Integration and Evaluation of Multi-Instance-Precommit Schemes within PostgreSQL

Paolo Romano and Francesco Quaglia
DIS, Sapienza Università di Roma

Abstract

Multi-Instance-Precommit (MIP) has been recently presented as an innovative transaction management scheme in support of reliability for Atomic Transactions in multi-tier (e.g. Web-based) systems. With this scheme, fail-over of a previously activated transaction can be supported via simple retry logics, which do not require knowledge about whether, and on which sites, the original transaction was precommitted. Mutual deadlock between the original and the retried transaction are prevented via MIP facilities, which also support reconciliation mechanisms for at-most-once transaction execution semantic. In this article we present an extension of the open source PostgreSQL database system in order to support MIP. The extension is based on the exploitation of PostgreSQL native multi-version concurrency control scheme. We also present an experimental evaluation based on the TPC-W benchmark, aimed at quantifying the relative overhead of MIP facilities on transaction execution latency, system throughput and storage usage.

1. Introduction

The design and development of supports for reliable transaction management in (Web-based) multi-tier distributed systems is a fundamental issue for most modern applications, such as e-business. One complex representative case is when middle-tier servers drive the execution of distributed transactions involving multiple (autonomous) back-end sites, and one cannot rely on application level compensation mechanisms to guarantee atomicity despite failures. In this scenario, the employment of an Atomic Commit Protocol (ACP) is mandatory.

The mainstream ACP is the so called Two-Phase-Commit (2PC), which is based on the precommit state as the expression of transactional sites endorsement for successful execution of local data manipulation statements. For this protocol, several frameworks have been proposed in order to achieve integration with the multi-tier system organization, among which we can mention classical Transaction Processing Monitors [1] and the recent e-Transaction specification [2]. The base idea in all these solutions is to achieve reliability via mutual fail-over capabilities across middle-tier server replicas or incarnations. This is done via diffusion of precommit/abort logs across the middle-tier (before any commit/abort message is sent out) so to prevent that different server replicas take different decisions on a same distributed transaction, possibly leading to a violation of the atomicity property. From a formal perspective, this means reaching consensus on the outcome of the distributed transaction across the middle-tier.

On the other hand, the cost for achieving consensus can become unaffordable in case of large scale geographical distribution of middle-tier servers, like in Application Delivery Network (ADN) infrastructures [9], namely representative expressions of the edge computing paradigm in service oriented applications. In order to cope with this issue, we have recently presented an innovative (application transparent) management model for distributed atomic transactions, which is referred to as Multi-Instance-Precommit (MIP) [6], which has been used as the building block for the construction of multi-tier reliability protocols framed by the e-Transaction specification.

With MIP, each replicated middle-tier server instance can retry the execution of a given transaction (e.g. upon client re-transmission), without explicit knowledge (and therefore consensus) about the state of a previously activated instance (if any) of that same transaction. The two transaction instances do not incur mutual deadlock and are reconciliated at commit time just thanks to the capabilities offered by MIP, hence obeying at-most-once semantic. We note that the avoidance of explicit consensus on transaction outcome across middle-tier servers means avoiding the need for accurate failure detection capabilities across those same servers. This further strengthens the relevance of the MIP model. Concerning the design and formal correctness of multi-tier reliability protocols (i.e. e-Transaction protocols) based on the MIP model, we remind the readers to [6]. Instead, in this article we focus on the integration of MIP within PostgreSQL (version 8.1.3).

We describe in a methodic manner the issues associated with the integration of MIP within PostgreSQL, including non-intrusive modifications to existing database kernel subsystems (such as concurrency control). This description can be also used as a reference for possible integration of MIP within the kernel of other database systems, especially those oriented to multi-version concurrency control (natively adopted by PosgreSQL). Finally, we present an experimental study relying on the TPC-W benchmark [8], aimed at evaluating the overhead associated with MIP facilities, in terms of transaction latency and system throughput,

as well as in terms of extra storage for tuples metadata.

2. The MIP Model

There are two main aspects that differentiate the MIP model compared to traditional (distributed) transaction management schemes, one is related to transaction demarcation and concurrency control, the other one is related to precommit/commit logs.

Concerning the first aspect, the basic idea is to allow a transactional data manipulation request, to be (concurrently) activated, and effectively processed, multiple times. This would permit fail-over of a previously activated transaction instance without its preventive extermination. Overall, the following features characterize demarcation and concurrency control in MIP:

Transaction Demarcation. A MIP transaction is univocally associated with a MIP-TID, which is formed by the couple $< XID, XINST >$, where XID is a base identifier, and $XINST$ is the so called instance identifier. Multiple MIP transactions can have the same XID, but they cannot have the same $< XID, XINST >$ pair. We say that all the transactions associated with the same XID, but with different $XINST$ values, form a family of *sibling* transactions.

Concurrency Control. In case a MIP transaction T requires (read/write) access to some data item d previously accessed (written/read) by a not yet committed (e.g. precommitted) transaction T', T is granted access to the pre-image of d with respect to the execution of T' if (A) T and T' are both MIP transactions and (B) they share the same XID (i.e. they are sibling transactions). Hence, any update performed by a not yet committed MIP transaction T' is not visible to any sibling transaction T. Operatively, this means that a newly activated sibling transaction does not get blocked waiting for commit/abort of a previously activated one, due to data conflicts. This building block allows effective fail-over (with no need for accurate failure detection and explicit extermination protocols of the original pending transaction), to be activated in case whichever anomaly (also including performance failures) occurs along the chain of multi-tier components originally involved in the processing of the transactional data manipulation request.

Concerning the other aspect of differentiation with standard transactional schemes, namely the management of precommit/commit logs, the MIP model relies on a data structure called MIP-Table (MIPT). The objective of this data structure is to provide supports for both (A) reconciliation of sibling transactions (hence allowing at-most-once semantic) and (B) retrievability of sibling transactions (non-deterministic) results, in order to select the one associated with the data manipulation pattern representative of reconciliation. Overall, the database is required to maintain a MIPT for each family of sibling transactions associated with a given XID. In the following, we will denote with MIPT$_x$ the table keeping track of transactions with $XID = x$. The y-th entry of MIPT$_x$ stores the following information related to the transaction with $XID = x$ and $XINST = y$: (1) $state$: a value, in the domain $\{null, prepared, abort\}$, reflecting the current transaction state ($null$ is the default initialization value); (2) $result$: the (non-deterministic) output produced by the execution of the transaction. Each MIPT$_x$ also keeps a special field, namely MIPT$_x.req$ which records the (client) request content that gave rise to sibling transactions with $XID = x$. The latter information is useful in order to autonomously allow the database server to trigger fail-over actions (e.g. via a stub) through a request push mechanism towards the middle-tier, which can simulate the client retransmission (see [6]), and whose aim is to promptly yield to transaction commit/abort so to improve data availability (by timely releasing any lock held by a precommitted transaction). The MIPT is accessible via proper prepare/commit APIs, which are quite similar to standard `xa_prepare`/`xa_decide` services prescribed by the XA specification [7]. Via these APIs, the middle-tier server coordinating the execution of whichever sibling transaction can (i) prepare that specific instance, (ii) retrieve the state (and result) of all the sibling transactions currently registered within the MIPT, and (iii) converge to a univocally identified data manipulation path, representative of reconciliation, associated with the minimum $XINST$ value identifying a distributed transaction instance successfully prepared at all the involved sites.

3. Integrating MIP within PostgreSQL

3.1. Transaction Demarcation

PostgreSQL automatically and transparently assigns a unique scalar identifier TID to a transaction when it starts. Many components of PostgreSQL use TIDs in different ways, hence changing the way they are generated and associated with transactions, in order to support MIP demarcation, would not represent a viable option. To address this issue, we associate each MIP transaction with two identifiers, namely, the original TID selected by PostgreSQL and a MIP-TID, which is instead selected by the overlying application and passed as a parameter to PostgreSQL when the transaction is started up. To achieve this, we have extended the demarcation API with the user-level SQL command `BEGIN_MIP` $< XID, XINST >$. We note that allowing the MIP-TID to be defined externally to the database kernel is an intentional design choice since it allows the transactional management logic (e.g. at the application server side) to easily correlate different (distributed) transaction instances with different instances of a same client (re-transmitted) request. A similar approach is used in standard XA technology for allowing the coordinator of a distributed transaction to associated, at precommit time, an application selected identifier with the transactions executing at the different sites. The difference with our proposal is that we allow global identification to be anticipated at transaction start time. This will be reflected in the way the re-engineered version of PostgreSQL manages sibling transactions during their whole execution. Concerning the association between the MIP-TID and the original TID, this has

been implemented via an in memory hash-table, indexed via MIP-TID values, which also allows retrieving the MIP-TIDs and the TIDs of all active sibling transactions. As it will be discussed, this is required while handling the reconciliation phase among sibling transactions during the commit phase.

3.2. Concurrency Control

PostgreSQL, as well as several other mainstream commercial DBMSs (such as Oracle [4]), implements a multi-version concurrency control scheme. This is achieved by creating a new version of a tuple whenever a write operation is executed on it, and by letting read operations access the most recent committed version of the tuple at the time the transaction started. PostgreSQL allows the existence of at most one uncommitted version of each tuple, which we refer to as the *active* version. Instead, the most recent version generated by a committed transaction is referred to as the *valid* version. To determine tuple visibility and detect conflicts the concurrency control scheme maintains within the metadata associated with each tuple version a couple of TIDs, namely $< t\_xmin, t\_xmax >$, which represent the identifiers of the transactions that, respectively, created and updated that tuple version, and a pointer, namely t_ctid, which links the tuple to the successive version, if any. Accordingly, when a transaction T_i creates an active version of a tuple, the tuple t_xmax value is set to the special value $null$, while the t_xmax value associated with the valid tuple version is set to T_i's TID and its t_ctid is linked to the active version. At starting time, each transaction T_i identifies its database *snapshot*, which is determined by its own TID as well as by its set of concurrent transactions, defined as those transactions that were already active upon activation of T_i (whose TIDs are stored within the in-memory transactional context of T_i) plus any transaction possibly activated after T_i (i.e. having TID greater than T_i's TID).

The concurrency control mechanism exploits the above described data structures to handle read/write operations as follows:

Read - upon read access to a tuple by transaction T_i, the history of committed tuple versions is used to retrieve the most recent tuple version committed by a transaction not concurrent with T_i (i.e. the version having maximum t_xmin value among the versions created by committed transactions not concurrent with T_i). It follows that the selected tuple might correspond to a version older than the valid one. On the other hand, if the read request is for a tuple previously written by T_i, it is satisfied by accessing the active version previously created by T_i itself.

Write - upon write access to a tuple by transaction T_i, the following version checks are performed: (1) If the valid version was created by a transaction concurrent with T_i (i.e. t_xmin on the valid version is the TID of a transaction concurrent with T_i), the abort of T_i is immediately forced. Otherwise, there are two cases: (2.A) There is currently no active version of the tuple. In this case T_i requests an exclusive lock on the valid version. If the exclusive lock is granted without any wait, T_i creates the active tuple version, which is used for any successive access by T_i. Otherwise, upon being woken up from the wait phase, T_i starts again the whole version checking. (2.B) An active version of the tuple exists. In this case T_i is queued for future access to the exclusive write lock associated with the tuple, and the whole version checking is repeated when T_i resumes.

Regarding the mechanism used by PostgreSQL for managing exclusive locks, an in memory lock table is used to store information on waiting transactions. Specifically, the lock table is indexed via transaction TIDs, and each entry records the TIDs of transactions waiting for the termination of the transaction indexing that entry. A tuple is considered locked by setting its t_xmax to the TID of the locking transaction. When a transaction completes (thus releasing its locks), it wakes up any transaction currently waiting on its corresponding lock table entry. In this way, per-transaction, rather than per-tuple, locking data structures are used, hence improving scalability in the management of the locking mechanism.

Beyond the above mechanisms for exclusive write locks, PostgreSQL also supports shared locks, which can be requested for, e.g., ensuring foreign keys integrity constraints, or upon explicit application request. In this case, a transaction waiting for the release of a shared lock may have to wait for the termination of a set of transactions. The association between a tuple and the TIDs of transactions holding the shared lock on it relies on indirection mechanisms. Essentially a so called MULTIXACT_ID is stored within the tuple header which is used as an indexing information to access an external table (maintained on a disk file and cached in RAM), which stores the list of TIDs associated with transactions holding the shared lock.

The integration of the MIP model within PostgreSQL led us to alter the synchronization scheme in order to regulate concurrent accesses to any tuple (also in write mode) by multiple sibling transactions. This has been done by mostly exploiting facilities already available within the database kernel in order to allow the modified synchronization scheme to efficiently and non-intrusively coexist with the native PostgreSQL concurrency control mechanism, and with the treatment of non-MIP transactions. From a methodological perspective, our solution is based on two new lock types, which we refer to as *Sibling-eXlcusive* (SX) and *Sibling-Shared* (SS). SX and SS locks can only be requested by MIP transactions, whereas the original Shared (S) and eXclusive (X) locks can only be requested by non-MIP transactions. The below table shows the compatibility of SX, SS, S and X locks:

	S	X	SS	SX
S	Yes	No	Yes	No
X	No	No	No	No
SS	Yes	No	Yes	Yes iff same XID
SX	No	No	Yes iff same XID	Yes iff same XID

Mutual compatibility between SX locks permits multiple sibling transactions to share the before-image of a given tuple, thus allowing the spawning of multiple active versions.

On the other hand, compatibility between SS and SX locks avoids mutual blocking situations between sibling transactions, which might otherwise compromise the timeliness of the fail-over phase and would require explicit preventive extermination of previously activated pending transactions. Finally, standard compatibility rules apply vs S and X locks, thus synchronizing MIP vs non-MIP transactions according to the native scheme adopted by PostgreSQL.

In order to support SS and SX locks, two fields, called XID and S_MULTIXACT_ID (each of 4 bytes), have been introduced within the tuple header. The XID field specifies whether the tuple valid version is currently locked by a MIP transaction (via either SS or SX locks). In the positive case, it also identifies the family of sibling transactions for which lock compatibility, as expressed by the above table, holds. The S_MULTIXACT_ID field is used in differentiated modes depending on the number of sibling transactions currently locking that tuple. In case only one of those transactions is active, it stores the transaction XINST forming the MIP-TID. Otherwise, it is used as indexing information (in a similar way to the previously discussed MULTIXACT_ID) to retrieve from a cached paged file the list of MIP-TIDs (and corresponding TIDs) of sibling transactions currently locking the tuple. The TIDs have been placed within that list in order to provide immediate identification of the parameters used by the lower level locking mechanism which, as discussed above, is based on a wait-for-TID policy. Also, given that the original tuple header maintains a single link (i.e. t_ctid) for the identification of the successive version (i.e. the active version in case of the valid tuple), multiple links required for coexistence and retrieval of multiple active versions associated with different sibling transactions have been also stored within such a list. We note that this external data structure does not need to be allocated in case of normal behavior (i.e. no failure, or suspect of failure in the execution of the original MIP transaction).

3.3. Precommit and Commit Phases

As pointed out in Section 2, the management of the precommit/commit phase of MIP transactions shows clear differences when compared to a conventional approach. These are mainly due to the fact that precommit logs must keep track of the (possible) precommit state of a family of sibling transactions. Also, the commit log must keep track of which one among the prepared sibling transactions has been eventually committed, as a result of the reconciliation scheme. This has required the development of an ad-hoc subsystem within PostgreSQL kernel, which is based on the MIP-Table (MIPT) data structure (see Section 2) as the base to address the previous issues. MIPT management has been non-intrusively integrated with typical kernel activities supporting generation and synchronous write of the Write-Ahead-Logs (WAL) [3].

Below we first describe the organization of the MIPT data structure and then provide insights on the related management activities. As a preliminary observation, similarly to the `BEGIN_MIP` statement, we have extended the SQL command set in order to support both prepare and commit requests for MIP transactions. Specifically, `PREPARE_MIP` $< XID, XINST >$ `'request_string' 'result_string'` can be used to request the database to precommit the MIP transaction associated with a specific MIP-TID, to atomically register the associated result and request strings within the corresponding MIPT entry, and to return the updated MIPT to the transaction coordinator. We recall again that, by explicit design choice, MIP-TIDs are selected by the overlying application (i.e. via the `BEGIN_MIP` statement - see Section 3.1), in a way to support mechanisms for correlating a specific request string with a family of transactions associated with a same XID. The usage of that family identifier in the prepare phase is reflected in a final association between the request string and the precommit log of that sibling transactions family. As hinted, this can even support database side retransmission activities (e.g. via a proper stub) in order to further speedup fail-over and increase data availability via prompt release of precommit locks. Analogously, the SQL command `COMMIT_MIP` $< XID, XINST >$ was introduced to support the final commitment of the MIP transaction representing the reconciliated execution path within that family, and to simultaneously request the abort of any other active or precommitted sibling transaction.

MIP-Tables. In order to ensure the scalability of the MIPT management logic, in our implementation MIPTs are maintained on a file residing on disk, which we refer to as *MIPT_data*, of which a small number of pages are explicitly cached in main memory to reduce I/O activity. To efficiently determine the position of the MIPT associated with a given family of sibling transactions within the MIPT_data file we use an indexing data structure, which we refer to as *MIPT_offset*, also maintained as a paged disk file cached in RAM. For performance reasons, we have structured the MIPT_offset index as a B-tree whose keys are transaction XIDs, and whose leaves contain the offset of the corresponding MIPTs within the MIPT_data file. We have used the B-tree since the keys correspond to application defined identifiers which can be generated in an arbitrary and uncorrelated (although univocal) manner. Therefore, the indexing data structure is not guaranteed to be accessed sequentially, which would lead to poor performance (due to reduced locality) in case it were implemented as a linear indexing data structure.

MIPTs are sequentially allocated within the MIPT_data file, and are composed by two main parts: (i) the header and (ii) the *results_area*. The MIPT header contains the following information:

1. $Sibs\_X\_MIPT$, namely the maximum number of sibling transactions that can be registered within the header without incurring an overflow ([1]).

[1] Overflows are tackled by allocating a new chunk for that same MIPT, which also includes a new header (identical to the original one, except for that it stores no request string) linked to the original one.

2. An array of size $Sibs\_X\_MIPT$, whose entries contain the following information: (i) the transaction $XINST$, (ii) the transaction state, and (iii) a pointer to the initial position of the corresponding result within the MIPT .

3. The actual number of the previous array entries that have already been used to register a sibling transaction, and the number of free bytes within the results_area.

4. A pointer to the initial position within the MIPT_data file of the memory area allocated due to the occurrence of an overflow, if any.

5. The request string associated with the family of sibling transactions.

Concerning the allocation of the results_area, its size is set to $Sibs\_X\_MIPT$ × `sizeof(result_string)`, where `result_string` is the result passed as input parameter to the `PREPARE_MIP` command that triggered the MIPT allocation. This simple heuristic is based on the idea that the results produced by sibling transactions are likely to exhibit similar size.

Precommit and Commit Log Management. To enable MIPTs recoverability and to guarantee the atomicity of transaction precommit and of the update of the corresponding MIPT, we rely on a conventional Write-Ahead-Logging (WAL) strategy [3]. More in detail, this is accomplished by writing the log entries describing the MIPT updates right after the typical log entries (related, e.g., to the locks maintained by the transaction) produced by the original precommit logic implemented within PostgreSQL, and just before emitting the *PRECOMMIT* log marker, whose presence on the log-file denotes that the transaction has been precommitted. On the other hand, updates of the MIPT_offset and MIPT_data files are performed only after having successfully flushed the transaction logs to disk. This avoids the need for undoing any update performed on the MIPT_offset and MIPT_data file in case of failure of log flushing.

Finally, during both precommit and commit phases, the database may be required to selectively abort active and/or prepared sibling transactions (this supports reconciliation). In our implementation, the abort of active sibling transactions is made possible by just retrieving the corresponding TIDs within the in memory hash table keeping track of the identity of each active MIP transaction (see Section 3.1). Conversely, in order to enforce the abort of precommitted sibling transactions, the corresponding MIPT is queried to retrieve the $XINST$ of the transaction to be aborted, so to reconstruct the internal identifier previously associated with the transaction when requesting its precommitment, which is used to request the abort through PosgtreSQL standard (internal) APIs.

4. Experimental Evaluation

Performance models in [5], have already highlighted how the avoidance of explicit consensus across middle-tier servers can increase system scalability and reduce end-to-end latency in multi-tier systems. Given that the MIP model is a building block for the avoidance of explicit consensus, and for additionally avoiding extermination schemes while handling fail-over, the performance analysis in [5] is representative of its performance benefits, compared to traditional transaction management schemes imposing coordination at the level of the middle-tier and extermination based fail-over. Hence, the experimental study in this section is rather aimed at quantifying both memory and computational overheads for the MIP-enhanced version of PostgreSQL. In order to assess such an overhead in a realistic scenario, our analysis is based on the well-known TPC-W [8] benchmark, representative of an on-line book store.

	Average tuple size (bytes)	Overhead %
Address	154.1	5.2%
Author	410.9	1.9%
CC_Xacts	126.6	6.3%
Country	63.2	12.6%
Customer	491.3	1.6%
Item	593.9	1.3%
Order	96.8	8.3%
Order_Line	115.8	6.9%
Weighted Average (smallest data-set)	163.8	4.9%
Weighted Average (largest data-set)	429.3	1.9%

Table 1. Overhead due to the extension of the tuple header for the TPC-W benchmark.

The main potential source of memory overhead in the MIP-enhanced version of PostgreSQL is related to the extension of the tuple header with the two additional fields XID and S_MULTIXACT_ID (both of size 4 bytes), which give rise to an increase of the size of the tuples stored and manipulated by the DBMS. To quantify the actual impact of this modification, we report in Table 1 the average percentage of spatial overhead for each of the database tables specified by TPC-W. In the last two rows of Table 1, we also report the average storage overhead, weighted according to the number of tuples in each table, when considering both the smallest and the largest data-sets specified by TPC-W, corresponding to 250k and 17M tuples, respectively. We note that the overhead introduced by the growth of the tuples header is actually very limited. Specifically, the average percentage overhead over the whole database is around 5% for the smallest data-set, and is below 2% for the largest data-set. This is because, in the former case, the Order_Line and Address tables, for which the average overhead is about 5%, account for 54% of the whole database. Conversely, in the largest data-set, the Item table tuples, whose average overhead is around 1%, take up about the 95% of the data-set size. The maximum overhead is introduced for the Country table, whose average tuple size is only 63 bytes. However, even in this (most unfavorable) case the overhead remains around 10%. Overall, we can conclude that in practical scenarios the additional memory consumption due to the increase of the tuple header size is expected to be very

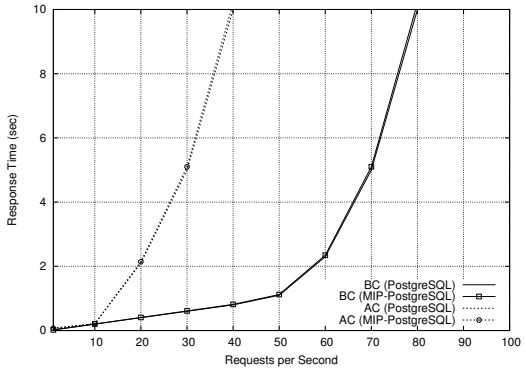

Figure 1. Execution time of non-MIP transactions on PostgreSQL and of MIP-transactions on the MIP-enhanced version.

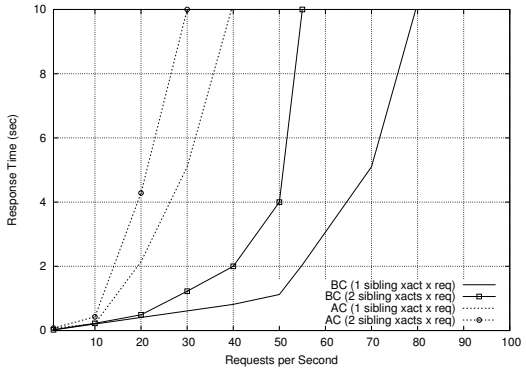

Figure 2. Execution time for fail-over transactions - 2 sibling xacts curve (original transactions in the precommit state) - vs the single sibling transaction case.

low, or even negligible.

In order to evaluate the latency overhead due to the employment of the MIP facilities, we have developed a prototype implementation of the TPC-W benchmark logic, based on JDBC. In this study, read-only transactions (which do not pose any reliability issue, and hence would not leverage MIP facilities) have been filtered out from the benchmark workload. Therefore, only non-idempotent transaction profiles have been considered. Such a choice allows us to evaluate a scenario in which MIP subsystems are used by every activated transaction so to spotlight their overhead. We plot in Figure 1 the results of a comparative performance test in which we contrast the response times of the unmodified PostgreSQL 8.1.3 and of the corresponding MIP-enhanced version, while processing two different TPC-W transaction profiles, namely *Buy Confirm* (BC) and *Admin Confirm* (AC). These are representative of lightweight and heavyweight transactional logics, respectively. Also, these performance data have been obtained for the largest dataset prescribed by the benchmark. The performance results were obtained by hosting the database server on a 4 CPUs - Xeon 2GHz - machine equipped with 4GB of RAM, 2 SCSI disks (10000 RPM) in RAID-0 configuration, and running the Linux operating system (kernel version 2.6.8). By the plots we get that the performance of the MIP-enhanced version is nearly undistinguishable from that of the original PostgreSQL version (the difference is about 2% over the whole curve), thus providing indications on the actual efficiency of the previously described MIP subsystems.

To further analyze the performance of the MIP subsystems, we have evaluated the transaction execution time also in the case where the data structures external to the tuple header (i.e. the table pointed by S_MULTIXACT_ID - see Section 3.2) are really allocated by the database kernel. This does not occur if a single sibling transaction of a given family is executed (as in the previously described test). To reach such a configuration, we execute the original transaction by leaving it pending in the precommit state. Next, we activate a fail-over sibling transaction, by also committing it, and we evaluate its execution latency. In such a case, the volume of requests we consider along the x-axis expresses half of the real transaction workload on the database (since each request is actually served via two sibling transactions). Interestingly, from the plots in Figure 2 we get that the system throughput gets actually reduced by only the 33% for the BC transaction profile, and the 25% for the AC transaction profile. This is because the data access patterns of sibling transactions show strong similarities, hence most of the data accesses performed by the fail-over transactions result in database buffer hits. This also explains the relatively smaller throughput reduction for the AC transaction profile, which requires accessing a larger number of data items in read mode with respect to the BC transaction profile. Independently of these considerations, the performance data in Figure 2 provide an experimental evidence of the efficiency of the MIP subsystems we have integrated within PostgreSQL.

References

[1] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing: for the Systems Professional*. Morgan Kaufmann Publishers Inc., 1997.

[2] S. Frølund and R. Guerraoui. e-Transactions: End-to-end reliability for three-tier architectures. *IEEE Transaction on Software Engineering*, 28(4):378–395, 2002.

[3] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[4] Oracle Corporation. *Oracle 9i replication*. 2001.

[5] F. Quaglia and P. Romano. Ensuring e-Transaction with asynchronous and uncoordinated application server replicas. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):364–378, 2007.

[6] P. Romano and F. Quaglia. Providing e-Transaction guarantees in asynchronous systems with inaccurate failure detection. In *Proc. of the 5th Symposium on Network Computing and Applications (NCA)*, pages 155 – 162. IEEE Computer Society Press, 2006.

[7] The Open Group. *Distributed TP: The XA+ Specification Version 2*. 1994.

[8] Transaction Processing Performance Council. *TPC Benchmark^{TM} W, Standard Specification, Version 1.8*. Transaction Processing Perfomance Council, 2002.

[9] A. Vakali and G. Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 07(6):68–74, 2003.