Cooperative Transaction Processing between Clients and Servers*

S. Jurk[1], U. Leser[2], J. L. Marzo[3]

[1] Brandenburgische Technische Universität Cottbus, Institut für Informatik, P.O.B. 101344, 03013 Cottbus, Germany, sj@informatik.tu-cottbus.de

[2] Humboldt-Universität zu Berlin, Institut für Informatik, Unter den Linden 6, 10099 Berlin, Germany, leser@informatik.hu-berlin.de

[3] Institut d'Informatica i Aplicacions (IIiA), Universitat de Girona, Lluis Santalo Av., 170 71, Girona, Spain, joseluis.marzo@udg.es

Abstract. Business rules are often implemented as stored procedures in a database server. These procedures are triggered by various clients, but the execution load is fully centralized on the server. We improve the overall response time and increase server throughput by balancing this load between the server and the clients. In our novel scheme, parts of the stored procedures are executed on cached data at the client. The critical issue in such a system is the trade-off between synchronization effort among clients and the server and the increase in systems performance gained by load balancing. We present an architecture using an optimistic synchronization protocol together with an algorithm for data verification at the server. The experience we gained through a detailed case study, based on a real-life eCommerce application, shows that in many situations a considerable speed-up is possible.

1 Introduction

In a typical client-server relational database environment, clients trigger processes at the central server. The server executes them on shared data and passes the result back to clients. Moving a portion of the servers execution tasks to clients can enhance the overall performance of such systems. This has become a feasible approach within the last years ([18, 16, 6, 11]), since, due to the rapid development and decreasing costs in computer hardware, low-end clients are replaced by more and more powerful ones. Such *fat clients* are fully able to perform some database operations on their own.

The situation is similar for multi-tier architectures, such as the one used by SAP R/3. Here, clients are considered to be very thin and to only function for information display and user-interaction. In the middle tier, application servers perform tasks such as session handling, rendering of forms and HTML pages, and minor data modifications. However, data-centric tasks are performed exclusively inside a central database. The sharing of load between the middle tier and the

* This research was supported by the DFG, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316) and by German Ministry for Science and Education, grant no. 0312705B

database server is hard programmed, often leaving the (powerful) application server hardware in an idle state.

In both cases, the central server clearly is a central bottleneck of the system, while at the same time valuable resources at the client (middle tier application server or user PC) are wasted. In this paper, we investigate the possibility to improve response time and system throughput by balancing load between the clients and the server. We assume that processes executed at the server are implemented as stored procedures, i.e., as pieces of database code executed under transactional semantics. Normally, the client invokes a procedure at the server and awaits the result before continuing. We suggest that clients, instead of being idle, themselves perform parts of the procedures on local replicas of the data.

Essentially, our method works as follows. We automatically translate the code of a procedure into two versions, one installed at the server and one installed at the clients. These two version differ in that the code is logically split into two parts, say A and B. When the procedure is triggered in a client, both versions start running in parallel. The client actually performs the commands in part A and informs the server about doing so, while the server performs part B, and only *verifies* the client computations of part A. Only if this verification fails due to stale data, the server also executes erroneous computations.

Before we give an example, we want to discuss the difficulties and premises of this scheme. The success of our method depends on a number of characteristics of the data distribution and the procedures. First, it will only pay off, if procedures can be logically split such that load is taken from the server, i.e., when server verification of results computed by the client is cheaper than executing the commands at the server. This will not be the case if only very simple operations are involved. Second, the effort for data synchronization between client and server must be taken into account. Our scheme is only advantageous, if the verification in most cases succeeds, i.e., if the client mostly works on up-to-date data; otherwise, server load would be increased rather than decreased. Therefore, our method will not work well in cases, where many clients constantly change some portion of data, or if the procedures require large amounts of data from the central database.

However, in many cases the situation is not as such, as illustrated by the following example of our real-world case study (Section 4). Its 3-tier architecture (web client, web server, database server) is depicted in Figure 1. Note that in this case the web server and not the web client acts as database client. Our scenario consists of a classical web shop with product groups, products, a shopping cart, paying, etc. and business processes, such as login, register, add-to-cart, buy, clear-shopping-cart, browse catalog, etc. The procedure code (in pseudo code notation) for adding products to the shopping cart is as follows.

```
PROCEDURE AddCart(@sessionID,@productID,@amount)
1 @session := (SELECT * FROM Session WHERE id=@sessionID);
2 IF @session is not valid THEN
3        DELETE FROM Session WHERE id=@sessionID;
4        RETURN ErrorNoSuchSession     END IF;
```

```
5 amount := (SELECT amount FROM Stock WHERE pid=@productID);
6 IF amount<@amount THEN
7        RETURN ErrorOutOfStock     END IF
8 INSERT INTO Cart VALUES (@sessionID,@productID,@amount);
9 RETURN SuccessCode
```

One possibility is to split the code by the lines of code. Consider a split A=5 and B=1-4,6-9. Then client and server are executing A and B in parallel. For this, a client hosts the table `Stock` that in our case at most is updated once a minute by purchased products. By propagating only updates on server data to clients (c.f. Section 2), the amount of client data and synchronization is low.

Once the client has finished step 5, it passes the result `amount` to the server. After the server has finished steps 1-4, it tries to use the clients result (if available) and validates it, before it proceeds with steps 6-9. The validation is necessary, since our optimistic synchronization scheme allows clients to operate on stale data. The validation can be achieved by a simple versioning scheme on tables. Versions are consistently maintained by the server and propagated to clients by the synchronization scheme. Thus, the result `amount` is correct, if the client and server version of `Stock` equals. Hence, comparing versions is much cheaper than computing step 5 at the server.

Without client computations we observed the following average execution time for `AddCart`: 22ms for step 1-4, 13ms for step 5, 2ms for step 6-9 — in total 37ms. Hence, the total execution time improves for all situations where the client computation (step 5), the delivery of `amount` to the server and the validation take less than 22+13ms (time to complete steps 1-5 at the server). In our case study we achieved an execution time of 25ms.

Consistency is fully preserved; if other clients concurrently change table `Stock` at the server and the result `amount` was computed on a previous version, the server will notice a conflict due to different versions and, instead of using the client result, will start executing step 5 by himself. In this case, our scheme puts some overhead on the server, but if conflicts are rare, considerable load is taken from the server. To avoid huge amount of data at clients, we use table fragments within our execution scheme.

The remainder of this article is organized as follows. We explain the architecture of our system, procedures and data distribution in Section 2. Code splitting, partial execution at the client and validation are described in Section 3. Within a case study (Section 4) we show how our approach has been applied successfully to an Internet shopping application. Section 5 gives an overview of related work. Finally, we provide an outlook in Section 6.

In [11], we only considered the sequencial execution of stored procedures and concentrated on breakpoints within the procedure code. Now we extend this solution by parallelism. Further, we have shown the feasibility on a real-world Internet shopping application.

2 Architecture

We describe the minimal infrastructure that is required to exemplify our approach. A discussion of further database concepts, such as triggers, sequences, or cursors, is beyond the scope of this paper. Furthermore, many of those concepts are fully compatible to our methods. Concepts posing conceptual problems shall be highlighted in the text.

This section covers all issues that are *not* directly involved in the cooperative execution of procedures. Those, especially code splitting, partial execution and validation, are discussed in Sections 3.

We consider a scenario where a central database (server) serves a number of distributed clients. The server provides its functionality *only* by means of a set of procedures. We do not consider client-side ad-hoc queries to the server as these can always be encapsulated in a parameterized procedure. Each client C_i is running a simple database system that is used for caching data and executing procedures locally. These database systems do not require advanced features, such as multi-processor capabilities, recovery, or multi-user support. Figure 1 depicts the main components of our architecture. An execution engine (c.f. Sec-

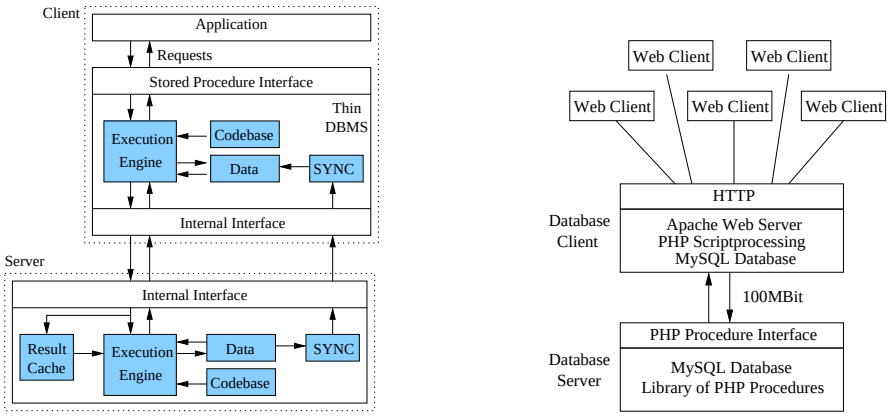

Fig. 1. (Left) General architecture of the Client-Server System. (Right) Architecture as used by the case study in Section 4.

tion 2.2) at the client and the server is responsible for performing procedures. Both engines work on the local database using local versions of the same set of procedures (see Section 2.1). At the server, a cache (c.f. Section 3) is used to store intermediate results computed by clients during parallel procedure execution. A synchronization component (c.f. Section 3.3) is responsible for updating those parts of the database that are replicated at the clients.

2.1 Data and Update Language

The database schema consists of flat tables, denoted R, where all attributes are atomic (1. NF). For simplicity of explanation, we do not consider constraints and advanced concepts, such as triggers, sequences, etc. Also they do not pose any conceptual problem. For keeping surrogate keys, each table contains a readonly `ID` column. We assume a procedure language in the flavor of pgplsql as used in PostgreSQL or PL/SQL as used in Oracle. A procedure has a name and may have input parameters and local variables, denoted as `<var>`. In this article we deal with the following commands `<stmts>`: `INSERT`, `DELETE`, `UPDATE`, `SELECT INTO <var> <sql-query>` and `<var>:=<expr>` as assignments of variables, `IF <cond> THEN <stmts> ELSE <stmts> END IF`, `FOR <var> IN <sql-query> LOOP <stmts> END FOR`, `RAISE EXCEPTION`, `RETURN`. The expressions `<cond>` and `<expr>` are restricted to contain only local variables variables, constant values, build-in functions and operators according to the used types. Note also that there are no subroutines.

The parallel cooperative execution (c.f. Section 3) uses updates of single tuples for synchronizing data and to validate client results. Since `INSERT`, `DELETE`, `UPDATE` statements usually include a query, e.g. `UPDATE R SET X=X+1 WHERE Y>X AND Z<>''`, we apply a pre-compilation step that translates them into low-level updates.

Low-level updates directly access single tuples of tables. For a table R, a tuple $t = (a_1, \ldots, a_i)$ and a tuple identifier tid, low-level database updates are: $insert(R, t)$, $delete(R, tid)$ and $replace(R, tid, t)$. We assume that a unique tuple identifier is maintained by $insert(R, t)$ which generates a tid.

The idea is to replace each `INSERT`, `DELETE`, `UPDATE` statement by: (1) A query that computes which tuples (including their tid, for delete and update) should be inserted, deleted or updated. (2) A loop that for each query result performs a low-level update. The `DELETE FROM` R `WHERE ..` statement, for example, is translated into:

```
FOR row IN SELECT ID FROM R WHERE .. LOOP
    delete(R, row.ID);
END LOOP;
```

Hence a pre-complied procedure interacts with data only in terms of low-level updates and queries, in the following denoted as *IO statements*. Since we later in Section 3 execute such procedures in parallel at client and server, we require that both behave equal, once executed on equal data. To achieve this, we pose one restriction on the types of queries, i.e., we only allow the use of build-in functions $f(x) = y$ that are time-invariant and do not access data in the database. This disallows functions such as those accessing the systems date (because they are time-dependent), functions maintaining sequences (these access and manipulate data), etc. This property is required by our validation algorithm for client results (c.f. Section 3.3).

2.2 Executing Procedures

The execution engine is responsible for processing pre-complied procedures. Intuitively, it is executing the code by performing IO statements (low-level updates and queries) on data. We briefly define the minimal requirements of the engine.

We uniquely identify IO statements within the code of a procedure S. Let $E(S)$ be the set of identifiers for S and let $sid(s)$ be a function which for each IO statement s returns its ID. In the following, letters s denote such IDs. Then, each execution of a procedure S is characterized as a sequence $Seq = (s_1, \#s_1), \ldots, (s_n, \#s_n)$ of executed IO statements $s_i \in E(S)$, where $\#s_i$ denotes number of already performed executions of statement s_i Hence, it represents a counter for each s_i that is maintained by the engine and used to distinguish between different executions of an s_i during a loop.

Given an IO statement s_i, first, the engine is instantiating s_i (resolve local variables, if any) and, second, is executing the instantiated IO statement e by a function $val = eval(s, \#s, e)$ that returns the result value val (empty for low-level updates, a table for queries) to the engine. By using val, the engine is further processing S until the next IO statement.

In Section 3 we logically split a procedure S as follows. Part A (executed at the client): $E_C \subseteq E(S)$ and part B (executed at the server): $E(S)$ except query statements in E_C. Client and server execute the sane version of S in parallel. The client is only allowed to execute IO statements in E_C and the server tries to use the results of these computations. For this, we will extend the clients engine to cope with this partial execution (c.f. Section 3.2) and will model our execution scheme by providing *only* different $eval(.)$ functions for client and server. Hence, parameters E_C determine the amount of client computations.

2.3 Data Distribution and Version Management

As stated above, each client has a local database that contains the data that is used during the execution of statements E_C. Since clients usually do not access all data of a table, data is replicated at the level of table fragments and only necessary fragments are placed at clients. Each table R at the server is partitioned into a set of fragments F_k^R, short F_k, that can be assigned to various clients. In Section 3.2 we show that only fragments are placed at clients that are required for executing statements E_C.

However, strategies for optimal fragment design are beyond the scope of this paper (see for instance [7]). Clearly, other fragmentation schemes could be used instead of our method (e.g. semantic or predicate based methods [10, 6, 9]). However, a detailed analysis of the impact of different fragmentation schemes is beyond the scope of this paper and subject of future work

Consider the example from Section 1. There, the table `Stock` could be partitioned by product groups, such that each fragment contains tuples of the same product group. Fragments, i.e. product groups, are then assigned to those clients that are mostly interested in these groups.

Additionally, the server maintains for each F_k a version number v_k. Version numbers are only updated by procedures S at the server (and not by the client). That is, once the execution of S completes, the version number is increased for all fragments F_k that have been modified by S. Each procedure runs as a single transaction. After the versions have been updated the transaction commits. Hence, versions are consistently updated under concurrent access at the server. Note that a procedure might perform multiple updates during its execution.

In Section 3.3 we use version numbers to verify client computations. For this, we require that client and server fragments F_k with equal versions imply equal fragment data. By the restriction on build-in function to prevent side-effects (c.f. Section 2.1) we conclude that a procedure call, executed at client and server on equal data, must produce the same low-level updates and query results. Hence, equal versions indicate that client and server executions are equal.

Since procedures might apply multiple updates before a version is updated, we must pay special attention on fragment placements at clients. For this, the content of client fragments F_k of version v_k must equal to the content of F_k at committing time of the procedure that updated the version to v_k. Thus, clients operate only on data (snapshots) that correspond to data at the completion time of procedures. In the following we call such fragments *initial*, since they reflect the initial state of a fragment version, where no other updates have been applied yet. Note that also procedures at the server operate on *initial* versions, since the transactional execution requires that other transactions do not see data that has not been committed yet.

Hence, a procedure at client and server is always executed on *initial* versions. An appropriate synchronization scheme achieving this property is explained in Section 3.4 after the client and server behavior have been defined.

3 Parallel Transaction Processing

This Section explains all issues that belong to the parallel cooperative execution of procedures. We address the splitting, the servers result cache, the execution at client and server, and the validation technique for checking client results.

Recall the enumeration $E(S)$ and the $eval(.)$ function as defined in Section 2.2. The code of a procedure is logically split into two parts (A) $E_C \subseteq E(S)$ and (B) $E(S)$ except query statements in E_C. In the following we assume that for each client-procedure pair (C_i, S_j) there exits a split $E_C(C_i, S_j)$, short E_C, within the code base of client and server. The finding of appropriate splits is discussed in Section 4. We shortly summarize the execution scheme and provide details in subsequent sections.

Once a procedure S has been initiated, the client computes a unique execution identifier *eid* (e.g. a number increased by one for each procedure call), sends it to the server and S is executed at client *and* server in parallel. Thereby the $eval_C(s, \#s, e)$ function of the clients execution engine handles only statements $s \in E_C$ and puts the result of each executed query statement into the servers cache and performs low-level updates on local data. This updates are necessary,

since they might affect the evaluation of queries at the client within the same procedure. Cached query results are later on re-used by the server.

The servers cache consists of tuples

$$(eid, s, \#s, e, val, ver)$$

with $val = eval(s, \#s, e)$ and ver as the set of all fragments versions accessed by the execution of e.

The $eval_S(s, \#s, e)$ function of the servers engine considers all $s \in E(S)$, but uses cached query results (if valid) for $s \in E_C$ instead of executing them. Thus the execution of queries in E_C has been outsourced to clients. To log updates on primary data, the server puts executed low-level updates into the cache ($val =$ for low-level updates). This updates are used by the synchronization scheme to update client data. Based on versions, the server validates client computations and re-executes a query on local data in case it is not valid.

3.1 The Partial Execution Model

Executing IO statements E_C at the client, requires the clients execution engine to cope with a partial execution of a procedure. The idea is that the clients $eval_C()$ function returns an $undef$ value for each IO statement $s \notin E_C$. Hence, the clients engine must cope with $undef$ values. For this, we also use an additional `ABORT` operation that stops the current execution and (different to `RAISE EXCEPTION`) does not undo updates. Arising data inconsistencies at the client are handled by the synchronization (c.f. Section 3.4). For our procedure language the engine behavior is sufficiently described by the following rules: (1) Since the $eval_C(.)$ function might return $undef$ values, an assigned variable is of value $undef$, if the assigned expression contains an $undef$ value. Hence, an $undef$ value can be further used within the execution of a procedure and might appear in instantiated IO statements, once it uses an variable of value $undef$. (2) An $undef$ value within a condition of a `IF` statement causes `ABORT UNDEF`, since the condition can not be evaluated and the program path could not be determined. (3) statements `RETURN`, `RAISE EXCEPTION` within the procedure code are ignored, since computations on stale data might result into wrong rejections or wrong returns. However, client results are always verified by the server, such that erroneous or useless client computations do not lead to inconsistencies at the server.

Accordingly, we require from the function $eval_C(s, \#s, e)$ to: (1) Return $undef$ for all $s \notin E_C$. (2) Return $undef$ for all e that contain $undef$ value, e.g. $insert(R, undef)$ or `SELECT * FROM R WHERE X=`$undef$ are not defined.

3.2 The Clients *eval*() Function

The clients $eval_C(s, \#s, e)$ function dynamically handles the placement of fragments. That is, once a fragment is not available, but required for an execution, it is requested from the server. For this, we assume a function $frag(e) =$

$\{F_1, \ldots, F_k\}$ that for an instantiated IO statement e returns all fragments that are required to execute e. Note that the function depends on the fragmentation scheme, the query and update language.

Each execution of a procedure at the client starts on an *initial* versions v_k of fragments that corresponds to the data at completion time of the procedure that increased the version to v_k. The clients $eval_C(s, \#s, e)$ function is defined as follows:

1. if $s \notin E_C$ return $undef$
2. if a fragment $F_k \in frag(e)$ is not hosted at the client perform `ABORT DATA` and request F_k from the server
3. if e contains $undef$, perform `ABORT UNDEF`, since IO statements on $undef$ values are not defined
4. $ver_C :=$ all tuples (F_k, v_k) with fragments $F_k \in frag(e)$ of version v_k, (note that $version(F_k)$ does not change during the execution of S and that fragment data correspond to *initial* versions)
5. compute $val := eval(s, \#s, e)$
6. if e is a query, put $(eid, s, \#s, e, val, ver_C)$ into the servers cache
7. return val

An `ABORT` stops the execution at the client, since necessary fragments are not available yet or the execution is not defined due to $undef$ values. No more results are put into the cache. Note that a client at run time does not necessarily execute all $s \in E_C$, e.g. `IF <cond> THEN` s_1 `ELSE` s_2 `END IF` executes either s_1 or s_2. Further, clients might perform erroneous updates on local data that are not consistent with the data on the server. Such inconsistencies are removed during the next data synchronization by the server (c.f. Section 3.4).

Clearly, choosing E_C has a direct impact on the occurrences of $undef$ values and therefore on aborted client executions. Choosing E_C is discussed in Section 4.

3.3 The Servers *eval*() Function

From a client C_i the server receives the original request S and the execution ID eid. In parallel to the client, the server executes S within a single transaction and maintains fragment versions. Whenever possible, the server tries to re-use query results that have been put into the cache by the client. The servers $eval_S(s, \#s, e)$ function is defined as follows.

1. $ver_S :=$ all tuples (F_k, v_k) with fragment $F_k \in frag(e)$ of version v_k (note that, due to the transactional execution at the server, procedures always start on *initial* versions)
2. if e is low-level update, put $(eid, s, \#s, e', \emptyset, ver_S)$ into the cache (maintain servers update log)
3. if $s \in E_C$ and e is query
   (a) retrieve tuple $(eid, s, \#s, e', val, ver_C)$ from result cache for the given eid, s and $\#s$,

(b) if none exists, jump to 4
(c) if $e \neq e'$ jump to 4 (syntactic different queries)
(d) if $ver_C = ver_S$ jump to 5 (client and server operated on equal data)
4. $val := eval(s, \#s, e)$
5. return val

If a cache result is not available yet (3b) and therefore not executed by the client, the server executes the statement by himself. Note again, that clients do not necessarily execute all statements in E_C, e.g. `IF <cond> THEN` s_1 `ELSE` s_2 `END IF` executes either s_1 or s_2. Hence, query results not not necessarily appear in the cache.

Once the queries e and e' executed by client and server differ syntactically (3c), we assume that they yield a different result. Note that `SELECT * FROM R WHERE X>5` and `SELECT * FROM R WHERE X>6` yield the same result, if there is no tuple with `X=6`. We do not consider such cases here. Due to the restriction on queries and updates (c.f. Section 2.3) that prevents side-effects and the use of *initial* versions, the client result *val* can be validated on a version basis (3d). Hence, *val* equals to the result of executing $eval(s, \#s, e)$ at validation time and therefore is valid on the server, if client and server have started the execution of S on equal data (initial fragment versions).

Only in case a client result is invalid, the server has to re-execute the query. The amount of re-execution is further discussed in Section 4.

After the execution of S completes at the server, version numbers are increased of those fragments that have been modified. Finally, the result of S is passed to the client and the transaction (started for S) commits.

3.4 Synchronization

After a transaction of a procedure S has committed, the server synchronizes client data. The basic idea is to set client data to *initial* versions of fragments which is required by the result verification.

Whenever a client fragment F_k is updated to a newer version, current running procedures at the client are not affected and keep operating on the same data. Hence, new versions of F_k are only visible to client executions, once the update of F_k has been completed. While working on previous versions of data, clients might compute erroneous results which are detected by the servers verification.

We use an optimistic log-transfer technique [4, 12], where changes (low-level updates) on server data are transferred to clients. For this, we use the updates that are logged in the servers cache. It is optimistic, since clients might operate on stale data, and since clients do not enforce consistency of their local data, e.g. by locking server-side tables. The synchronization is done in 3 phases:

1. Let F_k be all fragments that have been changed by S and C_i all clients that host fragments F_k. Let v_k be the clients version number of F_k and v'_k the new version caused by the server. First, the server undoes all local updates on F_k on clients C_i that have been applied after the last synchronization for

version v_k. Then, it retrieves all updates of the execution ID *eid* (updates applied during executing S) from the servers cache and applies them on client fragments F_k. Since these are all updates that are executed up to the completion of a procedure, clients C_i host *initial* fragment versions v'_k.
2. Let C_i be the client that delivered results to the execution of S and F_k be all client fragments of version v_k that have been modified by C_i. If a F_k has not be modified by the execution of S at the server, all client updates on F_k are undone, thus achieving the original *initial* version v_k at C_i. Client fragments F_k that have been modified by the server are handled by (1).
3. Once all client fragments are updated, those updates and query results with *eid* are removed from the cache.

Clearly, the synchronization overhead is determined by the applications behavior and the selection of parameters E_C, since they are used for fragment placement (c.f. Section 4).

4 Case Study

The aim of this case study is to discuss parameters that influence the execution time of procedures and to show that the execution time improves, once the parameters $E_C(C_i, S_j)$ have been chosen appropriately.

4.1 Parameters Influencing the Execution Time

In general the execution time of procedures of a database application is determined by server hardware, structure and amount of data, complexity of the procedures, use behavior, etc. Beside these, our execution scheme depends on hardware and parameters E_C as follows:

1. The tight coupling of client and server requires fast networks. Hence, our approach applies not to Internet clients, but to in-house office and company networks.
2. Client fragments that are frequently updated, cause more synchronization overhead, more inconsistencies (due to optimistic synchronization) and more re-executions at the server. Note, that parameters E_C determine which fragments are placed at clients (c.f. Section 3.2). Hence, the synchronization effort also depends on the parameters E_C.
3. Fast client computations (hardware) and an efficient validation technique (as proposed in Section 3.3) reduce the execution time.

To improve the execution time, parameters E_C must be chosen properly. This is a complex task that we can not discuss sufficiently here. Instead, we introduce the two main problems and evaluate the feasibility of our approach for a real world application (next Section).

The first problem is to find those $E_C \subseteq E(S)$ that do not cause `ABORT UNDEF` operations (c.f. Section 3.2), since then IO statements are not executed at the

client. Hence, such E_C depend on the possibility to *parallelize the code* and to find IO statements that can be executed independently at client and server (e.g. [17, 1]). Recall example in Section 1, where the identifiers for IO statements ($E(S)$) might be enumerated by the line of code they appear. Then, $\{5\}$, $\{1,3\}$, $\{1,3,5\}$, $\{1,3,5,8\}$, etc. are possible values for E_C that do not cause `ABORT UNDEF`. But $\{3\}$ does, since the execution of $1 \notin E_C$ at the client (`SELECT * FROM Session WHERE id=sessionID`) would assign $undef$ to the variable `@session`. Then the execution of `IF` $undef$ `is not valid THEN ..` is not defined and causes `ABORT UNDEF`. We compute such E_C by analyzing the procedure code at compile time. Our method takes the program path, local variables and data access within the code into account.

Once, such E_C have been identified, the second problem is their appropriate selection at run time. For different applications, hardware and load conditions, each of them might lead to a different execution time and we are interested in those E_C that lead to a minimal execution time. In the following section we show how different E_C parameters affect the execution time.

4.2 Experimental Results

The application is a real world Internet shop with product groups (#814), products (#7386), a shopping cart, paying, etc. and business processes, such as login, register, add-to-cart, buy, clear-shopping-cart, browse catalog, etc.

As typical for many medium-sized web applications it is running under MySQL and PHP. Its architecture is depict at Figure 1. The database consists of 26 tables and about 64MB table data (including media files). A library LIB of business processes provides 195 PHP procedures for interacting with the database, each having between 4 and 140 lines of code.

A user request of a HTML page is computed as follows: (1) a HTTP request is send to the web server (2.0GHz CPU/1024MB RAM/Linux) which executes appropriate PHP scripts that produce HTML and might call several procedures S at the database server for handling the content of pages and updates on data, (2) each S is executed at the database server (dual 1.4GHz CPU, 512MB RAM, Linux) and passes results to the web server. Both machines are connected by a 100MBit network.

The web server is a client to the database server and is used for the parallel cooperative execution of procedures. Thus, running a MySQL database for locally executing procedures. Since tables did not exceed 10.000 tuples and a size of 20MB, we did not applied table fragmentation and placed full tables at the web server, if required by the local execution.

We have chosen a subset of the procedures in LIB for testing our cooperative parallel execution scheme. Some of the procedures could not be tested, since they did not match to the restrictions of our update language (c.f. Section 2.1). We manually derived its pre-compiled version and implemented the clients and servers execution engine with PHP. Note that each PHP procedure is performing SQL statements s by using an function `mysql_exec(`s`)`. This function equals to the concept of our $eval(.)$ function. Hence, by replacing this function, we had

only a little effort to integrate the clients and servers *eval*(.) functions. For each pre-compiled procedure we considered all possible splits $E_C \subseteq E(S)$.

For applying our execution scheme, step (2) is modified as follows: (2a) Once the web server calls a procedure S, it is partially executed according to some E_C on the local database (c.f. Section 3.1) and query results are put into the database servers cache. (2b) The database server in parallel executes S and retrieves cached results (c.f. Section 3.2). The cache has been implemented as a separate process at the database server that accepts results from clients and retrieves the result whenever they are requested from executions at the server.

Within various experiments we measured the execution time of pre-compiled procedures for different E_C under different load conditions. One of the procedures (`AddCart`) and its behavior has been introduced in Section 1. As the experiments show, different load causes different E_C parameters to be the most efficient. As a tendency we observed for the given application that the highest improvement of the execution time results from splits E_C that perform most of the queries at the web server and therefore also put most of the tables at the web server. One reason is that the web server could compute most of the queries faster than the database server. The other reason is that queries could often be executed in parallel to updates executed at the server (see example in Section 1).

Due to the fast computer and network hardware, the synchronization effort for all tested E_C was relatively low and did not meaningful influence the execution time. Similar behavior has been observed for 2 and 3 web servers.

Summarizing the experimental results we present in Figure 2 the improvement of the execution time that has been achieved for pre-complied procedures. It shows the total average over the execution time of all pre-complied procedures before and after our execution scheme has been applied. The figure represents the

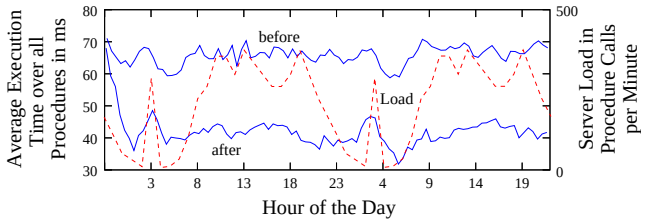

Fig. 2. System Performance *before* and *after* applying our execution scheme.

systems changing load of a 48h cycle according to the user behavior (maximum 10 procedure calls per second). The peak at night results from computing daily statistics. The curve *after* shows the execution time of those E_C that perform best for the given load. For all of those E_C the amount of re-execution (in case of erroneous client computations) was below 1%.

As a result we could show that a considerable speed-up is possible for the given application, once the parameters E_C have been set up properly.

5 Related Work

In the area of relational databases, several techniques for client-side caching and concurrency control has been proposed ([18, 16, 6]). The idea is that frequently used data constitutes a form of dynamic data replication that adapts to the clients interests. Those approaches are able to reuse locally cached data for associate query execution. Allowing clients to perform local updates or even the execution of full transactions imposes problems of client-side concurrency control. Then appropriate techniques (e.g. two-phase locking) have to be applied. In [6] a caching scheme has been combined with concurrency control techniques (e.g. locks), such that a client is able to execute a full transaction. The drawback of such a technique is that it requires a rather sophisticated concurrency control that has to scale well even for a high number of clients and that takes into account the unreliable nature of clients (e.g. disconnect from network, shutdown, etc.).

To avoid such problems we have proposed an optimistic approach where clients are *not* involved in concurrency control and where the server *alone* is responsible for handling transactions. Further, clients are able to perform only a *part* of a transaction locally. Hence, certain data (e.g. frequently updated tables) need not be handled by clients and therefore safe communication and synchronization effort. Optimistic approaches are also used for data intensive mobile applications, e.g. [15, 8, 4].

6 Outlook and Future Work

We presented a framework that is able to balance the execution of procedures between clients and a central server. Within a case study we demonstrated its feasibility and showed that for a real-world application the parameters could be set up accordingly, such that the execution time of procedures improves.

The main contributions of the paper are (a) a parallel cooperative execute scheme for stored procedures that (b) provides split parameters E_C for its configuration to specific applications and load conditions.

However, there are some interesting and challenging tasks in order to fill the remaining gaps. We outline them briefly. (1) In order to improve our data distribution scheme, we should consider to integrate existing fragmentation (e.g. [13]) or client-side caching (e.g. [6]) techniques. (2) To cope with standard DBMS, the proposed language for procedures must be extended by concepts, such as cursors, sequences, triggers, etc. Also we plan to remove low-level updates, such that SQL statements must not be replaced by a loop. (3) The most challenging problem is to propose an run time optimization technique that automatically determines the split parameters, hence, is able to adapt to changing load conditions. For this we envisage genetic algorithms for non-stationary optimization problems (e.g. [14, 3, 5]). (4) Continue the experiments in a multi-client environment.

References

1. D. F. Bacon and R. E. Strom. Optimistic parallelization of communicating sequential processes. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 155–166, Williamsburg, VA, April 1991.
2. Bartosz Bebel, Johann Eder, Christian Koncilia, Tadeusz Morzy, and Robert Wrembel. Creation and management of versions in multiversion data warehouse. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 717–723. ACM Press, 2004.
3. Jürgen Branke. Evolutionary Approaches to Dynamic Optimization Problems. In Jürgen Branke and Thomas Bäck, editors, *Evolutionary Algorithms for Dynamic Optimization Problems*, pages 27–30, San Francisco, California, USA, 7 2001.
4. Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.
5. John J. Grefenstette. Genetic algorithms for changing environments. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature 2 (Proc. 2nd Int. Conf. on Parallel Problem Solving from Nature, Brussels 1992)*, pages 137–144, Amsterdam, 1992. Elsevier.
6. Arthur M. Keller and Julie Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB Journal: Very Large Data Bases*, 5(1):35–47, 1996.
7. T. Oszu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
8. E. Pitoura and G. Samaras. *Data Management for Mobile Computing*, volume 10. Kluwer Academic Publishers, 1998.
9. M. Rodgriguez-Martinez and N. Roussopoulos. Mocha: a self-extensible database middleware system for distributed data sources. In *In Proc. ACM SIGMOD Conf., Dallas, USA*, pages 213–224, 2000.
10. S. Dar, M. Franklin, B. Jonsson, D. Srivastava, M. Tan. Semantic Data Caching and Replacement. In *In Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 330–341, 1996.
11. S. Jurk and M. Neiling. Client-Side Dynamic Preprocessing of Transactions. In *ADBIS, 7th East European Conf., Dresden, Germany*, volume 2798 of *LNCS*, pages 103–117. Springer, 2003.
12. Yasushi Saito and Henry M. Levy. Optimistic replication for internet data services. In *International Symposium on Distributed Computing*, pages 297–314, 2000.
13. Suk-Kyu Song and Narasimhaiah Gorla. A genetic algorithm for vertical fragmentation and access path selection. 43(1):81–93, 2000.
14. K. Trojanowski and Z. Michalewicz. Evolutionary Algorithms for Non-Stationary Environments. In *Proc. of 8th Workshop: Intelligent Information systems*, pages 229–240. ICS PAS Press, 1999.
15. Gary D. Walborn and Panos K. Chrysanthis. Supporting Semantics-Based Transaction Processing in Mobile Database Applications. In *Symposium on Reliable Distributed Systems*, pages 31–40, 1995.
16. Y. Wang and L. A. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proceedings of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, pages 367–376. ACM Press, 1991.
17. C. Wen and K. Yelick. Compiling sequential programs for speculative parallelism. In *In Proceedings of the International Conference on Parallel and Distributed Systems, Taiwan*, 1993.
18. W. K. Wilkinson and M. Neimat. Maintaining Consistency of Client-Cached Data. In *16th International Conf. on VLDB, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 122–133. Morgan Kaufmann, 1990.