Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Hybrid proof-of-concept for PostgreSQL

GORDA Deliverable D4.5

Due date of deliverable: 2006/09/30
Actual submission date: 2006/09/30
Revision date: 2006/12/30

Start date of project: 1 October 2004 Duration: 36 Months

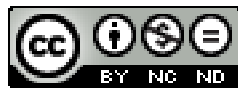
Universidade do Minho

Revision 1.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contributors

Alfrânio Correia Júnior
José Pereira
Luís Soares
Ricardo Vilaça
Rui Oliveira



(C) 2006 GORDA Consortium. Some rights reserved.
This work is licensed under the Attribution-NonCommercial-NoDerivs 2.5 Creative Commons License.
See http://creativecommons.org/licenses/by-nc-nd/2.5/legalcode for details.

Abstract

This document describes the mapping of the GORDA Architecture and Programming Interfaces to the open-source PostgreSQL database management system. It combines in-core components with middleware components to tackle specific technical challenges in the PostgreSQL architecture as well as to ease adoption by the PostgreSQL community, thus reflecting current experience and roadmap for ongoing work on the prototype.

Chapter 1

Introduction

This chapter describes the scope of the PostgreSQL prototype (PostgreSQL/G), regarding its objectives and the components of the GORDA Architecture and Programming Interface (GAPI) actually being implemented.

1.1 Objectives

The mapping of the GAPI to the open source PostgreSQL database management has the following goals:

- Allow implementation of a wide range of group-based database replication protocols thus requiring an extensive mapping of the GAPI.
- Allow full compatibility with all PostgreSQL clients interfaces and tools.
- Demonstrate the feasibility of efficiently extracting read sets.
- Modifications to the core PostgreSQL source code although unavoidable, should be as little intrusive as possible in order to facilitate dissemination by proposing patches for acceptance by the PostgreSQL open source community.

1.2 Components

The mapping to PostgreSQL is outlined in Figure 1.1. Components that have been introduced are shown as gray boxes. The GAPI is implemented mostly as middleware components that build on existing interfaces, such as triggers and JDBC. Missing functionality is provided by a small set of server modifications that extend such interfaces as necessary. Integration of components built using C and Java is achieved using the PL/J package [5]. In detail, the mapping introduces the following components:

Trigger Patch Adds support for life-cycle triggers, transaction triggers and statement triggers. This functionality can be used outside the GAPI.

Read-Set Patch Patch to extract read sets which might be used to provide serializability in a centralized PostgreSQL, thus being useful outside the GAPI.

PL/J Proxy Support for Java procedure calls that are routed to an external server process.

PL/J Server Java virtual machine that houses Java procedures and provides support for server-side JDBC. This acts as the container for both the reflector implementation as well as for replication protocols.

Reflector Implementation Java code implementing the GORDA Programming Interfaces.

Communications between applications and the PostgreSQL are performed using a legacy driver (e.g. JDBC driver) provided by a database vendor/supplier thus ensuring compatibility with all existing clients and tools.

As shown in Figure 1.2, the current mapping of the GORDA Programming Interfaces includes all contexts and the two main stages.

The resulting hybrid implementation strategy is required to achieve the goal of imposing minimal modifications into the PostgreSQL server to foster its acceptance, as a full in-core implementation would force the inclusion of a Java environment.

1.3 The PostgreSQL System

PostgreSQL [1] is a fully featured database management system distributed under an open source license. It is included in most Linux distributions as well as in recent versions of Solaris. Commercial support and numerous third party add-ons are available from multiple vendors.

PostgreSQL supports a number of standards, namely, ANSI SQL and JDBC. Since version 7.0, it provides a multi-version concurrency control mechanism supporting snapshot isolation.

The PostgreSQL server is written in C and has been ported to multiple operating systems. It is implemented as a traditional *forking server*, in which each client connection is serviced by a dedicated operating system process. Server processes hold shared state and communicate using explicitly shared memory blocks and semaphores, namely, using System V IPC in Unix systems. In detail, a daemon process ("postmaster") receives connection requests and spawns a backend process ("postgres"). A single postmaster manages a collection of databases on a single host called a database cluster. Clients access a database by means of a FEBE (Front End Back End) protocol encapsulated, for instance, in a JDBC driver or provided by a shared library as used by the interactive application "psql". Hence, the postmaster is always running, waiting for requests, whereas frontend and backend processes come and go.

SQL statements sent from a client to a "postgres" passes through the following stages:

- **Parser** checks for correct syntax and creates a query tree. It is built on the Lex and Yacc utilities and specifically the file "gram.y" defines a set of rules and actions that are used to build a parse tree which is then mapped to a query tree. The query tree is an internal representation used by the PostgreSQL to represent a statement sent to be processed. It is worth noticing that to introduce new commands or change syntax one needs to edit this file.

- **Rewriter** takes the query tree previously created and applies any available rule to perform transformations such as those specified by views. In other words, when a SQL is performed on a view, the rewriter redefines the SQL in order to access the base relations given in the view definition.

- **Optimizer** uses the rewritten query tree and produces an execution plan that will be handled by the executor. The optimizer uses statistic information on the data in order to chose the cheapest plan in terms of disk access and cpu usage. In detail, its task is to create an optimal execution plan. To do so, it combines all possible ways of scanning and joining the relations that appear in a query and evaluates costs based on statistic information maintained through histograms which are a mixture of end biased and equidepth histograms.

- **Executor** processes the execution plan produced by the optimizer. This plan is a tree structure that should be recursively executed to extract the required set of rows (i.e., data). This is essentially a demand-pull pipeline mechanism. It is worth noticing that data are retrieved from relations by two means: index or sequential scans, which means that in order to retrieve data either an index or a direct access to a relation is used.

This structure that might resemble any database system is used to organize the PostgreSQL source code as depicted in Figure 1.3:

- **bootstrap** creates initial template database via initdb.
- **main** passes control to postmaster or postgres.
- **postmaster** controls postgres server startup/termination.
- **libpq** implements the FEBE protocol.
- **tcop** traffic cop, dispatches request to proper modules.
- **parser** converts SQL statement to query tree.
- **optimizer** creates an execution plan.
- **commands** processes SQL statements such as "create trigger" that do not require complex handling.
- **catalog** defines system catalog manipulation.
- **storage** manages storage information such as buffers, pages, large objects, files, the ipc mechanism, locks and deadlocks.
- **executor** processes complex node plans produced by the optimizer.
- **access** provides various data access methods along with transaction, sub-transaction and logging functions.
- **nodes** defines structures and provides utilities to manipulate them that are used throughout the other modules.
- **rewrite** implements the rewrite subsystem.

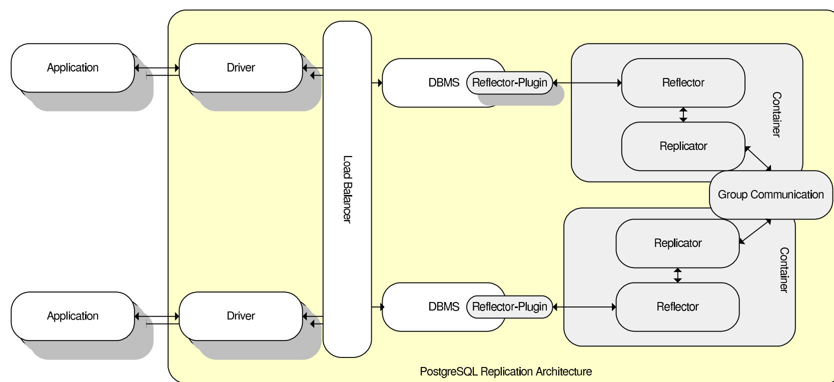


Figure 1.1: PostgreSQL Replication Architecture

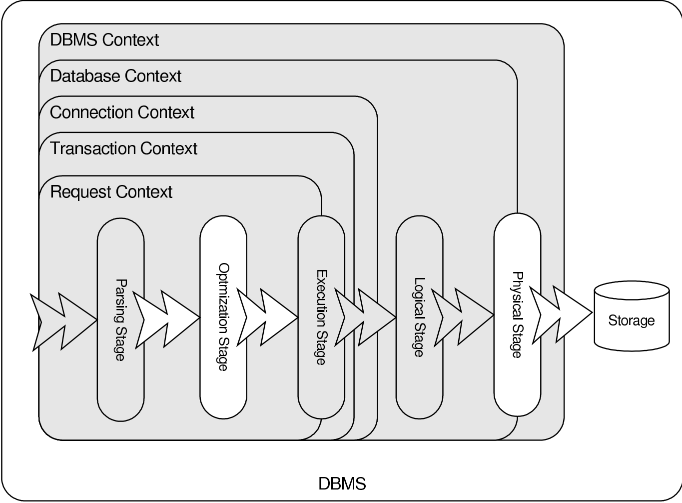


Figure 1.2: PostgreSQL: Contexts and Stages

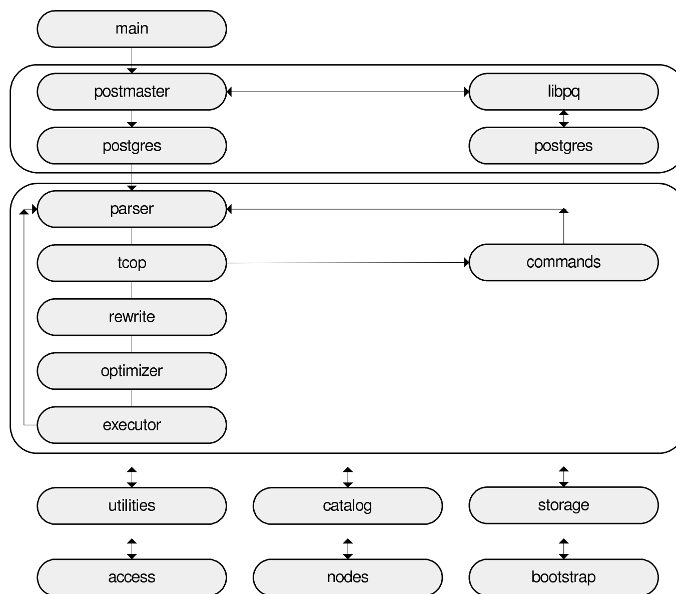


Figure 1.3: PostgreSQL Internals

Chapter 2

Server Modifications

In this chapter we describe the modifications to the PostgreSQL server. These implementations, namely triggers, transaction priority and read sets are useful by themselves even out of the scope of the GAPI.

2.1 Triggers

The proposed global triggers create hooks that capture connection and transaction life cycle events, as well as statements as issued by the client. Namely, connection start up and shutdown, transaction begin, commit, and rollback and statement execution. They are implemented as global triggers, associated with a database instead of a relation.

The start up hook is called when a client connects to PostgreSQL. On disconnection, the shutdown hook is called. During normal operation, whenever a transaction begins (implicitly or explicitly) the begin hook is called. The commit hook is called just after the client requests commit (implicitly or explicitly) and when the trigger function can still rollback the transaction. The rollback trigger is called whenever a transaction is aborted (implicitly or explicitly). The statement trigger is called when a request is made by a client.

The usage of global triggers closely follows the usual steps required by common triggers in PostgreSQL:

- Develop a function for each of the desired events using any supported procedure language-
- For each database, bind the function to the desired event by using the following set of commands (note the absence of a relation name):

```
create trigger <name> for { startup | shutdown |
   begin transaction | commit transaction |
   rollback transaction | statement }
   execute procedure <func> ( <funcargs> )

drop trigger <name> { cascade | restrict }

alter trigger <oldname> rename to <newname>

comment on trigger <name> is 'text comment'
```

If more than one trigger is defined for the same event, they will be invoked in alphabetical order of their names.

2.2 Priority Mechanism

A priority mechanism eases the task of ensuring that a commit order can be enforced regardless of competing unrestricted transactions being submitted by other connections. In detail, this requires that in the event that a high priority transaction is about to block due to a lower priority transaction, the latter is aborted.

To enable such features in PostgreSQL, we augmented the lock manager, the deadlock mechanism and transaction properties allowing to define a transaction as master as follows:

```
SET TRANSACTION MASTER
```

Considering two different transactions: t_1 and t_2, where t_1 is a high priority transaction that is about to block due to locks held by t_2. t_1 finds which process is holding a lock on behalf of t_2 and notifies it, requesting that t_2 aborts and therefore releases its locks. Then, transaction t_1 is put in a queue and waits for t_2. If t_2 is a high priority transaction nothing is done. Otherwise, t_2 aborts its execution by identifying itself as part of a deadlock. Due to the unreliable nature of signals such notifications are done periodically when the deadlock routine checks if a transaction that is waiting for a lock is not part of a deadlock cycle.

Our current implementation defines only two levels of priority, namely, "master" and "normal" (i.e., default). Once a transaction is defined as master, its priority is only brought to normal, after being rolled back or committed. It is worth noticing, however, that such levels might be easily extended to more than two and might be easily used to improve performance as described in [4].

2.3 Capturing Read-Sets

Read set is a key component while developing optimistic replication protocols as it is used in the certification procedure providing the ability to detect violations in consistency criteria such as serializability or repeatable reads. Its definition is highly depend on the adopted criterion and currently is subject of research [2, 3].

The current simplistic implementation is achieved by modifying the seqsan and indexscan functions in the executor module in PostgreSQL such that a pointer to every tuple that is read while processing a transaction is stored in an in-memory structure. This structure is then marshalled and transmitted to the external server upon commiting the transaction.

In order to avoid excessive memory consumption, a threshold is set upon which all tuples collected for a given table are flushed and the table identifier is used. In practice, this means that the entire table is considered to be in the read-set.

This simplistic approach is currently being improved by implementing filtering strategies that improve performance while ensuring specific consistency criteria [2, 3]. Namely, by extending and introducing changes in the executor, in particular, in the index and sequential scan functions along with a new pipeline iterator created to extract tuples in particular cases.

Chapter 3

Implementation

This chapter discusses the main challenges and design decisions of the implementation in PostgreSQL, and then describes in detail the implementation of each GAPI component.

3.1 Challenges

The major challenge in the PostgreSQL 8.1 implementation is the mismatch between the concurrency model used in the database server and the assumed multi-threaded runtime required by the GAPI. PostgreSQL 8.1, as all previous versions, uses multiple single-threaded operating system processes for concurrency. Such concurrency model would severely restrict the ability to reuse other software packages, such as group communication toolkits, which usually assume the more common multi-threaded process concurrency model.

This challenge is addressed by exposing the GORDA interfaces in a separate process, using an inter-process communication mechanism to communicate with the database server processes. Note that a similar design is used by Postgres/R, even being a PostgreSQL specific replication protocol. This is required to use the Spread group communication toolkit, which assumes the common multi-threaded process model.

A similar dilemma was faced by developers of server side binding for Java. The PL/Java proposal uses a local Java virtual machine in each PostgreSQL process, thus precluding shared Java objects. The alternative PL-J proposal, uses a single standalone Java virtual machine and inter-process communication. Development of a single multi-threaded backend has been discussed and is currently on the PostgreSQL developers to-do list.

A secondary challenge is contributing back to the open source community improvements to the PostgreSQL source code required to implement the GAPI. This requires avoiding major changes to source code while at the same time not introducing major dependencies on external packages, such as a Java virtual machine.

Besides factoring out some functionality as described in Chapter 2, we address this issue by relying on the existing PL-J package for binding PostgreSQL with Java code and on client connections using the standard JDBC driver. As described in detail in Appendix A, PL-J enables the execution of procedures written in Java, being used to deliver events generated by the reflector component inside PostgreSQL and to modify the PostgreSQL behavior as holding a transaction execution or aborting it, changing updated information and inserting new ones. Client connections are used to configure the GAPI container and to apply remote transactions

3.1.1 Improvements to PL-J

The current version of the PL-J package has several shortcomings when used to support the GAPI. Namely:

- The included build system, based on Maven, was not fully implemented and not working correctly. This was solved by converting the project to the simpler Ant build system, which is appropriate for the module's complexity.
- PL-J does not currently support all valid PostgreSQL data types (e.g., date, datetime, timestamp, float). Some of these were implemented, in both C and Java languages to suit both ends. The remaining were solved by adding an interim conversion to strings, as PostgreSQL converts them back as needed.

The resulting changes to the PL-J code base were contributed back to the original authors, although so far it was not yet possible to determine whether these will be included in a future version.

3.2 Component Mapping

These new triggers are used to capture events inside PostgreSQL, disseminating information on the following contexts and stages: Connection and Transaction contexts, Parsing, Optimization, Execution and Logical Storage stages. Events related to the DBSM and Database contexts are not captured inside PostgreSQL but are artificially generated by the PL-J server. This is due to the fact that PostgreSQL should be able to be started without a reflection infrastructure in order to have administrative tasks performed such as rebuilding statistics and organizing physical storage. PostgreSQL does not provide the ability to start, stop and resume databases individually and thus such events are artificially generated by the PL-J server too. Furthermore, in our current implementation, we assume that a new request is assigned to each statement.

Configuring which events are notified is a critical task in this prototype as an event invokes a remote method. For that reason, during start up the PL-J server turns on the appropriate triggers according to the set of registered callbacks. Basically, this either executes a "create trigger" or "drop trigger" command in order to turn on triggers that are going to be used or turn off the others that are not necessary.

3.2.1 Support and implementation of the DBMS module

The DBMS context is independently started without any synchronization between the modified PostgreSQL and the PL-J server. This feature enables administrative tasks to be carried on even if reflection is not initiated. To do so, one needs administrative privileges inside PostgreSQL. For instance, one might create, populate and configure databases that might or might not be reflected. For this reason, the DBMS startup is not propagated by the PostgreSQL and it is internally generated by the PL-J server. By contrast, the DBMS shutdown is captured and notified by the PostgreSQL.

Meta information is defined by the PL-J server. However, it is worth noticing that the DBMS identification is assigned upon start up of the PL-J server and should be unique. Its assignment and management is not done automatically and should be done by the administrator. There is no problem in reusing ids as long as distinct systems have distinct ids.

Canceling a DBSM startup invokes $System.exit()$, thus aborting the PL-J server but the PostgreSQL remains operational to administrative tasks.

3.2.2 Support and implementation of the database module

In PostgreSQL, one might create distinct databases that are used to organize information in an enterprise. In contrast to other systems however, PostgreSQL initiates every database simultaneously during its own start up and does not allow to pause or stop databases. The database start up and shutdown events are not therefore generated by the PostgreSQL but internally by the PL-J server.

Similar to DBMS meta data, meta information on databases are made available by the PL-J server and the database identification is assigned upon its start up. In particular, the $DatabaseMetaInfo$ is wrapper to the class $Database\ MetaData$ in the PostgreSQL JDBC driver. The methods $panic()$ and $cancel\ Startup()$ invoke $System.exit()$ thus aborting the PL-J server but the PostgreSQL remains operational to administrative tasks.

By calling $getDatabaseSource()$ one has access to a Server Side JDBC. If the url used to acquire a connection is $jdbc{:}default$, statements are processed in the context of a transaction if there is one, otherwise an exception is raised. In the former case, communication occurs through a PL-J interface and in the latter, a standard PostgreSQL JDBC Driver is used.

The methods related do transfer the entire database image, used for recovery purpose, are implemented using the pg_dump and $pg_restore$ tools.

3.2.3 Support and implementation of the connection module

This is partially implemented as due to limitations in the PL-J the only character-set allowed is the "*en_US*". PostgreSQL creates a new process for each user and thus we assign the process identification as a connection identification. Connection events are defined as lifecycle triggers in the PostgreSQL and then mapped to procedures in the PL-J server. One might allow a connection to proceed or cancel it, by calling *continueStartup()* or *cancelStartup()*, respectively. In the latter case, the postgres backend is safely ended. The ability to cancel a connection is quite interesting when a database is doing recovery. As the PostgreSQL might be started independently from the PL-J server, it is important to abort any activate while the database is synchronizing with remote replicas, allowing connections to proceed when the recovery is done and thus guaranteeing that clients access a consistent database replica.

3.2.4 Support and implementation of the transaction module

This feature is completely implemented as it plays an key role while holding a transaction execution to execute replication activities such as propagating conflict classes or updates among replicas.

We identify a transaction in the PostgreSQL/G by assigning the transaction identification given by the PostgreSQL. Transaction events are captured by lifecycle triggers and then mapped to procedures in the PL-J server. At any time, one might abort a transaction by calling *cancelExecution()*. In particular, before a commit request is issued besides aborting or allowing a transaction to continue, one might still update, delete and insert information on its behalf. This feature is exploited in our prototype to log information used during recovery.

3.2.5 Support and implementation of the request module

A new Request is generated for each Statement. Building a request with batch information would require changes in the FEBE thus reducing possible acceptance by the PostgreSQL community of the changes done in the PostgreSQL. Similar to other modules, it is possible to cancel a request by calling *cancel-Request()*. It is worth noticing however that while invoking this method, one aborts the transaction on behalf of what the request is being processed.

3.2.6 Support and implementation of the statement module

This is partially implemented. Statements cannot be changed in the current version.

Statements are captured by a trigger and sent to the PL-J server. In order to avoid sending any type of statement the trigger exploits information provided by the Parsing stage. For instance, select statements should not be propagate when byzantine faults are not handled. In particular, we believe that administrative statements such as those used to rebuild indexes and re-organize databases should not be replicated and should be executed individually per database. Thus, by exploiting the information provided by the Parsing stage we can easily determine whether a statement should be or not replicated. Along with the raw statements and its type is captured and transmitted information on statistics, i.e. costs and number of tuples affected by the statement. This might be used for instance to decide to use an active replication protocol thus avoid flooding the network with updates.

One drawback of our current approach is that it does not handle non-deterministic clauses in a statement such as *now()* as it is applied after the optimization stage. To circumvent this, one needs to replace the execution plan generated by the optimizer in order to replace the non-deterministic clauses.

3.2.7 Support and implementation of the execution module

This is partially implemented. Result sets attributes cannot be changed in the current version.

PostgreSQL provides the ability to capture modified information by means of insert, delete and update triggers and is augmented to provide information on reads as described in Section 2.3

Any update can be captured by means of triggers provided by the PostgreSQL and easily transmitted to the PL-J server. Nonetheless, for performance reasons, we provide two optimizations. First, we have meta information defined per database that specifies which relations should have their updates captured as it is common knowledge that triggers slowdown database performance. Second, for some replication protocols it is wise, regarding performance, to store transaction's updates in an in-memory structure (e.g., hash structure) and transmit them on commit request, thus reducing the number of interactions between the PostgreSQL and the PL-J server. One can easily define however whether an update should be immediately transmitted or not by assigning different functions to a trigger.

3.2.8 Support and implementation of the storage module

The Logical Storage is used only for recovery purposes and, in doing so, its events are not notified but rather an interface to browse its information is made available.

PostgreSQL maintains a transaction log `xlog` but does not have a direct way of read it. Currently we are using the [7] tool that is a utility for reading a PostgreSQL transaction log segment or set of segments. Xlogdump is used in a context of an on-line backend because it needs a connection to a backend to translate the oids and output the names of the database objects. This tool permit to extract the transactions info: xid and status (aborted or committed), and to build fake statements that produce the physical changes found within the xlog segments, currently supporting INSERTs, UPDATEs and DELETEs.

Chapter 4

Evaluation

In this chapter we evaluate the proposed implementation of the GAPI in PostgreSQL.

4.1 Lines of Code

We start by making and informal evaluation of the effort required to implement the GAPI in PostgreSQL by counting the lines of code modified relatively to the size of the original code base. The resulting statistics are the following:

- the size of PostgreSQL is 667586 lines;
- the PL/J package adds 7574 lines of C code and 16331 of Java code;
- 21 files changed by inserting 569 lines and deleting 152 lines;
- 1346 lines of C code were added in new files;
- 11512 lines of Java code added in new files.

4.2 Performance

In this section we evaluate the performance in order to assess the overhead introduced. It is important to evaluate also the overhead of the introduced changes when not in use, which if not negligible is a major obstacle to the adoption of the proposed architecture.

We use the workload generated by the industry standard TPC-W benchmark [6]. TPC-W defines an Internet commerce environment that resembles real world, business oriented, transactional web applications. The relatively heavy weight transactions of TPC-W make CPU processing the bottleneck. In addition, we use only the Ordering Mix, which has 50% read-only transactions and 50% update transactions. The average size of an update transaction write-set in TPC-W is 275 bytes.

We used 100 emulated browsers. While providing a realistic amount of concurrency, this does not overload the server and thus latency closely reflects processing overhead. If a very small concurrency level was used, concurrency bottlenecks would not be noticed. If a very large number of concurrent clients was used, latency would show mainly contention and not overhead.

In addition, we tested the following scenarios:

Unmodified DBMS This is the original DBMS, without any modification, serving as the baseline.

	Mean latency (ms)	Std. Dev.	# Samples
PostgreSQL	1.766	1.882	50464
PostgreSQL + patch	1.922	1.430	50574
PostgreSQL + without write-set	2.718	1.501	50528
PostgreSQL + all listeners	3.016	1.884	50554

Table 4.1: Benchmark results.

DBMS + patch This is the modified DBSM, as described in the previous section, but without any meta-level objects and thus with all reflection disabled. Ideally, this does not introduce any performance overhead.

DBMS + without write-set This is the modified DBMS with listeners registered for transactional events and statements. This means that each transaction generates at least 3 events.

DBSM + all listeners This is the modified DBMS with listeners registered for transactional events, statements, and all modified tuples. This causes a variable number of meta-level events allowing the capture of all modifications.

The results are presented in Table 4.1. When no meta-level objects are configured, the overhead is negligible. However, the impact of registering meta-level objects is noticeable, as this causes several round-trips to the external PL/J server process. This is most notable when collecting the write-set. It is however acceptable, especially as the PostgreSQL architecture makes it the worst case scenario for implementing the proposed interface.

Bibliography

[1] PostgreSQL. http://www.postgresql.org, 2006.

[2] A. Correia and R. Oliveira. Thrifty read sets for resilient database replication. Technical report, Departamento de Informática, Universidade do Minho, 2005.

[3] A. Correia, L. Soares, J. Grov, R. Oliveira, J. Orlando, and A. Sousa. Group-based replication and one-copy serializability. Technical report, University of Minho, University of Oslo and University of Salvador, 2006.

[4] D T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for oltp and transactional web applications. In *IEEE ICDE*, 2004.

[5] PL-J. http://plj.codehaus.org/, 2006.

[6] Transaction Processing Performance Council (TPC). TPC benchmark W (Web Commerce) Specification Version 1.7, 2001.

[7] XLogDump. http://xlogviewer.projects.postgresql.org/docs/, 2006.

Appendix A

PL-J

In this chapter we provide a brief overview of the PL-J package, a Java binding for PostgreSQL. We briefly describe how it integrates with PostgreSQL, how it can be used to call Java procedures, and how it supports a server-side JDBC connection.

A.1 Internals

PL-J exploits the ability of PostgreSQL to define new procedural languages. Briefly, if one wants to provide support for a new procedure language, one needs to write a native C handler. This function has the following signature:

```
PG_FUNCTION_INFO_V1(plpgj_call_handler);

Datum
plpgj_call_handler(PG_FUNCTION_ARGS) {
   ...
}
```

Then, one registers this function as a handler of the language. For the PL-J, this was done as follows:

```
create or replace function plpgj_call_handler()
   returns language_handler
as '$libdir/libHook.so', 'plpgj_call_handler'
language 'C';

create trusted language 'plj' handler
plpgj_call_handler;
```

In PL-J, this function acts as a remote procedure call proxy, redirecting the invocation the the external PL-J server. In detail, the communication between PostgreSQL and the PL-J server is done by three types of messages:

- a invocation is initiated with a *callmessage*, which might be a *procedure* or *trigger*;
- a *result* message is issued by the server as a reply;
- an *exception* that is raised when something goes wrong.

Upon receiving an invocation, the Java server matches the class and method that was invoked by a procedure or trigger and executes it. Classes may be loaded automatically avoiding to recompile the Java infrastructure provided by PL-J server. The PL-J package provides also commands to manage a repository of classes and libraries that are automatically loaded when it starts.

The *result* and the *exception* messages are generated in the PL-J server, from what one might request to execute procedures, statements and prepare statements.

Data is transmitted in binary format defined by the PostgreSQL's Front End Back End protocol. To do so, routines provided by this protocol are transparently used to do the conversion to and from a binary format. Of course, this requires that the PL-J server also provides routines to map from the binary format to Java types.

A.2 Procedures and Triggers

Any procedure specified in the new language should follow its syntax constraints and is defined as a string that will be interpreted by the handler. PL-J requires one to define which method in a class should be called when a procedure is executed. Below, we have an example of a procedure defined in PL-J:

```
CREATE OR REPLACE FUNCTION procedure()
RETURNS TRIGGER AS
'
   class=org.plj.demo.PljTest
   method=test
' LANGUAGE 'plj' IMMUTABLE;
```

A method called on behalf of a procedure must have the following signature:

```
public static <return type>
   <method>(Integer dbId, Integer conId,
            Integer tranId, <parameter>, ...)
```

where, *<return type>* is any valid Java data type including *void*, and *<parameter>* identifies a parameter passed from the PostgreSQL. The *dbId*, *conId* and *tranId* are sent automatically from the PostgreSQL and identify the database, connection and transaction contexts, respectively.

For triggers, there are two signatures. One that requires a class with the same name of a relation being updated and bean methods (i.e., sets and gets) to each attribute in that relation. In this case, the signature is defined as follows:

```
public static <relation>
   <method>(Integer dbId, Integer conId,
            Integer tranId, <relation> rel)
```

The other that uses a fixed signature:

```
public static boolean
   <method>(Integer dbId, Integer conId,
            Integer tranId, PljTriggerData data)
```

where *PljTriggerData* gives access to updated information as a Java *ResultSet*. Trigger description such as type of update is directly obtained through the *PljTriggerData*.

A.3 Server-Side JDBC

The PL-J package provides also server-side JDBC connections, which allow operations to be performed in the context of an ongoing transaction. In particular, the current implementation allows us to execute simple statements (e.g., updates and selects) and prepare statements. See the following example which is self-explanatory:

```
public static void select(Integer dbId, Integer conId,
   Integer tranId) throws SQLException {

   Connection conn = null;
   ResultSet res = null;
   PreparedStatement sta = null;

   try {
      conn = DriverManager
      .getConnection("jdbc:default:connection");
      sta = conn.
      prepareStatement("select * from test");
      res = sta.executeQuery();

      while (res.next()) {
         System.out.println("output: " +
         res.getString(1));
      }
   } catch (Exception ex) {
      ex.printStackTrace(System.err);
      System.exit(-1);
   } finally {
      if (res != null)
         res.close();
      if (sta != null)
         sta.close();
      if (conn != null)
         conn.close();
   }
}
```