

1 Phrase „does not belong to the engine“ means that engine was not linked with this code statically during build process.



Engine has access to the module factory that is responsible for module management. It is responsible for discovering the modules, for their initialization/shutdown. Currently it is not clear if `ModuleFactory` should act as a dispatcher for the modules or just as a factory. This is implementation issue, therefore we leave it open. However, we must note that call to `Module.execute(ProcedureContext)` should be executed in sandbox-like manner. Whether it will be done inside the engine or `ModuleFactory` depends on the implementation.

Typical usage scenario is following:



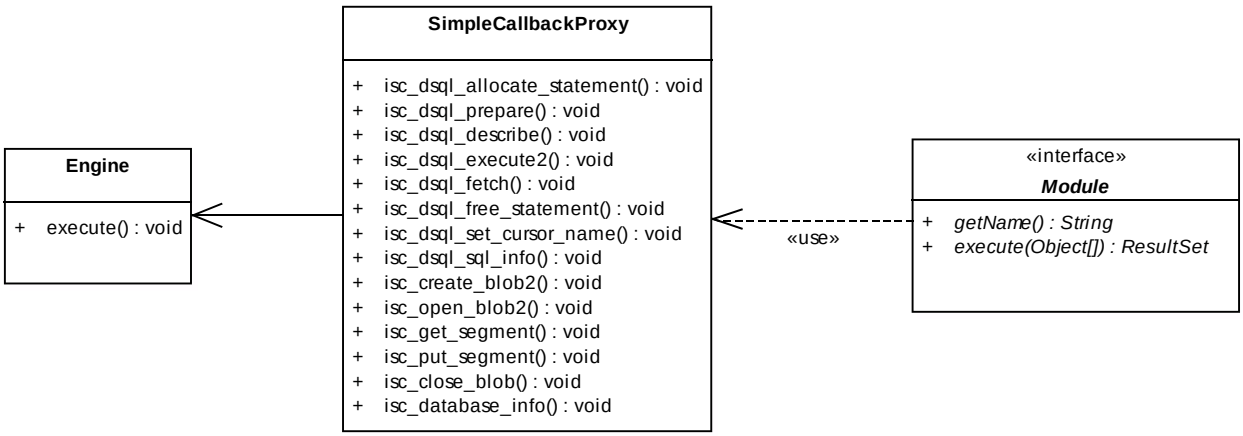
User submits `EXECUTE PROCEDURE` or `SELECT ... FROM` statement to the server. Engine discovers that procedure being executed is external one and asks module factory for a module with the name equal to the one specified in MODULE_NAME part of procedure declaration. If module factory finds corresponding module, it returns a reference to the `Module` interface, and `Module.execute(ProcedureContext)` method is invoked.

Accessing database from external procedure

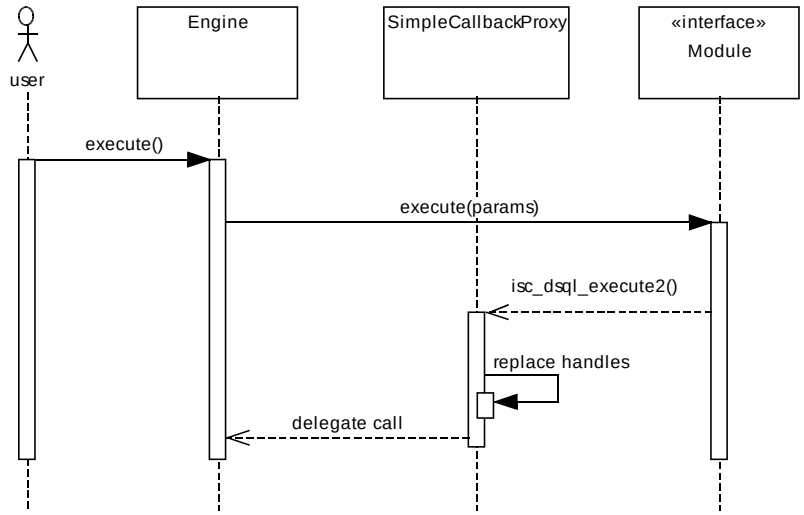
One of the key features of ordinal stored procedure is possibility to manipulate data in a database within the same transaction context. Contrary UDF does not have any possibility to access database, no API is currently available. It is clear that for external procedures we have to provide a possibility to call the engine during execution.

It seems to be natural to provide usual external API also for external procedures, or at least part of it (we will use current ISC API, however, in the light of recent post by Jim Starkey about internal JDBC-like API, it would be preferable way to work with the engine). However, almost all calls require passing either valid database handle or transaction handle or both. Where do we get them? We see two possibilities: all callbacks are executed within the same transaction, in this case we can fill correct values within the callback method, or we pass current database and transaction handles into the external procedure and require procedure to pass correct values. Both approaches has their drawbacks and advantages.

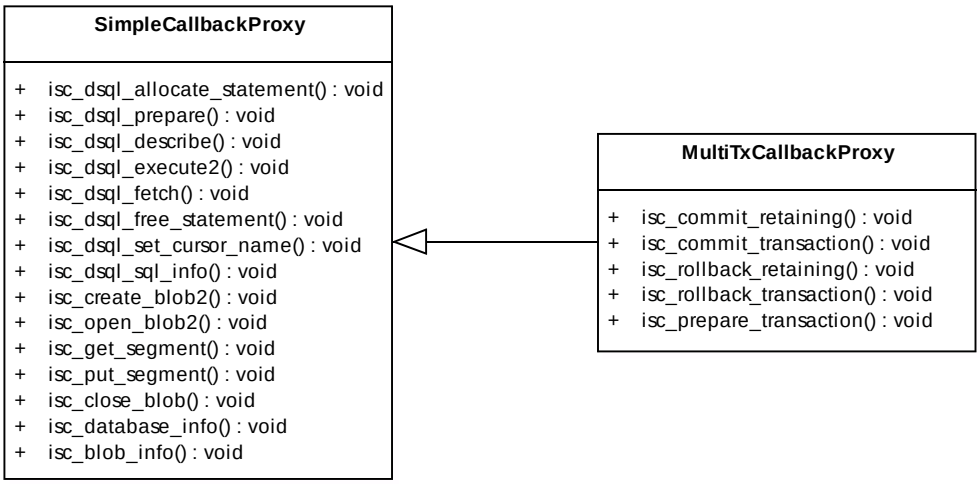
Single transaction mode



In this case module calls methods from a proxy class that provides a limited set of methods, namely methods that allow executing statements and obtaining information about statements and database. Methods responsible for database attachment management as well as transaction management are not available. Probably for the sake of API completeness they can be declared, but should return an error status code. Interaction between engine, proxy and module can be the following:



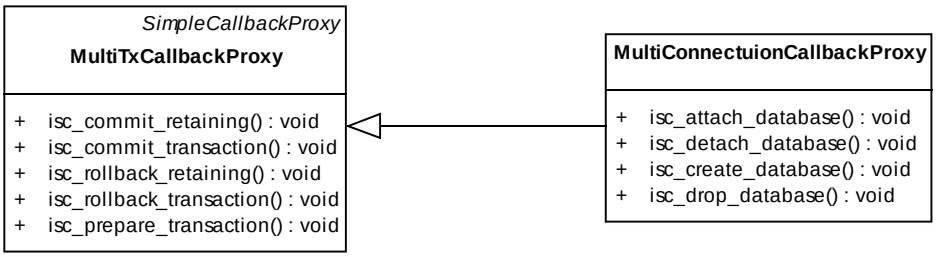
Multiple transactions



In case when we allow external procedure in addition to the current transaction context to start new

transaction using the same database attachment.

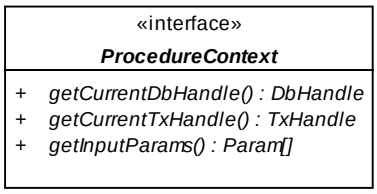
Multiple connections



Here we allow opening connections to other databases.

Procedural context

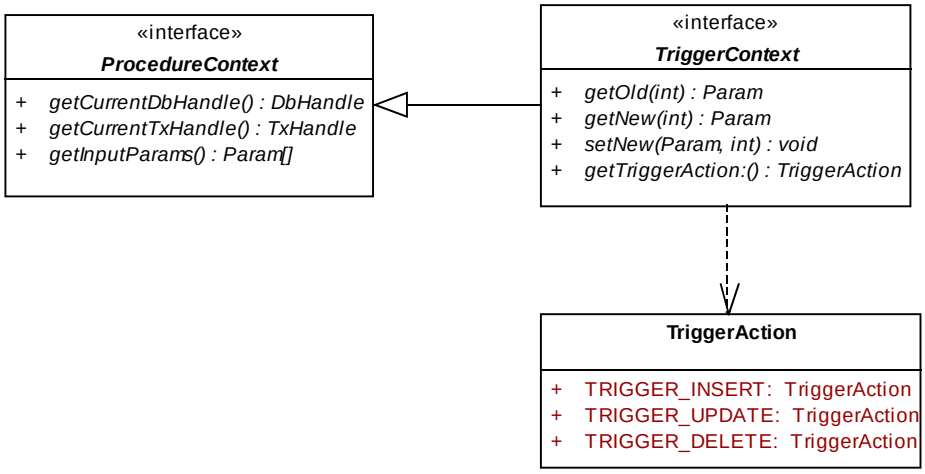
In order to provide an external procedure access to the context data, the following interface is used. Instance of `ProcedureContext` is passed as parameter into `execute(ProcedureContext)` method of `Module` interface.



It provides access to the current database and transactions handles and allows procedure to execute requests within a transaction context it was invoked in. Diagram shows that methods return instances of classes `DbHandle` and `TxHandle`. Actual implementation will have to return objects representing handles (currently int values, but this is subject to change). Additionally procedural context provides information about the input parameters. Again here we do not specify what `Param` type is, in current implementation it can be just `XSQLVAR` instance. However we leave this open for more sophisticated implementations.

Triggers

Concept of procedural context provides us an infrastructure that would allow using external stored procedures as triggers.



In this case procedural context also provides information about the old and new values. Also it

provides information about the action for which trigger was fired. Diagram uses type-safe enumeration pattern common in Java language, actual implementation might differ.

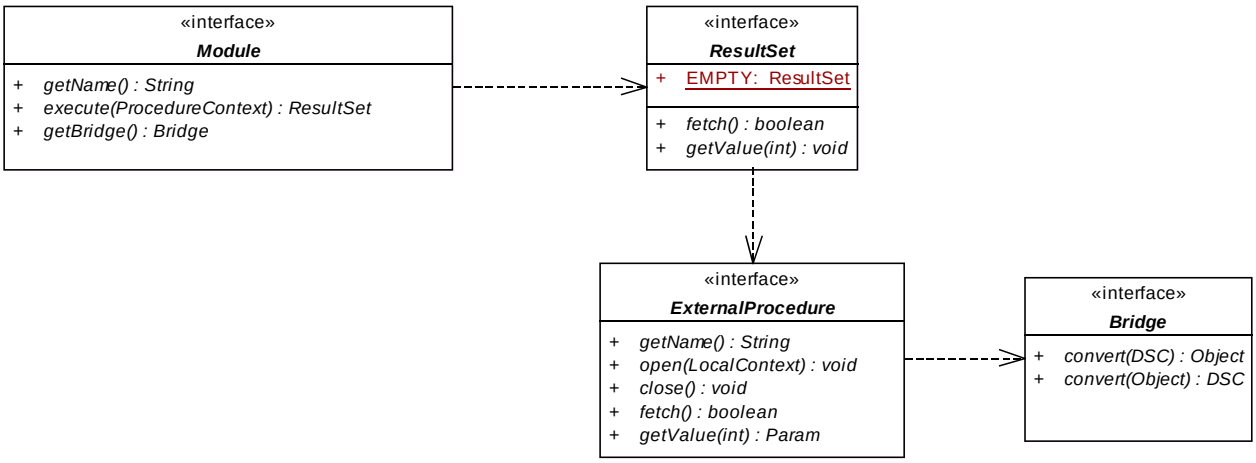
Triggers then are defined as

```
CREATE TRIGGER trigger_name FOR table
      {ACTIVE | INACTIVE}
      {BEFORE | AFTER}
      <multiple actions>
      [POSITION number]
      LANGUAGE module_name
      ENTRY POINT 'procedure_name'

<multiple actions> :=
      <single action> [OR <single action> [OR <single action>]]

<single action> := {INSERT | UPDATE | DELETE}

proc_name    := procedure name, valid identifier.

module_name := valid identifier, more about this below.

proc_name    := identifier, understandable by the specified module.
```

Main difference from the common syntax is that instead of PSQL trigger body we define a module and procedure name that will execute it.

Output values and selectable procedures

As it was already described in a “Modules“ chapter, module returns `ResultSet` instance as the result of execution. `ResultSet` interface allows us navigating the results[2]. Also it provides access to the values in the current row. All data are accessed by position, not by name. Method `fetch()` returns either `true` or `false`. True means that there are more records in the result set and we can continue fetching data from it.

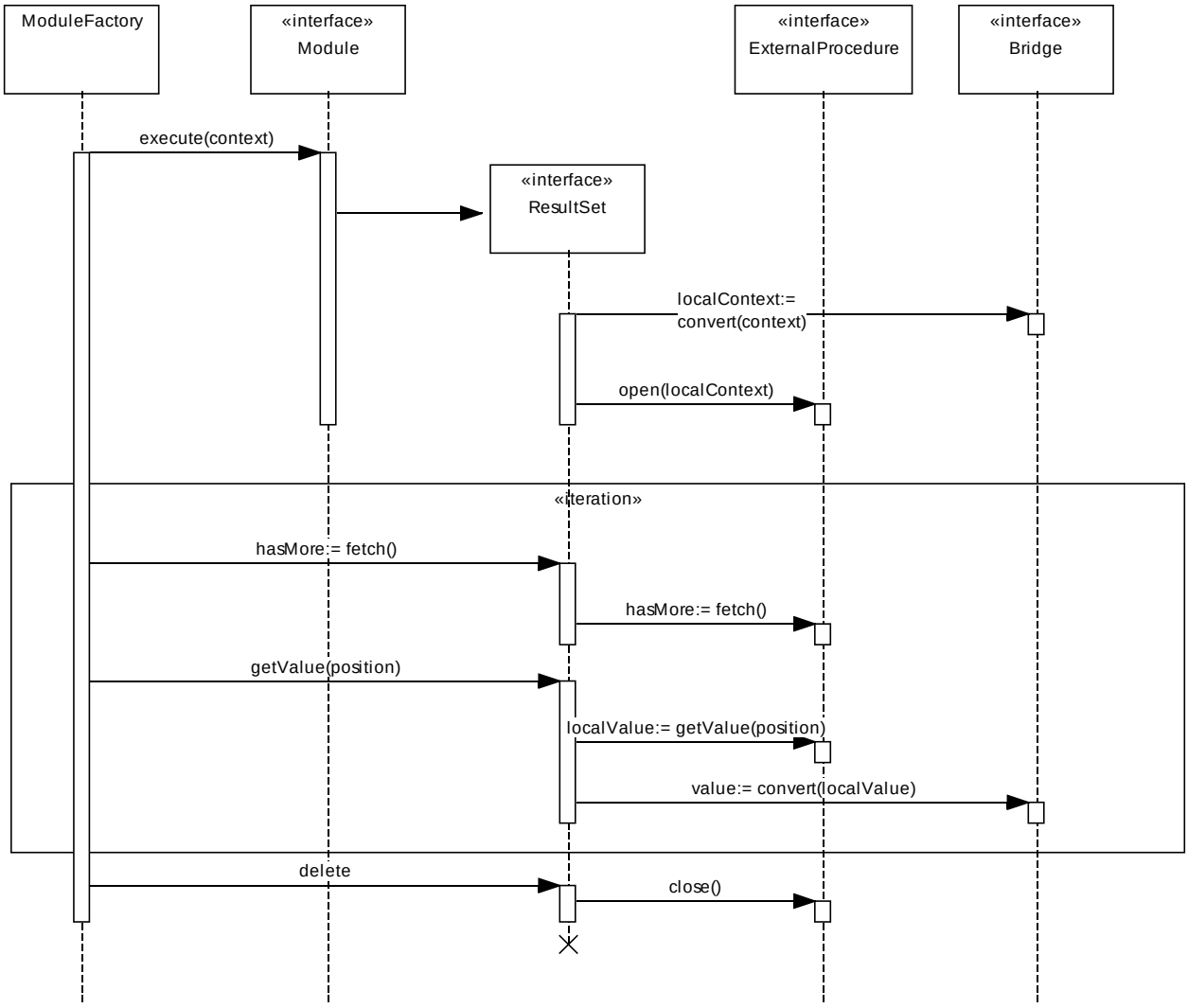


Procedures with no output parameters result in `ResultSet.EMPTY` instance (it is normal result set implementation, but `fetch()` operation always returns `false`). It is responsibility of the `Module` to convert interface for the actual procedure implementation into the `ResultSet`. Non-selectable procedures return `ResultSet` implementation that has exactly one row.

Execution of stored procedure consists then of three phases: initialization, data fetching and cleanup. When `ModuleFactory` calls `Module.execute(...)` method, it initializes procedure (method `ExternalProcedure.open()`). Module implementation creates an instance of `ResultSet`,

2 We actually model existing interface for the procedures in RSE module that uses open_procedure(), fetch_record() and close_procedure() methods.

a proxy that will control execution of the external procedure and perform type conversion between engine's internal types and types understandable by the procedure using `Bridge` implementation. After creation, instance of `ResultSet` opens external procedure and passes converted context object, procedure performs internal initialization. After this module factory starts looping by calling `ResultSet.fetch()` method until it returns `false`. This call triggers some execution within the procedure itself, `ResultSet` instance converts values returned by procedure into corresponding types of the engine. When no more data to return is available, `ModuleFactory` deletes `ResultSet` object, destructor of which calls `ExternalProcedure.close()` to perform a cleanup.



It seems that from the integration point of view it would be natural to integrate external procedures in the RSE module by using either existing `rsb_t` type or defining new type for exactly this case.

Acknowledgments

We would like to thank Paul Ruizendaal for his invaluable help during preparing this paper. His input helped us a lot in shaping this paper to a readable form.