Semantics of a Runtime Adaptable Transaction Manager

Florian Irmert, Frank Lauterwald, Christoph P. Neumann, Michael Daum, Richard Lenz, Klaus Meyer-Wegener
Department of Computer Science, Chair for Computer Science 6 (Data Management)
University of Erlangen-Nuremberg, Germany
{florian.irmert, frank.lauterwald, christoph.neumann, michael.daum, richard.lenz, kmw}@cs.fau.de

ABSTRACT

Database Management Systems (DBMSs) that can be tailored to specific requirements offer the potential to improve reliability and maintainability and simultaneously the ability to reduce the footprint of the code base. If the requirements of an application change during runtime the DBMS should be adapted without a shutdown. Runtime-adaptation is a new and promising research direction to dynamically change the behavior of a DBMS. Especially the adaptation of the Transaction Manager (TM) states a challenge.

In this paper, we present the session semantics of a runtime-adaptable TM. We define preliminaries and assumptions to activate the TM during sessions from a conceptual point of view. The advantages and disadvanteges of different approaches are discussed, especially regarding the occurence of ANSI SQL phenomena. From a technical point of view, we define requirements for the architecture of the TM and the DBMS that arose in our prototype.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases, Transaction processing*

General Terms

Theory, Design

Keywords

Runtime, Adaptation, Transaction Management, SQL Phenomena

1. INTRODUCTION

Many database vendors drive innovations by adding new functionality to their existing products with every new release. This trend results in an increasing product size of the Database Management Systems (DBMSs). In order to apply a DBMS in embedded or pervasive systems, customized DBMSs have to be implemented. Such customized DBMSs cause rising development and maintenance costs. In order to provide a sustainable solution, the advantages of a small customized (individually implemented) realization and the comprehensive functionality of traditional DBMSs have to be combined. Due to specific application semantics and reduced synchronization in embedded systems, particular parts like the transaction management are often not necessary. A lightweight DBMS without transaction management that increases throughput and reduces response time is therefore often more suitable for embedded systems. During the lifetime of a system the complexity (required functionality) tends to increase over the years and a transaction management may be necessary in the future. Thus, we propose the seamless addition of the transaction management into a running DBMS. The need for deactivating the Transaction Manager (TM) at runtime seems to be much smaller.

Gray postulates "if every file system, every disk and every piece of smart dust has a database inside, database systems will have to be self-managing, self-organizing, and self-healing" [9]. To provide a basis for this vision, a DBMS has to be adaptable at runtime. Nowadays changing the functional range of a DBMS requires a shutdown and redeployment of a new version. Taking a DBMS offline is often not feasible in some environments. In the majority of cases, more than one application relies on a single DBMS and for most of them it is crucial that the DBMS is working; i.e. it is particularly important for DBMSs to be "always up". An in-depth evaluation of the consequences of runtime adaptation is required to provide seamless operation in a dynamic environment.

The CoBRA DB (**Co**mponent **B**ased **R**untime **A**daptable **D**ata**B**ase) project [11] investigates modular DBMSs that can be adapted to different environments by assembling prefabricated CoBRA DB components. The adaptation of the transaction management poses a prominent challenge in this context. The architectural encapsulation is presented in [12] and the focus of this paper is to resume the discussion with regard to semantic issues. We propose different possibilities to activate the transaction management at runtime and discuss their pros and cons regarding anomalies and isolation levels.

2. INITIAL SCENARIO

Before we start discussing the semantics of activating the TM at runtime we introduce the behavior of a DBMS that is running without a TM.

2.1 Failure Semantics without TM

A DBMS without transaction management cannot guarantee the ACID properties atomicity, consistency, isolation, and durability. Because most database environments are used to transactional support, some consequences of missing transaction support are outlined:

- Atomicity: The occurrence of DBMS failures or system failures may lead to partially written data, rendering even a single statement non-atomic. Additionally, application failures may occur between several statements that are logically grouped (would-be transactions). This makes it impossible to guarantee atomicity for a whole transaction even in the absence of system failures. Furthermore, without TM an application cannot request roll-backs.
- Consistency: Again, DBMS failures or system failures may lead to data loss, leaving the database in a physically inconsistent state. Logical consistency, like constraints, may be enforced but they cannot be deferred until the end of a transaction. Application-specific consistency rules are difficult to implement because of the missing atomicity: In a DBMS with a TM, an application may check its rules and decide to just roll back if it is not satisfied with the state of the database. This is not possible without a TM and an application would have to provide compensations by itself.
- Isolation: Multiple applications are not isolated by the DBMS. Yet, applications in a system environment may concurrently access the same data items by synchronizing themselves on application layer.
- Durability: The point in time at which data is written to persistent storage depends on the buffer strategy of the DBMS. Without a TM the application has no way to influence or request the flushing of memory buffers which is conceptionally a part of a commit. The DBMS has no way to inform the application that its data is durable as it is done by a TM by acknowledging a commit request.

Depending on the scenario and the specific requirements of the applications, such constraints may be acceptable. For example, if only one application is using the DBMS, concurrency issues are not relevant.

Even if a DBMS without a TM does not provide any warranties in general or has to individually define its warranties, it is necessary, for the lucidity of this discussion, to make minor assumptions about such a basic DBMS: At least, access to individual records has to be protected in a way that 1) no reader can see partially-written records and 2) no two writers can update a record at the same time. These assumptions are, for example, appropriate for a DBMS with a buffer system that implements latches. Such a DBMS provides a minimum protection for multiple applications even without a TM.

2.2 Active Sessions

Investigation of the transactional semantics of run-time adaptation of a TM is only necessary with active sessions that are already running before the TM is activated. There are three types of conflicts that can arise from multiple sessions: read-write, write-read, and write-write. Read-write and write-read conflicts may occur in the case of at least one active reader and one writer; write-write conflicts may only occur in the case of multiple writers. As all of these conflicts involve at least one writer, we can distinguish several cases based on the number of writers connected to the database. Trivial cases, like only one active session, are intentionally left out.

1. No writer and $n > 0$ readers
2. One writer and $n \geq 0$ readers
3. More than one writer and $n \geq 0$ readers

We do not restrict the pre-TM application environment in any way. Therefore, it is necessary to comprehend which problems may arise in each possible case, because a different set of phenomena can occur: In the first case, none of the ANSI SQL-phenomena are possible. In the second case, dirty reads, non-repeatable reads, and phantoms are possible. In the third case, additional phenomena like dirty writes and lost updates may occur. Dirty reads, non-repeatable reads, and phantoms are subject of the considerations in the SQL-standard [14]; dirty writes and lost updates are not considered.

If the application environment has turned out to be not properly synchronized, a basic reason to deploy a TM is present because providing the synchronization by the DBMS avoids the need to change the applications. If an additional application connects to the database, a TM may also be required for two reasons: First, the new application might change the scenario, e.g. from zero writers to one or from one to several, leading to new possible phenomena which might not be acceptable. Second, the new application may require a form of synchronization that would be too expensive to provide on application layer. In these cases, a TM has to be installed.

2.3 Programming Model

The behavior of common database applications is similar to the behavior of applications that are employed in a runtime-adaptable environment. A logical sequence of operations starts at least whenever a user starts a session. During a session, new logical sequences of operations start whenever he uses the transactional keywords *COMMIT* or *ROLLBACK*. According to the SQL-standard, the DBMS assumes *begin transaction* implicitly when a connection is established and after every *commit* and *abort*. Using transactional keywords without having a TM only leads to a markup of logical sequences. Without TM, *COMMIT* does not guarantee the ACID properties. As no *ROLLBACK* is possible without a TM initial applications may not use *ROLLBACK* at all.

When a TM is activated at runtime, the working environment for the applications changes drastically. A primary goal in distinguishing and analyzing several TM activation strategies is to find one that keeps the implications for application developers as small as possible. A premise in the discussion is that it is not possible to alter the client applications when the TM is activated or deactivated at runtime. Neither are client applications informed about the TM activation or deactivation. Consequently, applications send the same SQL commands to the server regardless of the availability of transactional support. Applications that do not need transactions should be able to run without as well as with a TM. Applications are not forced to use *COMMIT*. However, using *COMMIT* is advisable because a TM may otherwise group a complete application session to a single transaction which can lead to semantic problems, lock contention, or both. Applications that genuinely require transactional support may use *COMMIT* as well as *ROLLBACK* but they must never be executed in an environment without an active TM.

In conclusion, for an application that works without as well as with a TM, we require at least the following:

1. The application does not try to perform an abort, as an abort is not possible without a TM.
2. The application does perform commits in certain intervals. In the presence of a TM, this is required to prevent accumulation of locks. In this regard, our system does not differ from any other DBMSs with a TM.

2.4 Scenario

During this paper we assume the following scenario: One or more applications depending on a DBMS are deployed on an embedded device. At the time of deployment, a transaction manager is not necessary due to the requirements of the applications and would cause unnecessary load. During the lifetime of the system, other applications are deployed or the existing applications are extended in a way that transaction management is indispensable. This may be due to requirements for any of the ACID properties.

In this paper, we focus on the isolation property as it is the most challenging. The required degree of isolation may vary. This leads to the existence of several different isolation levels in SQL [14]. For lucidity, we assume that the TM has to provide the *SERIALIZABLE* isolation mode to all transactions while it is running; individual transactions with lower isolation requirements are subsumed by such discussion. Subject to the discussion are problems during the activation and deactivation that prevent the guarantee of strict isolation. Several activation strategies can be distinguished, each with different drawbacks regarding the transactional session semantics during the activation of the TM at runtime.

3. DYNAMIC TM ACTIVATION

In this section we discuss the different possibilities to activate a TM at runtime. We focus on the isolation property as it is the most challenging. Atomicity, consistency and durability each refer to a single transaction, while isolation describes the allowed and disallowed interactions between multiple sessions.

Activating the TM between SQL statements provides an intuitive TM activation model. Therefore the activation of the TM never interrupts running statements. In the context of TM activation, we distinguish between sessions that originate before TM activation (*pre-sessions*) and sessions that are initiated after TM activation (*post-sessions*). In our discussion about transactional semantics for TM runtime activation, we have to consider *pre-sessions* and *post-sessions* separately:

- It is possible to either create transactional contexts for *pre-sessions* with immediate effect (*immediate pre-session transactions*) or the first transactional context within pre-sessions is created after their next occurring *commit* (*deferred pre-session transactions*).
- All *post-sessions* are isolated by transactions, but we distinguish immediate execution of initial transactions (*immediate post-session transactions*) from deferred execution (*deferred post-session transactions*): post-sessions are deferred until all *pre-sessions* have *committed* after the TM activation.

3.1 Activation Strategies

In figure 1 we present our three different possibilities (”DD”, ”DI” and ”II”) to activate the TM at runtime. In each case the first timeline represents a pre-session and the second timeline a post-session. Individual SQL statements are denoted by U$_{ij}$ where i identifies the session and j numbers the statements. Likewise, C$_{ij}$ denotes a *COMMIT*. All three possibilities have in common that a pre-session is only considered and treated in special ways until its first commit after TM activation. After a pre-session has committed once during TM presence, the session's subsequent transactions are not treated differently from the ones of any post-sessions (demonstrated in U$_{15}$ to U$_{17}$ that follow the pre-session commit C$_{11}$).

- DD - deferred pre- and deferred post-session transactions

  The first scenario in figure 1 outlines the activation strategy *deferred pre-session transactions* and *deferred post-session transactions* (abbr.: DD). In this case, transactions of post-sessions are blocked until all pre-session have committed or terminated. In the example, the *commit* C$_{11}$ represents the commit of all pre-sessions. The activation of the TM is deferred until this point in time. After C$_{11}$ the TM is active for all sessions.

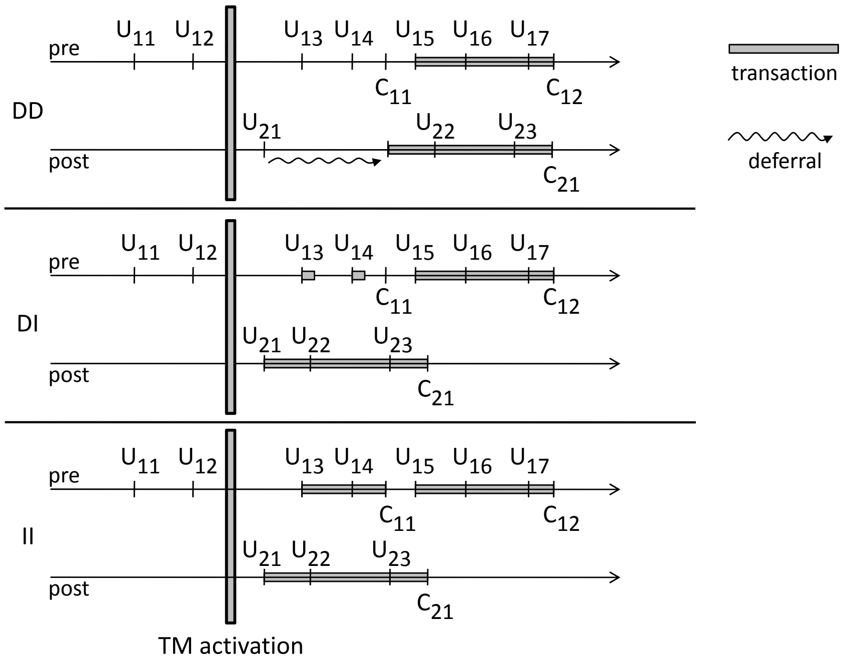
DD
pre
post
U_{11} U_{12} U_{13} U_{14} U_{15} U_{16} U_{17}
C_{11} C_{12}
U_{21} U_{22} U_{23}
C_{21}
DI
II
TM activation
transaction
deferral

Figure 1: TM activation semantics

- DI - deferred pre- and immediate post-session transactions

  The second scenario is an example for the activation strategy *deferred pre-session transactions* and *immediate post-session transactions* (abbr.: DI). Here a post-session can start a transaction directly after the TM activation without any deferral. Statements in a pre-session just acquire short locks. Thus, each statement in a pre-session is viewed as a transaction in its own right until the first commit occurs.

- II - immediate pre- and immediate post-session transactions

  The last scenario in figure 1 shows an example for *immediate pre-session transactions* and *immediate post-session transactions* (abbr.: II). After the activation of the TM a new statement starts a transaction in post-sessions as well as in pre-sessions.

3.2 Activation Strategy Evaluation

In this section we discuss the advantages and disadvantages of the activation strategies and their different behavior with regard to the three standard ANSI SQL phenomena *Dirty Read, Non-repeatable Read* and *Phantom.* Figure 2 shows the operation sequences (abbr.: OpSeq) for each phenomenon as they are described in [14].

We will discuss these scenarios for our three activation strategies. We focus on conflicts between a pre-session and a post-session: conflicts between two pre-sessions are not affected by TM activation, while two post-sessions are always isolated from each other. By convention the first operation in a schedule is always performed by OpSeq1 (figure 2). Therefore we have to distinguish if OpSeq1 is executed in a pre- or post-session:

1. OpSeq1 starts in a pre-session and OpSeq2 in a post-session. In this case we have to distinguish two different TM installation times (abbr.: TMI) (see figure 2):
   (a) TMI_1: The TM is installed before the first operation of the schedule (scenario (1a)).
   (b) TMI_2: The TM is installed while the pre-session is already running but before the first operation of the post-session (scenario (1b)).
2. OpSeq1 starts in a post-session while OpSeq2 still runs in a pre-session (scenario (2)). Obviously in this case installation is only possible at TMI_1: As the first operation in the schedule is performed by post-session OpSeq1 and post-sessions always run under TM control, the TM installation has to be done before the beginning of the schedule.

DD - deferred pre- and deferred post-session transactions

The DD strategy does not allow pre-sessions and post-sessions to exist at the same point in time. The first statement in a post-session is deferred until all pre-sessions have committed (after a commit, they are no longer pre-sessions)[1]. Therefore none of the phenomena can occur with this strategy.

The drawback of this solution is caused by long running pre-sessions that do not commit. The activation of the

[1]Incidentally, Microsoft SQL Server does something similar when switching to snapshot isolation at runtime. See e.g. www.sqlteam.com/article/transaction-isolation-and-the-new-snapshot-isolation-level

Installation Time	Phenomena	DI	II
TMI_1	Dirty Read	No	No
	Non-Repeatable Read	Yes	No
	Phantom	Yes	No
TMI_2	Dirty Read	No	No
	Non-Repeatable Read	Yes	Yes
	Phantom	Yes	Yes

Table 1: Phenomena in (1a) and (1b)

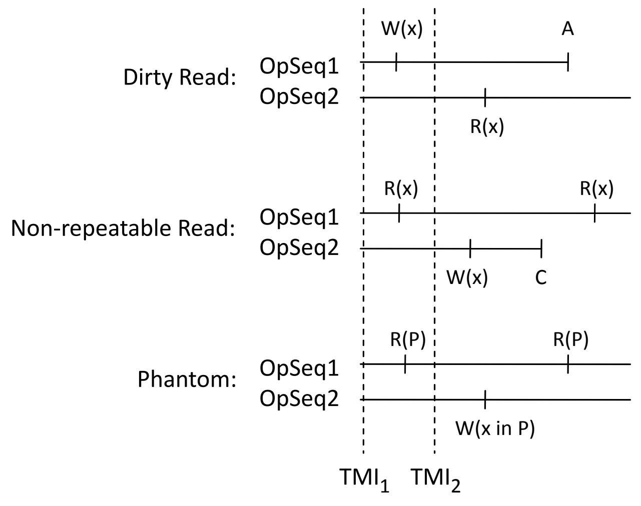

Figure 2: Phenomena

TM is deferred by such sessions for unpredictable time. Particularly in embedded systems, sessions may run without termination and therefore the TM will not be activated. However, this problem is not specific to a dynamically activated TM: If the TM were activated from the beginning, long-running transactions would keep locks for a long time, thus possibly preventing other transactions from running. The solution is requiring pre-sessions to commit from time to time as described in section 2.3.

DI - deferred pre- and immediate post-session transactions

In order to discuss the behavior of the DI strategy, we have to examine the three scenarios (1a), (1b) and (2) separately: In (1a) OpSeq2 can read an item x that has been modified by OpSeq1 before, because OpSeq1 only requests a short lock on x. Nevertheless, a *Dirty Read* is not possible because pre-sessions are not allowed to abort (see section 2.3) and therefore x will not be rolled back by a pre-session after it is read by OpSeq2. *Non-repeatable Read* and *Phantom* are possible because the read operation in OpSeq1 does only request a short lock on x, respectively on P. The scenario (1b) is in this strategy equivalent to (1a). A *Dirty Read* is not possible for the same reason and *Non-repeatable Read* and *Phantom* are possible because the read operation in OpSeq1 does not request a lock. In (2) none of the phenomena are possible. OpSeq1 request a lock on x, respectively P, and when OpSeq2 requests short locks it has to wait until OpSeq1 releases the lock.

In this strategy, transactions and SQL commands that are not protected by a transaction are executed concurrently. This necessitates a discussion about possible deadlocks. While a deadlock situation may usually be resolved by aborting any of the conflicting transactions, this may be impossible in our case, as pre-sessions cannot be rolled back. Thus, any conflict that can only be resolved by rolling back the pre-session may be unacceptable. Normally a deadlock occurs in a sequence like $R_1(x)$ $R_2(y)$ $W_1(y)$ $W_2(x)$. OpSeq1 waits for the lock on y and OpSeq2 waits for the lock on x. Due to the fact that in the DI strategy only short locks are requested for operations in pre-sessions, the transaction in the post-session is always able to commit. Therefore a deadlock situation is not possible.

The drawback of this solution is the loss of isolation between TM installation and the commit of the last pre-session. This may be problematic because during this time interval post-sessions are already allowed to run, leading to problems that were not possible before TM installation and in the absence of post-sessions.

II - immediate pre- and immediate post-session transactions

Regarding the discussed phenomena the II strategy is similar to DI. As in DI none of the phenomena can occur if OpSeq1 runs inside a post-session (scenario (2)). It only differs in the scenario where OpSeq1 starts in a pre-session and OpSeq2 runs inside a post-session (scenario (1a) and (1b)). In contrast to DI strategy, if the TM is installed at TMI_1, neither *Non-repeatable Read* nor *Phantom* are possible for the II strategy. The read operation in OpSeq1 requests a long duration lock due to the fact that it still runs inside a transaction. The phenomena that are possible in (1a) and (1b) are summarized for strategies DI and II in table 1.

Again, a discussion about possible deadlocks is necessary. Different from DI, deadlocks are possible in II because pre-sessions acquire long duration locks. In case of a deadlock it is necessary to roll back the post-session (Remember that pre-sessions cannot be rolled back. While it is technically possible to roll back a pre-session to the TM installation time in II, it seems unlikely that applications could handle this situation gracefully). The assumption that pre-sessions commit in reasonable intervals is of equal importance as it is for the DD strategy: Long-running pre-sessions might otherwise accumulate locks causing post-sessions to wait for an indeterminate amount of time.

Discussion

As already mentioned, only the DD strategy provides seamless isolation. If this is required, DD will be the only sensible choice. The differences between DI and II are a bit more subtle. Both strategies have problems with pre-session operations that occur before TM installation. Thus, their possible phenomena are identical when installing the TM at TMI_2. II does however provide better isolation for pre-session operations that occur after TM installation: While the same phenomena can occur in DI regardless of TM installation time, II does not suffer from any phenomena when installing the TM at TMI_1. This difference is explained by the fact that in DI pre-sessions acquire only short locks while in II they acquire long ones. It is noteworthy that the installation time cannot be influenced: The TM is installed as soon as the need for it arises. At this time, the pre-sessions may already have performed actions which permit phenomena.

3.3 Deferred Commit Strategy

In this section we present a fourth strategy, named *Deferred Commit*, that is able to avoid all phenomena and yet allows transactions in post-sessions to start immediately after the TM installation.

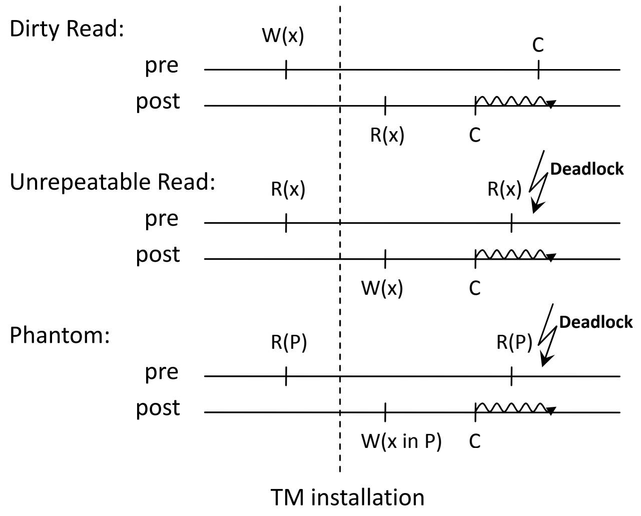

Figure 3: Phenomena

Deferred Commit is based on the II strategy with the difference that the commits of all post-sessions are deferred until all pre-sessions have committed. As it can be seen in figure 3, the deferral of the *COMMIT* in the post-session causes the pre-session to wait for the lock in the phenomena *Non-repeatable Read* und *Phantom.* In these scenarios the pre-session will never obtain the lock because the transaction in the post-session is not able to commit until all pre-session have committed. Therefore these scenarios result in a deadlock. In order to resolve the deadlock, the transaction inside the post-session has to be rolled back (see discussion in section 3.2, paragraph "DD").

The advantage of the *Deferred Commit* strategy is that all post-session are able to start their transaction immediately after the TM installation which reduces the deferral problems of DD. Transactions in post-sessions, however, still have to wait for the *COMMIT* of all pre-session before being able to commit. In conclusion, the DC strategy allows higher concurrency than DD by being able to process post-session operations while pre-sessions are still running.

4. RUNTIME-ADAPTABLE TM

The CoBRA DB architecture of the runtime-adaptable TM has been described in [12]. This section provides a short outline. The basic challenge of the CoBRA DB project is to modularize a DBMS architecture and to define interfaces for DBMS modules such that a modular DBMS can be adapted to different environments by assembling prefabricated CoBRA DB components. The decomposition and interface definitions in CoBRA DB target the creation of reusable DBMS infrastructure components for utilization in the development of arbitrary DBMSs projects. Since transaction management is a cross-cutting concern, CoBRA DB instruments aspect-orientation for its modularization. From a technical point of view, we used dynamic Aspect-Oriented Programming (d-AOP) [6] as basis for runtime adaptation. For proof of concept, we chose SimpleDB [20] with its well-described architecture as a basis, removed the TM, and re-integrated it with d-AOP.

A clear definition of requirements regarding the architecture of a DBMS with runtime-adaptable TM helps the understanding of the field of problems that arise from the separation. There are two perspectives of design autonomy, one from the viewpoint of the TM and another from the viewpoint of the residual DBMS.

DBMS autonomy: independence of the DBMS from the TM

For two reasons, it is required that the DBMS is independent from the TM: 1) The TM can be activated and deactivated on demand. 2) As the TM represents a cross-cutting concern in traditional database systems' architectures, many changes of different components might be necessary if the TM changes.

TM autonomy: independence of the TM from the DBMS

The independence of the TM from the other DBMS components is required by several reasons: 1) Changes to the underlying DBMS must not require changes to the TM. 2) The TM is developed once and deployed often in different customized DBMSs. This can minimize costs of maintenance and development. 3) In a modular DBMSs also the residual layers or components can be exchanged on demand. Such reconfiguration must not require the change of other modules like the TM. Therefore, the TM has to be designed such that it is affected as little as possible by adaption of other modules.

In contrast to the DBMS autonomy, the TM autonomy can only be achieved to a certain degree. Linking the TM to a DBMS depends intrinsically on the database system: especially the subsystem for recovery requires access to internal structures of the DBMS for accomplishing its task. Nevertheless, maintainability can be achieved by a subtle architecture as far as possible by encapsulating the code that takes part in the linkage in self-contained classes apart from the core logic of the TM.

Architecture outline

The linkage between the TM and the DBMS is represented by `Transaction` and `TransactionAspect` in figure 4. Each instance of `Transaction` represents exactly one transaction. `Transaction` is DBMS-independent. It is used to couple the operations of the underlying DBMS to the TM and for this purpose its interface provides operations like `commit` and `rollback`. The `TransactionAspect` realizes the DBMS-dependent linkage of `Transaction` to the DBMS. It contains the relevant advices, from aspect-oriented programming methodology, that invoke the corresponding methods of `Transaction`.

The `log` subsystem provides logging required by the recovery system and it is independent of the recovery strategy and mechanisms. The `concurrency` subsystem realizes the concurrency manager of the TM. `ConcurrencyMgr` provides the interface for `Transaction`; i.e. there is an instance of `ConcurrencyMgr` for each transaction. The implementation is DBMS-independent and the locking policy can be exchanged by switching the concurrency module with different implementations. Recovery management is represented by the `recovery` subsystem. It depends on the storage structure of the DBMS. In order to achieve DBMS-independence, the *Template Method Pattern* is implemented to achieve the autonomy of the essential recovery procedures. In our prototype, we use a pure undo recovery strategy that could be exchanged by a redo/undo-recovery strategy.

In order to facilitate a generic representation of data, being independent of the types that are present in the underlying system, a hierarchy of special classes in the subsystem `types` has been designed. `Serializable` represents a basic interface for serializing and deserializing classes. Classes that model types of the underlying DBMS additionally implement the `DBType` interface such that it provides a type identifier. The type identifier is used inside the log file. In conclusion, the DBMS-specific parts of the TM are represented by the `TransactionAspect` with its advices and some DBMS-specific `types` classes for log file serialization purpose.

Global Tracking of Transactions

One of the most challenging technical problems is the global tracking of transactions. Global tracking of transactions is necessary for relating method calls to corresponding transactions. This is required for the locking manager as well as commit processing and recovery. [13] has already discussed this fundamental problem without offering an acceptable solution.

Our solution is to use information about threads in order to track the transactions. We save the correlation between thread and transaction for each client session. With this solution, we can determine the context of a transaction for any method during runtime.

5. RELATED WORK

Realizing the need for adaptable DBMSs in specific environments respectively for specific applications is not new. Dittrich and Geppert [7] point out the drawbacks of monolithic DBMSs and propose componentization as a possible solution. They identify four different categories of CDBMSs (component DBMSs):

- Plug-in Components. The DBMS implements all standard functionality, and non-standard features can be plugged into the system. Examples for this category are Oracle Database [1], Informix [16] and IBM's DB2 [5].
- Database Middleware. DBMSs falling in this category integrate existing data stores, leaving data items under the control of their original management system, e.g. Garlic [19], Harmony [17], and OLE DB [4].
- DBMS Services. DBMSs of this type provide functionality in standardized form unbundled into services, e.g. CORBAservices [2].
- Configurable DBMS. DBMSs falling in this category are similar to DBMS services, but it is possible to adapt service implementations to new requirements or to define new services. An example is the KIDS project [8].

The principles introduced in the context of SOA (service-oriented architecture) are also suitable for building modular DBMSs. [21] present an approach towards service-based DBMSs. They adopted the architectural levels from Härder [10] and plan to include the advantages introduced by SOA like loosely coupled services. They define four different layers and propose their realization as services. It is argued that a DBMS that is built upon this architecture is easily extensible because services can be invoked when they are needed and in case of failure of services alternative services can answer the request. Tok and Bressan [23] also introduce a DBMS architecture based on service orientation, called SODA (Service-Oriented Database Architecture). In their DBNet prototype, web services are used as the basic building blocks for a query engine. In contrast to our approach these approaches use distributed services involving a coarse grained architecture and communication via RPC, RMI, web services or other technologies for distributed communication.

[22] discusses the advantages and disadvantages of the aspect-oriented methodology for DBMS architecture; for this purpose Berkeley-DB as a modular DBMS was refactored with standard Aspect-Oriented Programming (AOP) for maintenance purposes. The FAME-DBMS project [18] utilizes the Software Product Lines (SPLs) approach to implement customized DBMSs. "With this approach a concrete instance of a DBMS is derived by composing features of the DBMS product line that are needed for a specific application scenario" [18]. They use aspect and feature oriented programming to implement the SPLs. In the scope of this project, [13] refactored Berkeley-DB with AspectJ.

Another approach to develop a tailor-made DBMS is presented in [15]. The COMET DBMS (component-based embedded real-time database management system) is intended for resource constrained embedded vehicle control-systems. The customization is achieved by providing a set

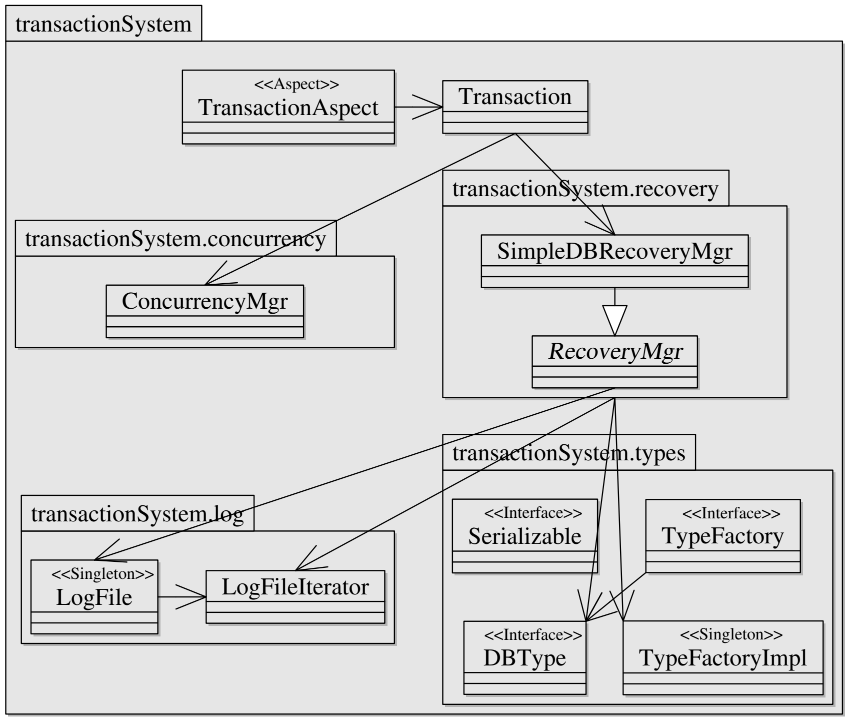

Figure 4: Architecture of the TM

of components that can be made up to a specialized DBMS by selecting components that meet the requirements. The partitioning and therefore the possibilities to customize are coarse grained as there are only seven components identified. The major difference to our approach is that these DBMSs are tailored at compile time and do not support adaptation at runtime. Therefore, to best of our knowledge, this is the first research discussion about the semantics of a runtime-adaptable TM.

6. CONCLUSION AND FUTURE WORK

Existing component- and service-based DBMSs have in common that changing their behavior often requires a shutdown of the whole system. In the CoBRA DB project we are implementing a DBMS that is adaptable at runtime. The addition of the TM at runtime poses a challenge due to its cross-cutting characteristics. We outlined the TM architecture of our prototype to provide a technical foundation for a runtime-adaptable TM.

The core of this paper is the introduction of different approaches to install a TM during runtime. We discussed the implications that arise in a DBMS that is working without TM and proposed a programming model regarding a prospective TM installation. Potential schedules of operations are examined with regard to the ANSI SQL phenomena: *Dirty Read, Non-repeatable Read*, and *Phantom.* Therefore the different schedules of these phenomena are discussed with regard to their execution inside the pre- and post-session. We concluded this discussion with the *Deferred Commit* strategy as the strategy that solves the most of the discussed problems.

In this paper we restricted our discussion to the three ANSI SQL phenomena. However there exist other semantical problems that are not discussed in the SQL standard [3]. We are currently investigating these phenomena with regard to our proposed solution.

7. REFERENCES

[1] S. Banerjee, V. Krishnamurthy, and R. Murthy. All your data: the Oracle extensibility architecture. *Component database systems*, pages 71–104, 2001.

[2] R. Bastide and O. Sy. Formal specification of CORBA services: experience and lessons learned. *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 105–117, 2000.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, 1995.

[4] J. A. Blakeley. Data access for the masses through OLE DB. *SIGMOD Rec.*, 25(2):161–172, 1996.

[5] J. Cheng, J.; Xu. XML and DB2. *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, pages 569–573, 2000.

[6] R. Chitchyan and I. Sommerville. Comparing Dynamic AO Systems. In *Dynamic Aspects Workshop (DAW 2004)*, Lancaster, England, März 2004.

[7] A. Geppert and K. R. Dittrich. Component database systems: introduction, foundations, and overview. *Component database systems*, pages 2–28, 2001.

[8] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of database management systems based on reuse. Technical Report ifi-97.01, University of Zurich, 1997.

[9] J. Gray. The Next Database Revolution. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2004)*, pages 1–4, New York, NY, USA, Juni 2004. ACM.

[10] T. Härder. DBMS Architecture - Still an Open Problem. In G. Vossen, F. Leymann, P. C. Lockemann, and W. Stucky, editors, *11. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW 2005)*, volume 65 of *LNI*, pages 2–28. GI, März 2005.

[11] F. Irmert, M. Daum, and K. Meyer-Wegener. A New Approach to Modular Database Systems. In *Software Engineering for Tailor-made Data Management (SETMDM 2008)*, pages 41–45, März 2008.

[12] F. Irmert, C. Neumann, M. Daum, N. Pollner, and K. Meyer-Wegener. Technische Grundlagen für eine laufzeitadaptierbare Transaktionsverwaltung. In *BTW 2009*, 2009.

[13] C. Kästner. Aspect-Oriented Refactoring of Berkeley DB. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, School of Computer Science, Department of Technical and Business Information Systems, Februar 2007.

[14] J. Melton. *ISO/IEC 9075-2:2003 - Foundation (SQL/Foundation)*. International Organization for Standardization (ISO), 2003.

[15] D. Nyström, A. Tesanovic, C. Norström, and J. Hansson. The COMET database management system. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-98/2003-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2003.

[16] M. A. Olson. Datablade extensions for informix-universal server. In *COMPCON '97: Proceedings of the 42nd IEEE International Computer Conference*, page 143, Washington, DC, USA, 1997. IEEE Computer Society.

[17] U. Röhm and K. Böhm. Working together in harmony - an implementation of the CORBA object query service and its evaluation. In *ICDE'99: Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 238–247, 1999.

[18] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. FAME-DBMS: Tailor-made data management solutions for embedded systems. In *Software Engineering for Tailor-made Data Management*, pages 1–6, 2008.

[19] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 266–275. Morgan Kaufmann, 1997.

[20] E. Sciore. SimpleDB: a simple java-based multiuser system for teaching database internals. In *38th ACM Technical Symposium on Computer Science Education (SIGCSE 2007)*, pages 561–565, New York, NY, USA, März 2007. ACM.

[21] I. Subasu, P. Ziegler, and K. R. Dittrich. Towards Service-Based Data Management Systems. In *Datenbanksysteme in Business, Technologie und Web (BTW 2007), Workshop Proceedings*, 3-86130-929-7, pages 296–306, March 2007.

[22] A. Tešanović, K. Sheng, and J. Hansson. Application-tailored database systems: A case of aspects in an embedded database. In *8th International Database Engineering and Applications Symposium (IDEAS 2004)*, pages 291–301, Washington, DC, USA, Juli 2004. IEEE Computer Society.

[23] W. H. Tok and S. Bressan. DBNet: A service-oriented database architecture. In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 727–731, Washington, DC, USA, 2006. IEEE Computer Society.