# On the use of a reflective architecture to augment Database Management Systems*

Nuno Carvalho
Universidade de Lisboa
nunomrc@di.fc.ul.pt

Alfranio Correia Jr.
Universidade do Minho
alfranio@di.uminho.pt

José Pereira
Universidade do Minho
jop@di.uminho.pt

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

Rui Oliveira
Universidade do Minho
rco@di.uminho.pt

Susana Guedes
Universidade de Lisboa
sguedes@di.fc.ul.pt

**Abstract**

The Database Management System (DBMS) used to be a commodity software component, with well known standard interfaces and semantics. However, the performance and reliability expectations being placed on DBMSs have increased the demand for a variety add-ons, that augment the functionality of the database in a wide range of deployment scenarios, offering support for features such as clustering, replication, and self-management, among others. The effectiveness of such extensions largely rests on closely matching the actual needs of applications, hence on a wide range of tradeoffs and configuration options out of the scope of traditional client interfaces.

A well known software engineering approach to systems with such requirements is reflection. Unfortunately, standard reflective interfaces in DBMSs are very limited (for instance, they often do not support the desired range of atomicity guarantees in a distributed setting). Some of these limitations may be circumvented by implementing reflective features as a wrapper to the DBMS server. Unfortunately, this solutions comes at the expense of a large development effort and significant performance penalty.

In this paper we propose a general purpose DBMS reflection architecture and interface, that supports multiple extensions while, at the same time, admitting efficient implementations. We illustrate the usefulness of our proposal with concrete examples, and evaluate its cost and performance under different implementation strategies.

*Parts of this report were published in Journal of Universal Computer Science. This work was partially supported by the IST project GORDA (FP6-IST2-004758).

# 1 Introduction

The usefulness of Database Management Systems (DBMS) owes much to the standardization of client interfaces, namely, regarding the query language and call level interfaces, as well as to a precise definitions of atomicity semantics that are obtained with different configurations. However, the performance and reliability expectations being placed on DBMSs in a wider range of deployment scenarios have increased the demand for a variety of clustering [9, 7], replication [18, 27], and self-management [24] add-ons.

The effectiveness of such extensions largely rests on closely matching the actual needs of applications, in particular, on what atomicity guarantees are provided in distributed settings. Although performance and scalability can benefit from relaxed atomicity guarantees, stronger guarantees reduce the likelihood of having to resort to human intervention and thus can certainly help to achieve self-managed systems. The best tradeoff is therefore application specific and opens up a wide range of implementation and configuration options.

A well known software engineering approach to build systems with such complex requirements is reflection [22, 20]. By exposing at the interface an abstract representation of the systems' inner functionality, the later can be inspected and manipulated, thus changing its behaviour without loss of encapsulation. DBMS have long taken advantage of this, namely, on the database schema, on triggers, and when exposing the log.

Unfortunately, standard reflective interfaces in DBMSs fall short, namely, in inspecting and modifying client requests, on controlling operation scheduling, and on influencing the commit order. These are required to support the desired range atomicity guarantees in distributed settings and thus play a major role in different kinds of middleware. Reflection by wrapping the server solves some of the issues but at the expense of a large development effort and significant performance penalty [32, 9, 7]. Another alternative is to build the extension directly within the server, which has a profound impact in portability and maintainability [18].

In this paper we propose a general purpose DBMS reflection architecture and interface, that supports a number of useful extensions while at the same time admitting efficient implementations. Note that the interface described includes also some functionality that is available through standard client interfaces, but which might admit custom implementations with higher performance.

We illustrate the usefulness of the interface with concrete examples, and evaluate its cost and performance using different implementation strategies, by comparing Apache Derby and PostgreSQL prototypes with the state-of-the-art Sequoia JDBC interceptor. We also show that the implementation of such interface is a small effort and can easily be supported by DBMS vendors.

The work reported here is being developed in the context of an EU funded research project, GORDA (Open Replication of DAtabases),[1] that intends to foster database replication as a means to address the challenges of trust, in-

[1] `http://gorda.di.uminho.pt/`

tegration, performance, and cost in current database systems underlying the information society. The GORDA project has a mix of academic and industrial partners, including U. do Minho, U. della Svizzera Italiana, U. de Lisboa, INRIA Rhône-Alpes, Continuent Oy, and MySQL AB.

The rest of this paper is structured as follows. The Section 2 and the Section 3 gives some background about reflection on systems and interfaces. The Section 4 overviews the architecture and API. Section 5 presents the use cases and the Section 6 discusses the several implementations and evaluates different approaches to reflection. Finally, Section 7 concludes the paper.

## 2 Background

Logging, debugging, tracing facilities and autonomic functions, such as self-optimization or self-healing are some examples of important add-ons to DBMS that are today widely available [19]. The computation performed by such plugins is known as a computational reflection, and the systems that provide them are known as reflective systems. Specifically, a reflective system can be defined as a system that can reason about its computation and change it. Reflective architectures ease and smooth the development of systems by encapsulating functionalities that are not directly related to the application domains. This can be done to a certain extent in an ad-hoc manner, by defining hooks in specific points of a system, or with support from a programming language. In both cases, there is a need for providing a reflective architecture where the interaction between a system (i.e. base-level objects) and its reflective counterpart is done by a meta-level object protocol and the reflective computation is performed by meta-level objects. These objects exhibit a meta-level programming interface.

In this paper, we propose to use hooks into database management systems to develop a meta-level protocol, along with meta-level objects, which exploit a set of concepts based on a common transaction processing abstraction (e.g. parsing, optimization, execution) albeit implementations are highly dependent on database management systems. By exposing a common meta-level programming interface, our approach eases the development of a variety of plugins (e.g. replication, query caching, self-optimization). We name it the GORDA DBMS reflective Architecture and Programming Interfaces (GAPI) [15].

## 3 Related Technologies and Problems

Developing distributed applications, such as replication protocols, is a difficult task that involves interprocess communications, support to heterogeneous platforms, and the ability to adapt, for instance, to failures. In order to deal with these challenges, several key technologies have been employed. Examples are computational reflection [22, 20], component based design [5], and design patterns [13]. Different projects provide different perspectives on how to use and exploit these technologies. In particular, computational reflection has been used

for multiple purposes, ranging from dependable architectures to database systems. In this section, we highlight how the GAPI uses these technologies and compares different strategies based on computational reflection.

## 3.1 Design patterns and Component Based Design

The design of a meta-programming interface is based on design patterns that have been useful in the broader context of object oriented distributed applications. Namely, façade [13], inversion-of-control, and container managed concurrency patterns are used[2]:

- The façade pattern allows inspection of diverse data structures through a common interface. A very well known example is the ResultSet of the JDBC[3] specification, which allows results to be stored in a DBMS native format. The alternative is the potentially expensive conversion to a common format such as XML. The proposed architecture suggests using this for most of the data that is conveyed between processing stages (e.g., parser, optimizer).

- The inversion-of-control pattern eases the deployment of software components. In detail, meta-objects such as transactions, are exposed to an object container, which is configured with reflection components. The container is then responsible for injecting the required meta-objects into each reflection component during initialization.

- The container managed concurrency pattern allows the container implementation to schedule event notifications according to its own performance and correctness criteria. For instance, by ensuring that no two transaction commit notifications are issued concurrently, implicitly exposes a commit order. For instance, a notification of available write-sets of two different transactions can be issued concurrently.

## 3.2 Reflective architectures

### 3.2.1 Dependable Systems

The use of computational reflection is not new in the field of dependable applications and the first approach is from the early nineties. The MAUD, GARF, FRIENDS, IRL, and FTS systems briefly introduced below are representative of this approach:

- The MAUD (Meta-level Architecture for Ultra dependability) uses a high-level language [1] based on the actor model which provides a mathematical framework for concurrent systems. Actors are first-class entities that can make decisions, create other actors, receive and send messages.

[2] `http://www.martinfowler.com/articles/injection.html`
[3] `http://java.sun.com/javase/technologies/database/`

- The GARF System [16] is an extension to a Smalltalk Environment based on a set of classes and a runtime environment. It divides computation among data objects, common objects created by the Smalltalk, which handle functional properties of a system and behavioral objects which handle crosscutting concerns. Whenever a data object is created, the GARF runtime environment wraps it with a behavioral object enabling to intercept invocations to the data object.

- The FRIENDS System (Flexible and Reusable Implementation Environment for your Next Dependable System) [11] uses a specialized meta-level protocol based on Open C++, a pre-processing extension to C++, which provides special statements to associate an object with a meta-level object.

- Interoperable Replication Logic (IRL) and Fault-Tolerant Service (FTS) provide fault tolerance by using CORBA request portable interceptors to forward requests to proxies that furnish fault-tolerant services by means of replication [23, 12].

In such projects, a reflective architecture along with object-oriented programming methods frees developers from details on particular dependability protocols and promotes reusability. In that broad sense, our approach aims at the same goals, given that it encapsulates details on databases by means of the GAPI, thus easing the development of plugins and promoting their reusability among different database vendors. In contrast to previous approaches, GAPI does not rely on extensions to a programming language nor is bounded to one.

### 3.2.2 Database Management Systems

Most database management systems provide a reflective mechanism where hooks are defined by means of triggers, notifying meta-level code whenever a relation is updated. Albeit native approach might be used to handle some functionalities required by the GAPI, it might generate an unbearable overhead as the life-cycle of the meta-information produced (i.e. write-sets) is restricted to the meta-level execution. Furthermore, this native approach does not provide other important requirements to easy the development of add-ons. For instance, by using it one cannot hold the execution when a transaction commits thus forbidding the development of add-ons such as synchronous replication protocols that require processing in transactions' contexts.

In [24], reflection is used to introduce self-tuning properties into database management systems. Configuration and performance parameters are reflected into tables and triggers are used to orchestrate interaction among components. Periodically, a monitor tool, that uses a DB2 UDB snapshot API, collects performance information and store it into tables. Right after, a trigger notifies diagnosis functions that decide whether to change configuration information into tables or not. Once a configuration is modified a trigger notifies the DB2 which applies the changes. In [30], the same idea is presented but without relying on

a specific DBMS vendor, thus assuming that most DBMSs provide the means to inspect performance information and change configuration parameters.

The GAPI reflective architecture can also be used to develop self-tuning solutions by inspecting information on query plans and tracking the number of concurrent transactions and throughput. The proposed interface was designed to be generic and it provides only some capabilities to collect this information, but it can be used and easily extended to achieve those purposes.

In [4], it is described a reflective system on a TP Monitor (Transaction Processing Monitor) in order to support the development of extended transaction models (e.g long running transactions). The functional aspects such as transaction execution, lock management and conflict detection are reflected through adapters which provide meta-level objects and a meta-level programming interface. This interface is different from what the GAPI provides as Roger Barga et al. are concerned on mechanisms that enable, for instance, to join and split transactions thus requiring more information on locks and conflicts and different meta-information on transactions. Not only information on which tuples were read or updated need to be reflected but also information on types of locks and pending lock requests. This is necessary to transfer locks acquired and to be acquired on behalf of a transaction to another transaction. The conflict detection adapter is used to relax consistency when executing extended transaction thus providing access to shared tuples which would be locked by the normal conflict detection mechanisms.

In [32], it is proposed a reflection mechanism for database replication. In contrast to our approach, it assumes that reflection is achieved by wrapping the DBMS server and intercepting requests as they are issued by clients. By choosing beforehand such implementation approach, one can only reflect computation at the first stage (statements), i.e. with a very large granularity. Exposing further details requires rewriting large portions of DBMS functionality at the wrapper level. As an example, Sequoia [9] does additional parsing and scheduling stages at the middleware level. Theoretically, this proposal can be more generic and usable on closed systems. In practice, this is not always true, since DBMSs usually do not support exactly the same language and middleware solutions must be customized for a certain system. Despite that, we will see later in this paper that this approach can introduce a significant overhead to latency of transactions by requiring extra communication steps and/or extra processing of requests.

# 4 Reflective Architecture and Interfaces

In this section we overview and motivate the GORDA DBMS reflective Architecture and Programming Interface, simply denoted GAPI. The full details about the architecture and interfaces are described in [15]. In the next sections we will illustrate GAPI with several use cases and evaluate its overhead in real implementations.

## 4.1 Target Reflection Domain

The GAPI has been designed having in mind the support for database replication applications. Although the use of the GAPI is not limited to this class of applications, replication protocols are quite demanding in terms of functionality that needs to be exposed, and their requirements strongly influenced the design and implementation of our reflective interface.

Previous reflective interfaces for database management systems were mainly targeted at application programmers using the relational model. Their domain is therefore the relational model itself. Using this model, one can intercept operations that modify relations by inserting, updating, or deleting tuples, observe the tuples being changed and then enforce referential integrity by vetoing the operation (all at the meta-level) or by issuing additional relational operations (base-level).

In contrast, there are protocols concerned with details that are not visible in the relational model, such as modifying query text to remove non-deterministic statements, as those involving `NOW()` and `RANDOM()`. For instance, one may be interested in intercepting a statement as it is submitted, whose text can be inspected, modified (meta-level) and then re-executed, locally or remotely, within some transactional context (base-level).

Therefore, a more expressive target domain is required. We select an object-oriented concurrent programming environment. Specifically, we use the Java platform (but any similar language would also fit our purposes). The fact that a series of activities (e.g. parsing) is taking place on behalf of a transaction is reflected as a transaction object, which can be used to inspect the transaction (e.g. wait for it to commit) or to act on it (e.g. force a rollback).

Meta-level code can register to be notified when specific events occur. For instance, when a transaction commits, a notification is issued and contains a reference to the corresponding transaction object (meta-object). Actually, handling notifications is the way that meta-level code dynamically acquires references to meta-objects describing the on-going computation.

## 4.2 Processing Stages

The usefulness of the meta-level interface depends on what is exposed as meta-objects. If a very fine granularity is chosen, the interface cannot be easily mapped to different DBMSs and the resulting performance overhead is likely to be high. On the other hand, if a very large granularity is chosen, the interface may expose too little to be useful.

In contrast, previous approaches assume that reflection is achieved by wrapping the DBMS server and intercepting requests as they are issued by clients [32]. By choosing beforehand such implementation approach, one can only reflect computation at the first stage, i.e. with a very large granularity. Exposing further details requires rewriting large portions of DBMS functionality at the wrapper level. As an example, Sequoia [9] does additional parsing and scheduling stages at the middleware level.

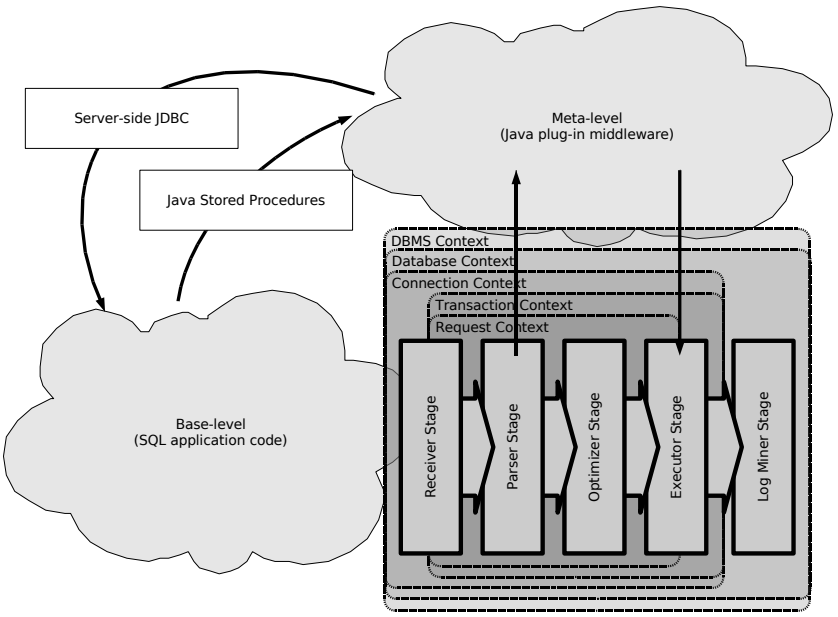



Figure 1: Major meta-level interfaces: processing stages and contexts.

Therefore, we abstract transaction processing as a pipeline as described in [14] and illustrated in Figure 1. The meta-objects exposed by the GAPI, at each stage of the pipeline processing, are briefly listed below (for further details, please see [15]). The plugin is notified of these meta-objects in the order they are listed.

**Receiver Stage** receives new statements from the clients. Notifies the reception of a new statement that can be inspected and/or modified at this moment;

**Parser Stage** parses single statements received thus producing a parse tree;

**Optimizer Stage** receives the parse tree and transforms it, according to various optimization criteria, heuristics and statistics into an execution plan;

**Executor Stage** executes the plan and produces object-sets;

**Log Miner Stage** deals with mapping from logical objects to physical storage.

In general, one wants to issue notifications at the meta-level whenever computation proceeds from one stage to the next. For instance, when write-sets are issued at the execution stage, a notification is issued such that they can be observed. The interface thus exposes meta-objects for each stage and for data that moves between them.

## 4.3 Processing Contexts

The meta-interface exposed by the processing pipeline is complemented by nested context meta-objects, also shown in Figure 1. These show on behalf of whom some operation is being performed. In detail, the contexts are the following:

**DBMS Context** represents the database management system, exposes metadata and allow notification of life-cycle events;

**Database Context** interface represents a database, also exposes metadata and allow notification of life-cycle events;

**Connection Context** reflects existing client connections to databases. They can be used to retrieve connection specific information, such as user authentication or the character set encoding used;

**Transaction Context** is used to notify events related to a transaction such as its startup, commit or rollback. Synchronous event handlers available here are the key to the synchronous replication protocols;

**Request Context** used to ease the manipulation of the requests within a connection to a database and the corresponding transactions.

Events fired by processing stages refer to the directly enclosing context. Each context has then a reference to the next enclosing context and can enumerate all enclosed contexts. This allows, for instance, to determine all connections to a database or which is the current active transaction in a specific connection. Notice that some contexts are not valid at the lowest abstraction levels. Namely, it is not possible to determine on behalf of which transaction a specific disk block is being flushed by the log miner stage.

Furthermore, a plugin can attach an arbitrary object to each context. This allows context information to be extended as required by each plugin. As an example, when handling an event fired by the first stage of the pipeline, signaling the arrival of a statement in textual format, the plugin gets a reference to the enclosing transaction context. It can then attach additional information to that context. Later, when handling an event signaling the readiness of parts of the write-set, the plugin follows the reference to the same transaction context to retrieve the information previously placed there.

## 4.4 Event handling

The meta-level code registers event handlers to intercept the flow of data structures within the execution pipeline. An event handler can be set in two different modes: blocking and non-blocking. This is chosen at run time when setting the handler. When a handler is set in blocking mode, the database server suspends 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. When a handler is set in non-blocking mode, the database server does not wait for the order to continue or cancel its execution.

These handlers can be issued concurrently, even if they were registered in blocking mode, unless if 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. The notification to continue or cancel execution can be issued by a different order that were received. This implies that the commit order is determined by the order by which meta-level code orders the execution to continue. 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 not specified.

### 4.5 Base-level and Meta-level Calls

In this architecture, a base-level call is the client call that makes a request using SQL statements. The meta-level calls are the reflection calls exposed by the database engine, that expose the pipeline stages of the execution control. Meta-level programing allows to have a clean separation between the base- and meta-level architecture concerns, but it has the advantage that the base- and meta-level calls can be mixed, as there is no inherent difference between base- and meta-objects. This happens also in the proposed interface, albeit with some limitations.

In detail, a direct call to meta-level code can be forced by the plugin 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 *Execute* stage. The target procedure can then retrieve a pointer to the enclosing *Request* context and thus to all relevant meta-interfaces. The reason for allowing this only from the *Execute* stage is simplicity, as this is inherently supported by any DBMS, and does not seem to impact generality. A second reason is that this is where the pipeline can be reentered, should the meta-level procedure need to callback into the base-level.

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 context. This can be achieved using the JDBC client 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. The reason for allowing base-level to use the JDBC interface is again simplicity, as this avoids the need to have interfaces that build contexts and inject external data into internal structures. This may however have an impact on performance, and is thus the subject of future work as discussed in Section 7.

The calls between meta-level and base-level are exemplified in the Section 5.1. This example shows how to build a caching mechanism using the proposed interface. As it can be seen in Listing 1, the `cacheLookup` procedure is stored in the database when it starts (lines 39 to 58) and is called later when

a new incoming statement is notified (lines 26 to 38). This exemplifies the direct call to the meta-level code. The stored procedure (lines 12 to 25) uses a database connection to populate the cache in the case that the request is not present, which exemplifies a call from the base-level to the meta-level.

A second issue when considering base-level calls is whether these also get reflected. The proposed option is to disable reflection on a case-by-case basis by invoking an operation on context meta-objects. Therefore, meta-level code can disable reflection for a given request, a transaction, a specific connection or even an entire database. Actually this can be used on any context meta-object and thus for performance optimization. For one, consider a replication protocol, that is notified that a connection will only issue read-only operations, and thus ceases monitoring them.

A third issue is how base-level calls issued by meta-level code interact with regular transaction processing regarding concurrency control. Namely, how are conflicts that require rollback resolved, namely, in multi-version concurrency control where the first commiter wins or, more generally, when resolving deadlocks. The proposed interface solves this by ensuring that transactions issued by the meta-level do not abort in face of conflicts with regular base-level transactions. Given that a plugin code running at the meta-level has a precise control on which base-level transactions are scheduled, and thus can prevent conflicts among those, has been sufficient to solve all considered use cases. The simplicity of the solution means that implementation within the DBMS resulted in a small set of localized changes.

## 4.6 Exception Handling

Regarding base-level exceptions, DBMSs handle most of them by aborting the affected transaction and generating an error to the application. The proposed architecture does not change this behavior. Furthermore, the meta-level is notified by an event issued by the transaction context object; This allows meta-level to cleanup after an exception has occurred.

Most exceptions within a transaction context that are unhandled at the meta-level can be resolved by aborting the transaction. However, some event handlers should not raise exceptions to avoid incoherent information on databases or recursive exceptions, namely: while starting up or shutting down a database, while rolling back a transaction, or after committing one. In such points, any unhandled exception will leave the database in a panic mode requiring manual intervention to repair the system. Furthermore, in such points, interactions between the meta-level and base-level are forbidden and any attempt of doing so, puts the database in a panic mode.

Exceptions from meta-level to base-level calls need additional management. 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. Such cases are handled as meta-level errors, to avoid disseminating errors inside the database while executing

the base-level code.

# 5 Case Studies

Providing the ability to easily develop and test new features in a database is highly important both for academic and business purposes. Challenging implementation problems such as online analytical processing, data mining, caching mechanisms, replication, self-management, stream processing, approximate answers and probabilistic reasoning, represent some of the interests that have dominated the attention of these communities. In this section, we demonstrate the usefulness of the GORDA reflective architecture and interface (GAPI) showing case studies that are driven by the need to change and improve existing database architectures. In particular, we focus on providing caching mechanisms, replication and self-management infra-structures.

## 5.1 Database Caching

Database caching is an important technique to improve system performance and scalability, increasing throughput and reducing latency, by offloading database management systems. It is particular suited for applications that have a high number of read operations when compared to the number of writes. Multi-tier environments, where a middle tier application server access a database backend, might also take advantage of this technique, by caching intermediate results thus reducing communication steps and database usage [21].

In both cases, the GAPI might be used to create a plugin to intercept statements and its results. Using the intercepted information, cache entries may be created and invalidated. For instance, one might capture statements and information provided by the parser and optimizer. In particular, the parser should be used to easily identify a statement as a query or an update and the optimizer to provide information on the cost to process the statement. When identified as query, the result set might be used to populate the cache. This would be done only when a threshold based on the cost provided by the optimizer was reached. When identified as an update, the cache should be invalidated, automatically refreshed or tagged as invalid. Tags might be used to identify, for instance, the number of changed items since the last refresh, thus allowing a query to specify the number of stale information that it tolerates. In other words, such tags might be used to postpone invalidation by relaxing the consistency criterion provided by the database and thus further improving performance and scalability [17].

In the listing follows, to avoid many details in the description, we illustrate the ability to capture statements and result sets.

Listing 1: Query Cache example.

```
1 public class QueryCache implements StatementExecutionListener,
    DatabaseStartupListener, ObjectSetWriteListener {
  private static RequestProcessor reqProc = null;
```

```
    public QueryCache(DatabaseProcessor dbProc, RequestProcessor reqProc,
6   ReceiverStage stmtProc, ExecutorStage objProc) {
      this.reqProc = reqProc;
        dbProc.setDatabaseStartupListener(this, true);
        stmtProc.setStatementExecutionListener(this, true);
        objProc.setObjectSetWriteListener(this, true);
11    }
      public static ResultSet cacheLookup(String reqId, String query)
       throws SQLException {
        Connection c =
          DriverManager.getConnection("jdbc:default:connection");
16      Transaction tx = reqProc.getRequest(reqId).getTransaction();
        Object cached = cache.get(query);
        if (cached == null) {
           java.sql.Statement s = c.createStatement();
           ResultSet rs = s.executeQuery(query);
21         cache.put(tx, query, rs);
        }
        c.close();
        return (new ResultSet[]{cached};);
      }
26    public void handleStatementExecution(Statement st)
       throws SQLException {
        switch (st.getState()) {
           case Statement.PIPELINE_PROCESSING:
              if(st.getStatement().startsWith("select")) {
31             st.setStatement("CALL cacheLookup('"
                   + st.getRequest().getId() + "', '"
                   + st.getStatement() + "')");
              }
              st.continueExecution();
36            break;
        }
      }
      public void handleDatabaseStartup(Database db)
        throws SQLException {
41      switch (db.getContextState()) {
           case Database.DATABASE_STARTING:
              db.continueExecution();
            break;
           case Database.DATABASE_UP:
46            DataSource ds = db.getDataSource();
              Connection con = ds.getConnection();
              java.sql.Statement st = con.createStatement();
              st.execute("CREATE PROCEDURE "
               + "cacheLookup(reqid VARCHAR(10),"
51             + "query VARCHAR(100)) "
               + "EXTERNAL NAME 'gorda.demo.QueryCache.cacheLookup'");
              st.close();
              con.close();
              db.continueExecution();
56            break;
        }
      }
      public void handleObjectSetWrite(ObjectSet objSet)
       throws SQLException {
61      switch (objSet.getState()) {
          case ObjectSet.PIPELINE_PROCESSING:
             cache.invalidate(objSet);
        break;
        }
66      objSet.continueExecution();
    }
  }
  abstract class Cache {
    void put(Transaction tx, String query, ResultSet rs);
71  String get(String statement);
```

```
  void invalidate(ObjectSet ws);
}
```

**Query Cache Plugin Execution**

The Query Caching plugin requires the Database context to capture the moment when the database is started and the Transaction context to define its boundaries. It also needs statements provided by the Receiver Stage and write sets extracted by the Executor Stage. The execution of the Query Cache plugin consists of the following steps:

*Step 1:* The database is started and a procedure is registered to make cache lookups and execute statements that are not in the cache (lines 45 to 55);

*Step 2:* When a statement is received, the plugin is notified and executes the previously stored procedure that verifies if the cache contains the required statement. If the statement is a read operation and the cache contains the statement, the results are returned to the client. Otherwise, the statement is executed, the results are added to the cache and then returned to the client (lines 26 to 36 and the procedure in lines 12 to 25);

*Step 3:* When a request is executed and the object sets contain a write set, the plugin is notified and uses the write set to invalidate possible entries in the cache that contain obsoleted information (lines 59 to 67). It is worth noticing that this should be done only when (and if) a transaction commits. This feature and the code of cache invalidation are omitted for simplification.

## 5.2 Database Replication

As a use case for database replication, we will use the primary-backup approach, also called passive replication [25]. In this approach, update transactions are executed at a single master site under the control of local concurrency control mechanisms. Updates are then captured and propagated to other sites. The main advantage of this approach is that it can easily cope with non-deterministic servers. A major drawback is that all updates are centralized at the primary and little scalability is gained, even if read-only transactions may execute at the backups. It can only be extended to multi-master by partitioning data or defining reconciliation rules for conflicting updates. Asynchronous primary-backup is the standard replication in most DBMSs and third-party offers. An example is the Slony-I package for PostgreSQL [29].

Implementations of the primary-backup approach differ whether propagation occurs synchronously within the boundaries of the transaction or, most likely, is deferred and done asynchronously. The later provides optimum performance when synchronous update is not required, as multiple updates can be batched and sent in the background. It also tolerates extended periods of disconnected operation. The Primary-Backup protocol has a primary replica

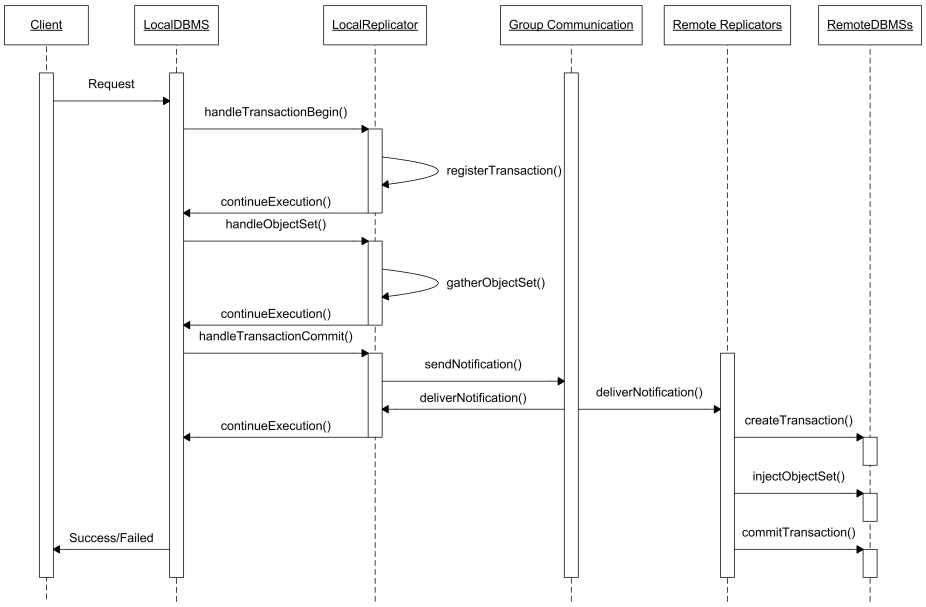



Figure 2: Replication Plugin.

where all transactions that update the database are executed. Updates are either disseminated in transaction's boundaries (i.e., synchronous replication) or periodically propagated to other replicas in background (i.e., asynchronous replication).

### Replication Plugin Execution

Synchronous primary-backup replication requires the component that reflects the Transaction context to capture the moment where the transaction starts executing, commits, or rollbacks at the primary. It will also need the object set provided by the Execution stage to extract the write set of a transaction from the primary and insert it at the backup replicas. The execution of a primary-backup replicator is depicted in Figure 2. We start by describing the synchronous variant. It consists of the following steps:

*Step 1:* Clients send their requests to the primary replica.

*Step 2:* When a transaction begins, the replicator at the primary is notified, registers information about this event, and allows the primary replica to proceed.

*Step 3:* Right after processing a SQL command the database notifies the replicator through the Execution stage component sending an *ObjectSet*. Roughly, the ObjectSet provides an interface to iterate on a statement's result set (e.g.,write set). Specifically, in this case, it is used to retrieve statement's

updates which are immediately stored in a in-memory structure with all other updates from the same transaction context.

*Step 4:* When a transaction is ready to commit, the transaction context component notifies the replicator of the primary. The replicator atomically broadcasts the gathered updates to all backup replicas (this broadcast should be *uniform* [8]).

*Step 5:* The write set is received at all replicas. On the primary, the replicator allows the transaction to commit. On the backups, the replicator injects the changes in the DBMS.

*Final Step:* After the transaction execution, the primary replica replies to the client.

An asynchronous variant of the algorithm can be achieved by postponing Step 4 (and, consequently, Step 5) for a tunable amount of time.

## 5.3 Self-Management

Database management systems are becoming more sophisticated and it has become increasingly more difficult to tune them for particular environments. The features and tuning variables that DBMSs provide (and the interactions among these features) have made the administration task a serious challenge. At the same time, modern computing environments in which DBMSs are deployed pose new challenges such as data fluctuations, unexpected delays and frequent outages. Tuning the underlying system parameters became significantly harder in these complex and unpredictable environments. To address this challenge, several research efforts emerged from the area of autonomic computing [33, 26]. DBMSs management tools can be manipulated by autonomic systems to monitor the DBMS and change the system parameters according to defined policies. For fine grained monitoring, the autonomic system needs more than what is provided by the DBMS standard API. Weikum et. al. in [33] advocates that the feedback control loop provided by a reflection API is an appropriate framework for self tuning and self management algorithms.

There are two types of resources that can influence the behavior of the system: *physical resources* (e.g. CPU and memory) and *logical resources* (the ones provided by the DBMS). By tuning the logical resources, the system can make a better use of the physical resources. Given that the system usage changes with time, the system needs to be continuously monitored and tuned. One way to implement self-tuning is to use the GAPI to collect information about the execution of databases and act according to the system needs and policies. For example, a plugin can be built to calculate the latency of transactions or to verify the locks grabbed by a transaction.

The following example manages the maximum number of concurrent transactions that are allowed to run on the system, at a certain moment and calculates it based on information collected as the system executes. It monitors the last

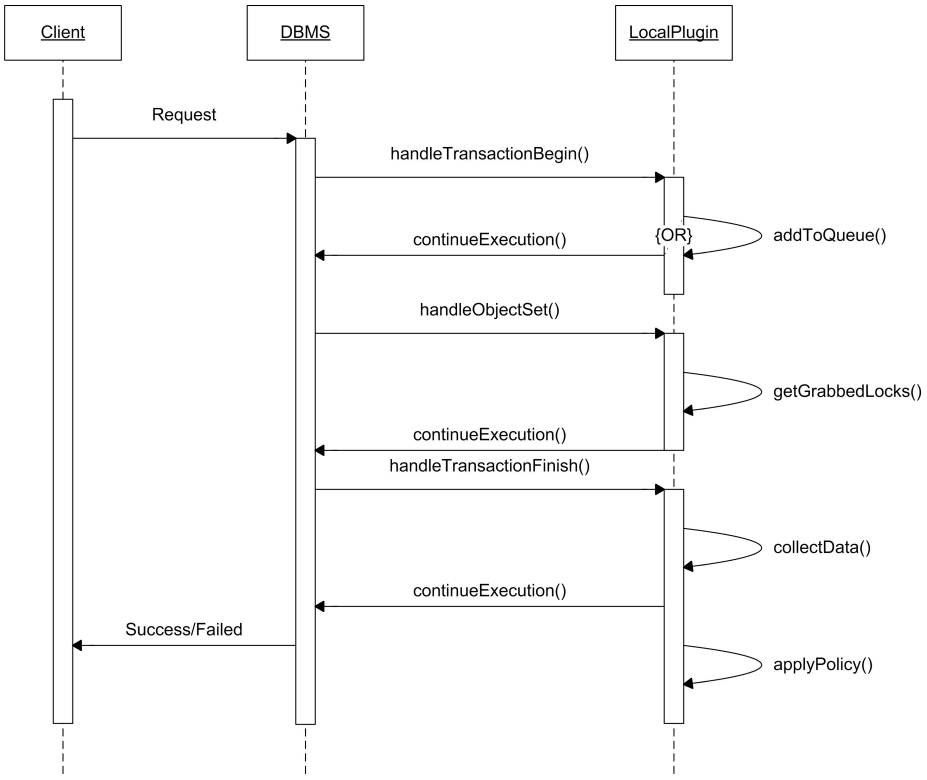



Figure 3: Self Management Plugin.

transaction latency and inter arrival time, the average transaction latency and inter arrival time and the locks acquired by the transaction.

### Self Management Plugin Execution

This plugin requires the component that reflects the Transaction context to capture the moment where the transaction starts and commit or rollback. It will also need the object set provided by the Execution Stage to extract the acquired locks (e.g. write sets without considering lock scaling nor lock hints). The execution of the Self Tuning plugin, depicted in Figure 3, consists of the following steps:

*Step 1:* When a transaction begins, the plugin is notified and logs the information about current time for that transaction. The number of running transactions is also updated. If the maximum threshold is reached, the transaction is queued for later processing, otherwise it runs normally.

*Step 2:* When a request is executed, the acquired locks for that transaction are also logged.

*Step 3:* When a transaction ends, either by committing or aborting, the information about acquired locks, the current transaction latency and inter arrival time, and the average transaction latency and inter arrival time, is updated.

*Step 4:* Based on these new values the system policy is tested and applied, updating the maximum number of transactions that are allowed to execute in the system at the same time.

*Final Step:* If there are pending transactions, try to execute them provided that concurrent transactions are unlikely to conflict.

As shown, the GAPI can be used as a monitoring API, but also as an API to act and accommodate the behavior of the DBMS to the system needs. This example shows how transactions can be delayed to avoid concurrency in the system, thus avoiding conflicts in the lock manager based on previous executions.

# 6 Evaluation

The GORDA API has been implemented on three different systems, namely, Apache Derby, PostgreSQL, and Sequoia. These systems illustrate the effort required to implement the GAPI using different apporaches. In this section, we provide a brief description of each of these implementations, including information about the number of lines of code required to implement GAPI on each architecture.

We also evaluate the performance of the Apache Derby GAPI implementation and compare different approaches to database reflection, namely the in-core implementation of the GAPI interface and the database wrapper.

## 6.1 Implementation effort

### Apache Derby 10.2

Apache Derby 10.2 [3] is a fully featured database management system with a small footprint that uses locking to provide serializability. It can either be embedded in applications or run as a standalone server. It was developed by the Apache Software Foundation and distributed under an open source license; It is also distributed as IBM Cloudscape and in the Sun JDK 1.6 as JavaDB.

The GAPI prototype implementation takes advantage of Derby being natively implemented in Java to load meta-level components within the same JVM and thus closely coupled with the base-level components. Furthermore, Derby uses a different thread to service each client connection, thus making it possible that notifications to the meta-level are done by the same thread and thus reduce to a method invocation, which has negligible overhead. This is therefore the preferred implementation scenario.

**Implementation Effort** The total size of the Apache Derby engine is 514941 lines of code. In order to implement the GAPI interface, 29 files were

changed by inserting 1250 lines and deleting 25 lines; in total, 9464 lines of code were added in new files.

### PostgreSQL 8.1

PostgreSQL 8.1 [28] is a fully featured database management system distributed under an open source license. Written in C, it has been ported to multiple operating systems, and 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. Since version 7.0, it provides a multi-version concurrency control mechanism supporting snapshot isolation.

A challenge in implementing the proposed architecture in Postgres is the mismatch between its concurrency model and the multi-threaded meta-level runtime. PostgreSQL 8.1, as all previous versions, uses multiple single-threaded operating system processes for concurrency. This is masked by using the existing PL/J binding to Java, which uses a single standalone Java virtual machine and inter-process communication. This imposes an inter-process remote procedure call overhead on all communication between base and meta-level.

Therefore, the prototype implementation of the GORDA interface in PostgreSQL 8.1 uses a hybrid approach. Instead of directly patching the reflector interface on the server, key functionality is added to existing client interfaces and as loadable modules. The proposed meta-level interface is then built on these modules. The two layer approach avoids introducing a large number of additional dependencies in the PostgreSQL code, most notably on the Java virtual machine. As an example, transaction events are obtained by implementing triggers on transaction begin and end. A loadable module is then provided to route such events to meta-objects in the external PL/J server.

**Implementation Effort** The size of PostgreSQL is 667586 lines of code; the PL/J package adds 7574 lines of C code and 16331 of Java code. In order to implement the GAPI interface on Postgres 21 files changed by inserting 569 lines and deleting 152 lines, 1346 lines of C code were added in new files, and 11512 lines of Java code added in new files.

### Sequoia 3.0

Sequoia [9] is a middleware package for database clustering built as a server wrapper. It is primarily targeted at obtaining replication or partitioning by configuring the controller with multiple backends, as well as improving availability by using several interconnected controllers.

Nevertheless, when configured with a single controller and a single backend, Sequoia provides a state-of-the-art JDBC interceptor. It works by creating a virtual database at the middleware level, which re-implements part of the abstract transaction processing pipeline and delegates the rest to the backend database.

The current prototype exposes all context objects and the parsing and execution objects, as well as calling from meta-level to base-level with a separate

connection. It does not allow calling from base-level to meta-level, as execution runs in a separate process. It can however be implemented by directly intercepting such statements at the parsing stage. It does also not avoid that base-level operations interfere with meta-level operations, and this cannot be implemented as described in the previous sections as one does not modify the backend DBMS. It is however possible to the clustering scheduler already present in Sequoia to avoid concurrently scheduling base-level and meta-level operations to the backend, thus precluding conflicts.

**Implementation Effort** The size of the generic portion of Sequoia is 137238 lines, which includes the controller and the JDBC driver; additional 29373 lines implement plugable replication and partitioning strategies, that are not used by GAPI. In order to implement the GAPI interface on Sequoia, 7 files were changed by inserting 180 lines and deleting 23 lines, and 8625 lines of code were added in new files.

#### Notes on the GAPI Implementation Effort

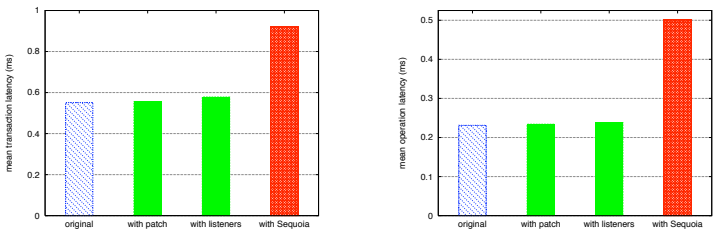
The effort required to implement a subset of the GAPI interface can roughly be estimated by the amount of lines changed in the original source tree as well as the amount of new code added. The numbers presented in the previous sections show that it is possible to implement the GAPI interface in three different architectures, with consistently low intrusion in the original source code. This translates in low effort both when implementing it but also when maintaining the code when the DBMS server evolves.

Note also that a significant part of the additional code is shared, namely in the definition of the interfaces (6144 lines). There is also a firm belief most of the rest of the code could also be shared, as it performs the same container and notification support functionality. This has not happened as each implementation was developed independently and concurrently.

Finally, it is interesting to note that the amount of code involved in developing a state-of-the-art server-wrapper is in the same order of magnitude as a fully-featured database (i.e. hundreds of Klines of code). In comparison, implementing the GAPI involves 100 times less effort as measured in lines of code.

### 6.2 Performance

In this section we evaluate the performance of one prototype implementation of the proposed interface in Apache Derby, and compare different approaches to database reflection, namely the in-core implementation of the GAPI interface and the database wrapper. The purpose of the evaluation is to assess the overhead introduced by an in-core implementation and compare this overhead with other solutions that are based on a DBMS wrapper. 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.




(a) Transactions. (b) Operations.

Figure 4: Performance results.

We use the workload generated by the Poleposition benchmark. The Poleposition [10] benchmark is a framework to build benchmark tests. The tests create a small database in the DBMS. The size of transactions can change with the number of operations that are defined for each transaction. The results are measured in the client and it measures the latency of transactions, in milliseconds. In these tests, we measured the latency of update operations, with 1 and 100 operations on each transaction.

The following scenarios were tested: (*i*) *Unmodified DBMS* is the original DBMS, without any modification, serving as the baseline; (*ii*) *DBMS with patch* 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; (*iii*) *DBMS with listeners* is the modified DBMS with listeners registered for transactional events, statements and object sets. This means that each transaction generates at several events for each transaction; and (*iv*) *DBSM with Sequoia* is the unmodified DBMS with the Sequoia database wrapper, but without doing any reflection.

The results are presented in Figure 4. Figure 4(a) shows the mean transaction latency of one transaction with a single operation. As it can be seen, when no meta-level objects are configured the overhead introduced by the patches is negligible. We can observe the same behavior when we add a plugin that listens to events. It is worth noticing that one of the events is the notification of the object set produced by the transactions. An extra processing is done inside the DBMS to collect the object set, but it is also negligible. As we can see in the final test, the impact adding a DBMS wrapper is noticeable, as this causes an extra communication step and extra processing to parse incoming statements. Figure 4(b) depicts the mean operation latency and was measured by making 100 operations per transaction. The expected behavior of this test is to have some of the latency caused by the DBMS wrapper masked by the low number of commits. Note that the overhead caused by the wrapper is very significant.

# 7 Conclusions

The development of new DBMS plugins for different purposes, such as replication and clustering, require more functionality than the one currently provided current DBMSs transactional APIs. Previous solutions to meet these demands, such as patching the database kernel or building complex wrappers, require a large development effort in supporting code, cause performance overhead, and reduce the portability of middleware. In this paper we advocate the use of a reflective architecture and interface to expose the relevant information about transaction processing in a useful way, namely allowing it to be observed and modified by external plugins.

We have shown the usefulness of the approach by illustrating how the interface can by applied to different settings, such a replication, query caching, among others. We have also shown that the approach is viable and cost-effective, by describing its instantiation on three different and representative architectures, namely the Apache Derby, PostgreSQL, and the Sequoia server wrapper. We measured the overhead introduced by the in-core implementation on Apache Derby and compared it with a middleware solution. These prototypes are published as open source, can be downloaded from the GORDA project's home page, examined, and benchmarked. A modular replication framework that builds on the proposed architecture and thus runs on PostgreSQL, Apache Derby, or any DBMS wrapped by Sequoia, is also available there.

The architecture presented here still has some limitations, that should be addressed by future work. It needs to be extended to perform the composition of multiple independente meta-level plugins. For instance, how to configure a DBMS to use a self-management plugin and a replication plugin at the same time. Again, previous work on reflective systems might provide a direction [2]. Another open issue is the adequacy of the proposed architecture to non-classical database architectures. Namely, how to match it with a column-oriented DBMS, such as MonetDB [6], or with an inherently clustered DBMS such as Oracle RAC [31].

# References


[1] Gul Agha, Svend Frlund, Rajendra Panwar, and Daniel Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *Proceedings of the IFIP Conference on Dependable Computing for Critical Applications*, 1992.

[2] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[3] Apache DB Project. Apache Derby version 10.2. http://db.apache.org/derby/, 2006.

[4] Roger Barga and Calton Pu. A Practical and Modular Implementation of Extended Transaction Models. In *VLDB Conference*, 1995.

[5] Don S. Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM TOSEM*, 11:191–214, 2002.

[6] P. A. Boncz and M. L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proceedings Basque International Workshop on Information Technology*, San Sebastian, Spain, July 1995.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Annual Technical Conference*, 2004.

[8] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33 - 4:427 – 469, 2001.

[9] Continuent. Sequoia v2.9. http://sequoia.continuent.org, 2006.

[10] db4o DeveloperCommunity. Poleposition benchmark. http://polepos.sourceforge.net/, 2006.

[11] Jean-Charles Fabre and Tanguy Pérennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach. *ACM TOCS*, 47:78–95, 1998.

[12] Roy Friedman and Erez Hadad. Client-side Enhancements using Portable Interceptors. In *WORDS*, 2001.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Desing Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[14] H. Garcia-Mollina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[15] GORDA Consortium. Database support for replication revision 0.1. http://gorda.di.uminho.pt/download/reports/dbspec.pdf, April 2007.

[16] Rachid Guerraoui, Benoît Garbinato, and Karim Mazouni. Garf: A Tool for Programming Reliable Distributed Applications. *IEEE Parallel Distributed Technologies*, 5:32–39, 1997.

[17] H. Guo, P. Larson, and R. Ramkrishnan. Caching with "God Enough", Concurrency, Consistency, and Completeness. In *VLDB Journal*, 2005.

[18] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *VLDB Conference*, 2000.

[19] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36:41–50, 2003.

[20] Gregor Kiczales. Towards a New Model of Abstraction in Software Engineering. In *IMSA Workshop on Reflection and Meta-level Architectures*, 1992.

[21] P. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *IEEE ICDCS*, 2004.

[22] Pattie Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA*, 1987.

[23] C. Marchetti. CORBA Request Portable Interceptors: A Performance Analysis. In *DOA*, 2001.

[24] Patrick Martin, Wendy Powley, and Darcy Benoit. Using Reflection to Introduce Self-Tunning Technology into DBMSs. In *IEEE IDEAS*, 2004.

[25] S. Mullender. *Distributed Systems*. ACM Press, 1989.

[26] Baoning Niu, Patrick Martin, Wendy Powley, Randy Horman, and Paul Bird. Workload adaptation in autonomic dbmss. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 13, New York, NY, USA, 2006. ACM Press.

[27] R. Jiménez Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *IEEE ICDCS*, 2002.

[28] PostgreSQL Global Development Group. Postgresql version 8.1. http://www.postgresql.org/, 2006.

[29] PostgreSQL Global Development Group. Slony-I version 1.1.5. http://slony.info, 2006.

[30] Wendy Powley and Pat Martin. A Reflective Database-Oriented Framework for Autonomic Managers. In *ICAS*, 2006.

[31] Angelo Pruscino. Oracle rac: Architecture and performance. In *SIGMOD Conference*, page 635, 2003.

[32] J. Salas, R. Jiménez Peris, M. Patiño Martínez, and B. Kemme. Lightweight reflection for middleware-based database replication. In *SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 377–390, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Gerhard Weikum, Axel Mönkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB Conference*, pages 20–31, Hongkong, China, 2002. Morgan Kaufmann.