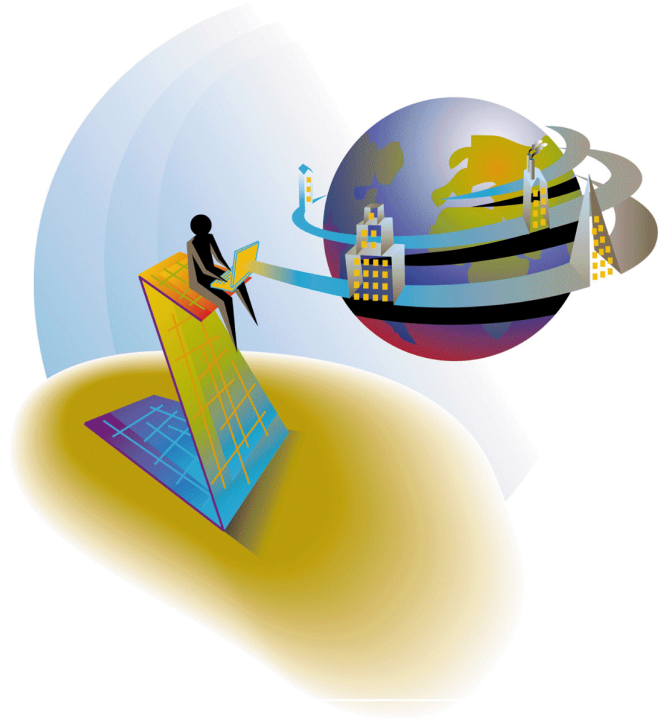


eDeveloper V9.4



Data Management and eDeveloper

MAGIC
www.magicsoftware.com

Enabling Business with Superior Technology

The information in this document is subject to change without prior notice and does not represent a commitment on the part of MSE.

MSE makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose.

The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms and conditions of the license agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement.

No part of this document and/or databases may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or information recording and retrieval systems, for any purpose other than the purchaser's personal use, without the prior express written permission of MSE.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

This product includes software developed by Computing Services at Carnegie Mellon University (<http://www.cmu.edu/computing/>).

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (<http://www.openssl.org/>).

All references made to third party trademarks are for informational purposes only regarding compatibility with the products of Magic Software Enterprises Ltd.

Unless otherwise noted, all names of companies, products, street addresses, and persons contained herein are part of a completely fictitious scenario or scenarios and are designed solely to document the use of Magic.

Magic® is a registered trademark of Magic Software Enterprises Ltd.

PC/TCP® Network Software is a registered trademark of FTP Software Inc.

Microsoft® and FrontPage® are registered trademarks, and Windows™, WindowsNT™ and ActiveX™ are trademarks of Microsoft Corp.

Macromedia® Dreamweaver® is a registered trademark of Macromedia, Inc.

VeriSign® is a registered trademark of VeriSign, Inc.

Clip art images copyright by Presentation Task Force, a registered trademark of New Vision Technologies Inc.

All other product names are trademarks or registered trademarks of their respective holders.

07 06 05 04 5 4 3 2 1

© Copyright • 2004 by Magic Software Enterprises Ltd. All rights reserved.

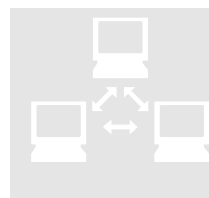
Table of Contents

WHAT IS A TRANSACTION?	4	ADDITIONAL BENEFITS	17
Why We Need Transactions	4	Updating Numeric Data	17
What About Multi-User Environments?	5	Referential Integrity – Parent-Child Situation	19
Sounds Great, or Does It?	7	ERROR HANDLING	22
Isolation Levels	8	Error Handling – the Concept	22
Locked Records	8	PROGRAMMING CONSIDERATIONS	26
DEFERRED TRANSACTIONS	10	Developing for the Web	26
Concepts	10	Logical Serialization	27
Is Everything So Great?	11	Cross-Database Programming	28
Identify Modified Row	11	SUMMARY	29
Are Nested Transactions Possible?	16		

What is a Transaction?

What are transactions? Why do we need to use transactions?

Transactions are an integral part of developing data-bound applications. A key to developing applications in a database environment is the ability to optimally use transactions to ensure data integrity.



The word “transaction” is used very often when discussing SQL applications. A transaction is an integral part of a process, contributing to the composition of the whole application, and may be defined as:

“...An atomic unit of work consisting of a set of data modifications which must be committed in its entirety or aborted altogether”. This means that either the entire process succeeded or the entire process failed. There is no middle ground. Several operations such as UPDATE, DELETE and INSERT may create a single unit. Only if all the operations are successful, will this logical unit be successful.

A transaction process may be an entire business logic process or a smaller unit that is part of a business logic process.

Why We Need Transactions



Every application developer has the same goal of ensuring database integrity. There can be problems if you don't use transactions to ensure database integrity and this can be illustrated by the following scenario:

A user, called Fred, wants to transfer \$50 from his Savings account to his Checking account. In this case, the arithmetic is simple:

$$\text{Savings} = \text{Savings} - 50$$
$$\text{Checking} = \text{Checking} + 50$$

What would happen if there were a failure of some sort between the two operations? In this case, the Savings account would be decreased by \$50, but the Checking account would not be updated. Poor Fred is out \$50 and there is also the question of missing funds. Wait until an audit! We can say that, in this example, data integrity is severely damaged.

If a transaction was used to ensure that the two operations were performed as a single logical unit, then if there was a failure during the execution of this simplistic process, the entire process would have been aborted or rolled back (in the language of SQL) and data integrity is preserved. So the money in the accounts will return to the amounts that were there before initiating the transaction.

Here, I should also add the obvious: In the above scenario in which there seemed to be only two operations, there were actually two more operations:

$$\text{Read Savings amount}$$
$$\text{Read Checking amount}$$

This is without taking into account the actual changes to the database. As we can see, the seemingly simple process is not as simple as meets the eye.

In short, developing with transactions is a necessary part of data-bound applications.

What About Multi-User Environments?



When developing an application for a multi-user situation, in which it is possible that two users will access the same data at the same time, the necessity of developing with transactions is increased.

Let us look at this with the same simple scenario. Let us assume that Fred wants to transfer the last \$50 from his Savings account to his Checking account (sound familiar?). But, his wife Wilma is currently at another branch of the bank and is withdrawing \$50 from the Checking account in order to purchase the weekly groceries.

What can happen? In this case, the scenario is not as simple as it may seem:

Fred	Wilma
Read Savings amount	
Savings = Savings - 50	
Read Checking amount	
Checking = Checking + 50	Read Checking amount

At the same time that Wilma is checking the current balance of the Checking account, in order to withdraw the cash, Fred is putting the money in. In this case, she will probably see that the money has not yet gone through. She will then check the balance of the Savings account and what will she see? It is empty (remember we mentioned that Fred is withdrawing the last \$50). The result will now be angry conversations between the happy couple.

In the world of transactions, this scenario would be slightly different. When the process is executed as a single logical unit, a useful by-product of this is that any updated data is locked until the transaction has completed. In this case, Wilma will still probably see that the money has not yet gone through, but when she then checks the balance of the Savings account, she will see that there still is money in that account. The telephone call to Fred will take on a different tone.

It should be pointed out that the locks are accumulative within the transaction. For example, after updating a certain record (which is locked as a result), the user updates a second record. Both of these records are part of the transaction and will be locked until the transaction is either committed or rolled back.

Supposing Wilma decides that instead of phoning Fred, she is going to transfer the money herself. But by the time this happens, the transaction with Fred has been completed (committed). Now, in this case, when she tries to check the current balance of the Savings account, the amount has already been updated. She can now happily withdraw the money from the Checking account.

What actually happens in this particular case really depends on the DBMS used, but the purpose here is to give a simple overview of transactions.

Sounds Great, or Does It?



Working with transactions is the only way to develop data-bound applications. But, in order to work with them, the developer has to know the pitfalls of using transactions. Let us go back to the problem with Fred and Wilma. Wilma received the wrong information. In the case of the transaction, Wilma received information that the transaction had not been carried out at all. Actually with some RDBMSs, she will actually have to wait until the transaction has been finished.

In our case, it was a simple situation. We can make this situation slightly different. We discussed that one of the by-products of the transaction is that the modified data is locked. Now let us take a scenario when Fred wants to make the transaction. The bank clerk has the Savings account details on the screen and wants to start the transaction now. All of a sudden, the phone rings and the clerk answers it. The data is locked and waiting for the completion of the transaction. Now, dear Wilma has found that Fred has not made the transaction and she wants to execute it herself. (She will take up the matter with her husband later.) But the data is locked; remember the clerk is on the telephone. She will have to wait until Fred's transaction has finished before continuing. We will discuss locking in more detail in the section about isolation levels.

This simplistic situation may not be as simplistic as it seems. At the same moment that this couple are making their transactions, there may be the greengrocer who is currently cashing one of their checks; or the bank manager who is running a report on various accounts, one of them being Fred and Wilma's account.

What is evident here is that despite the fact that database integrity is preserved, the application is time-dependent. That means that while a single user or process is currently updating the data, all others who need to use the same data either have to wait in line or make do with old data. This is perfectly correct behavior in a multi-user situation. But things start getting complicated when one user is holding up the execution, such as the clerk speaking on the phone.

The scenario that we described here is concurrency.

Definition

Concurrency is when at least two users are accessing the same data at the same time.

An application should be developed in such a way that data integrity is preserved and concurrency is maximized. In other words, as many users as possible can access the same data simultaneously. What often happens is that one has an adverse effect on the other. When we try and improve data integrity, often by having a longer transaction, other users will be affected. We previously discussed

before that locks are accumulative within the transaction. If the transaction is long, then all the modified data is locked until the release of the transaction. Other users will have problems when accessing that data. More about this in the next section.

Isolation Levels



At this stage, we should talk a little bit about the database's *isolation level*. We have indirectly touched on the subject throughout this chapter and now it is time for a definition of sorts.

The isolation level of a process specifies the degree to which the rows that are read and updated by a process are available to other concurrently executing processes.

Quite often there is a situation where one transaction can change a value, and a second transaction can read the changed value before the change has actually been committed or rolled back. This situation is known as a “Dirty Read”.

Let us look at this in the context of our scenario with our beloved couple, Fred and Wilma. As you remember, Fred is in the middle of his transaction of transferring \$50 from the Savings account to the Checking account. Remember, there are two sides to this transaction: the decreasing of the Savings account by \$50 and the increasing of the Checking account by \$50.

Wilma has now checked the Checking account and the \$50 is not there (the clerk is on the phone) and when she checks the Savings account, the amount is \$0. Wilma checked the Savings account during the course of Fred's transaction before he completed the transaction.

There may be a problem here if Fred aborts his transaction. All changes will be reverted to their original status. Unfortunately, the second user will not know that Fred has aborted this transaction. In order to know that Fred has aborted the transaction, the data has to be reread. Dirty Reads obviously increase concurrency but reduce consistency.

Locked Records

During the course of this chapter we discussed locked records or rows as they are referred to in the SQL world. When a record is locked, what data another user will actually see depends on the isolation level defined and on the RDBMS you are using. There are three situations that may occur:

- The data is locked. The second user will either receive a message or will have to wait for the data to be released. In eDeveloper, the user will receive a message: "Waiting for locked record".
- The data that will be displayed is the modified data (the "dirty read" situation).
- The data that will be displayed is the data that was there before the lock was initiated, in other words the data as it was before the transaction was initiated (what is known as the "Before Image").

This section touched briefly on isolation levels. There are actually 4 isolation levels in which you can define how the processes interact with one another. For more information about isolation levels please refer to the reference guide of your RDBMS.

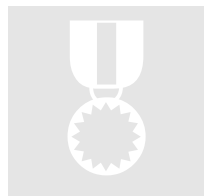
Deferred Transactions

What is a deferred transaction? How does it differ from other transactions?

A deferred transaction is a transaction in every sense of the word. The developer does not have to learn new skills in order to use a deferred transaction. But despite all this, the developer should know when and how to use this new mechanism, its advantages and problems that may arise and how to overcome them. It is important to point out that a deferred transaction is an internal eDeveloper concept.

How to go about this will be dealt with briefly in this chapter and the following chapters. Stay tuned.

Concepts



As we saw in the previous chapter in the case of our happy couple, Fred and Wilma, it is important to make the transaction as short as possible. By making the transaction short, the lock resulting from a transaction is also short and therefore the other users will wait less for the data. This obviously increases concurrency.

The idea behind a deferred transaction is that all the changes to the database are kept in the cache memory (client memory or server memory). What do we mean by changes to the database? These are the Data Manipulation statements such as INSERT, UPDATE and DELETE.

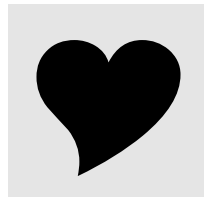
When this transaction has completed, all the database changes will be written to the database in one short transaction, which we will refer to as the physical transaction. Now, the actual transaction is short. By using this type of transaction, the concurrency of the application has been greatly increased.

Let us take our happy couple's predicament and see how this will help them out. Dear Fred is standing in front of the clerk, waiting patiently for the clerk to finish the conversation. The clerk is using data that has been implemented as a deferred transaction. Now, Wilma comes along and wants to transfer the money. Lo and

behold, there is no problem. She performs the transaction, as a deferred transaction as well. The database is updated; there is now money in the Checking account and she can withdraw the money that she wanted to withdraw in the first place. She can take up matters with Fred later.

What we can see here is a major advantage of using a deferred transaction; the actual physical transaction is much shorter and, by default, the data locking is also shorter. The clerk sitting on the phone no longer influences Wilma.

Is Everything So Great?



If this new transaction mechanism solves the problem of concurrency, then we should all use and keep our transactions to a minimum. Wait a minute; there is a problem here. Fred is standing in front of the clerk who is on the telephone. At some stage, the clerk is going to get off the telephone and complete the transaction he started. Now, he completes the transaction but Wilma has already done so! In other words, Wilma has already updated the balance. The amounts that he started off with are not the current amounts! This common problem is sometimes referred to as a “lost update”.

What should be done here? Do we want the bank clerk to receive a message, saying that the data has already been updated? Or do we want the clerk’s transaction to complete?

Identify Modified Row



How can we overcome the problem that we just raised, the issue of the “lost update”? Let us look at our problem again. While Fred is trying to draw money from the Savings account, Wilma has already updated the amount. The way that an update normally works is that before an update is performed, the original data is checked. What does this mean? Let us assume that the couple had \$100 in the Savings account, of which they wanted to transfer only \$50. After the transaction, they would have \$50 in the balance. But let us confuse the issue slightly and say that Wilma realized that she only needed to withdraw \$40.

Let us look at the underlying SQL commands of Fred and Wilma:

```
Wilma: UPDATE savings SET balance = 60 WHERE id=300  
Fred:  UPDATE savings SET balance = 50 WHERE id=300
```

If Wilma performs her transaction before Fred completes his; but Fred does complete the transaction; then the balance of the Savings account is \$50. Wilma's transaction is lost – “lost update”.

How can this be circumvented and controlled? **Identify Modified Row** to the rescue!

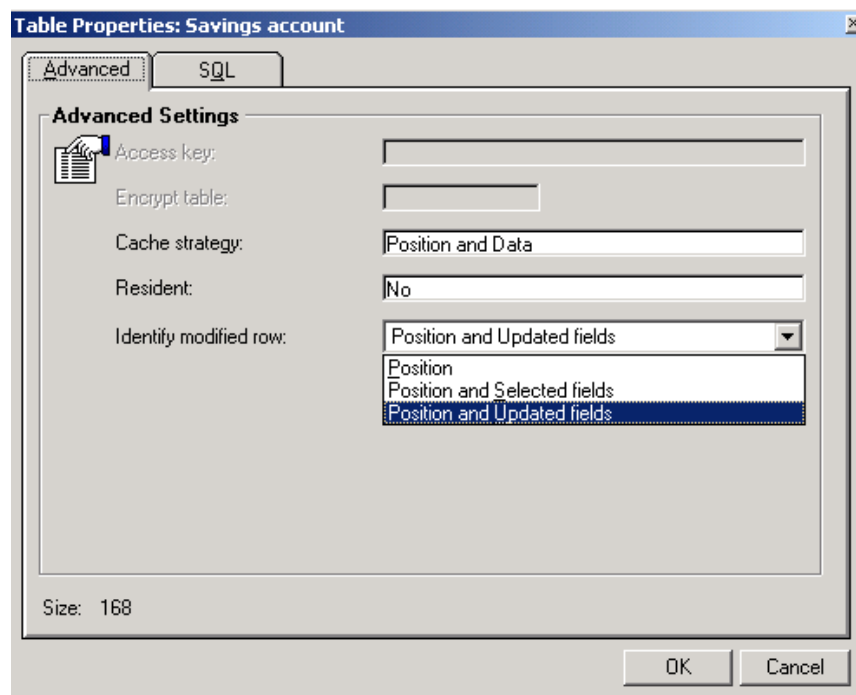
Let us try and understand why we reached this situation. The reason is that in the WHERE clause, we used WHERE id=300. This is the record's position or unique identifier. The position depends on the RDBMS. Some RDBMSs such as Oracle work with a rowid, others such as MSSQL will use the shortest unique index such as the account number. But, by using id=300 in the WHERE clause, there is not sufficient information to know that the record has been updated. More information is required.

Identify Modified Row

This table property appears both on the table in the Table repository and in the DB Table repository of a task. It has the following options:

- Position
- Position and selected fields
- Position and updated fields

The default for this property is “Position and updated fields”.

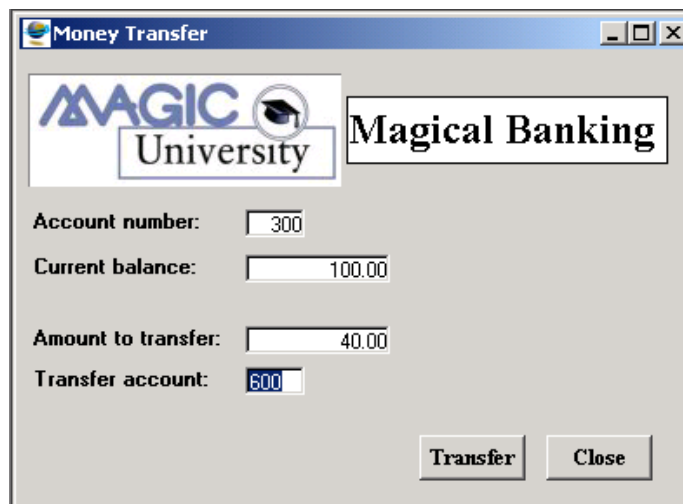


Let us look at how this property can help us or hinder us. In the example, we have already seen the use of the “Position” option of this property. The effect that this may have on our scenario is the “lost update” problem. This option enables the user to update the data even though someone else has already updated it without regards to the other update. You may be asking yourself in what situation you may want to do this. Once again, let us return to our happy couple Fred and Wilma. Suppose that they had been checking their bank account details and realized that the bank had the wrong zip code for their home address. Oops! They decide that while they are transferring the money from account to account, they will also ask the clerk to change the zip code. Now in our scenario in which both Fred and Wilma are performing the same transactions, it is not really important who enters the updated zip code, since the data remains the same.

Let us look at the second option of the *Identify Modified Row*, Position and Selected Fields. This option will add all the fields that were selected in the program to the resulting WHERE clause. Now what will that mean for Fred and Wilma? Well, let us assume that the program that is being used to display the balance and to enable the transfer from the Savings account only displays the following information to the clerk:

- Account Number
- Account Balance

It also shows a non-database field for the amount to transfer and another for the receiving account (see the figure below).



Now, the resulting WHERE clause would have both the Account Number and the Account Balance added to it.

Let us look at the underlying SQL commands of Fred and Wilma (remember that they had \$100 in the Savings account before the transfer):

Wilma: UPDATE savings SET balance = 60 WHERE id=300 and balance = 100

Fred: UPDATE savings SET balance = 50 WHERE id=300 and balance = 100

What would happen here? Well, Fred's transaction would begin but the clerk's phone call is holding it up. Whilst he is on the phone, Wilma makes her transaction and decreases the balance to \$60. Now when Fred tries to complete the transaction, the balance is no longer \$100 and so it fails. Data integrity is upheld. This is a very comprehensive way of checking for an update. In the example that we used, it seems to have been the ideal solution; but is it? Well let us take the same program from before and add the account details. These details are used to show who the account belongs to. So now we have the following information being displayed:

- Account Number
- Name
- Last Name
- Address
- City
- Zip Code
- Telephone Number
- Account Balance

Money Transfer - full details

MAGIC University **Magical Banking**

Account number:

Account details

Name:	<input type="text" value="Fred and Wilma"/>	<input type="text" value="Smith"/>
Address	<input type="text" value="10 Upping Avenue"/>	
City	<input type="text" value="Londera"/>	<input type="text" value="65232"/>
Telephone	<input type="text" value="555-2055"/>	

Current balance:

Amount to transfer:

Transfer account:

Now let us look at Wilma's SQL command again:

```
UPDATE savings SET balance = 60 WHERE id=300 AND first_name = "Fred and Wilma" AND surname = "Smith" AND address = "10 Upping Avenue" AND city = "Londera" AND zip = 65232 AND telnum = "555-2055" AND balance = 100
```

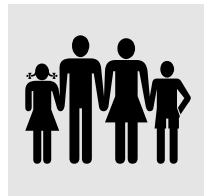
Quite a mouthful! If another user updates any of the above data, the transaction will fail. This is very useful but is it useful for all cases? Let us look at our scenario again. Remember that our happy couple wanted to change their zip code. It appears as 65232, but it should be 65233! Now, while Wilma was doing the transfer, Fred was updating the zip code to 65233. What will happen if we use the Position and Selected Fields option? The transaction would fail because "zip = 65232" is no longer true.

What do we do here? Welcome to the third option, Position and Updated Fields. This option will only add the updated fields to the WHERE clause. So in our case, let us look at the underlying SQL command:

```
UPDATE savings SET balance = 60 WHERE id=300 and balance = 100
```

It is the same as the program with only two fields in it. The advantage here is that Fred can update the zip code without interfering with Wilma's transaction. Where would this fail? This would fail if both members of the family tried to update the same field at the same time.

Are Nested Transactions Possible?



Let us think for a moment what a transaction within a transaction is. In a nutshell, while there is an open transaction, the developer would like to have an inner transaction that will commit its commands to the database before the other transaction has finished. This transaction is independent of the other transaction. Let us look once again at our scenario. Dear

Wilma wants to transfer money from the Savings account, but while she is doing that she also wants to update the zip code. This can actually be performed as a separate, independent transaction. In eDeveloper, this mechanism is called a Nested Deferred transaction. In the above scenario, we would have a separate program or task in which the transaction mode is defined as Nested Deferred, which will update the details of the account.

Note

When an inner transaction is committed, it is committed! If the external transaction is now rolled back, the inner transaction will not be rolled back. There is no connection between these transactions.

In this chapter, we have seen the concept of deferred transactions and seen how this can help us solve other problems that can arise.

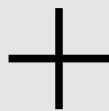
Additional Benefits

What additional advantages do we get from deferred transactions?

We learned about deferred transactions and how to use them and how using them improves concurrency in deployment in a multi-user environment.

In addition to the advantages that we have already mentioned, there are other benefits that are available to the Magician when working with deferred transactions, which will assist in solving other real-life problems.

Updating Numeric Data



Let us look at our scenario to show the problem and the solution. Remember our scenario in which Wilma wanted to update the balance? We updated the balance with a specific value i.e. 60 (\$100 minus the \$40 withdrawn by Wilma). This is referred to as an absolute value. The balance field was updated by a constant value.

Remember the resulting SQL command:

```
UPDATE savings SET balance = 60 WHERE id=300
```

The balance was updated with 60. Now let us look at this differently. We seem to have forgotten about the other part of this transaction – the Checking account. Let us assume that there was also \$100 in this account before Wilma started the transfer. So now let us look at both sides of the transfer:

```
UPDATE savings SET balance = 60 WHERE id=300  
UPDATE checking SET balance = 140 WHERE id=600
```

The balance of the Checking account has been increased with a constant value of \$40. This is the amount that was transferred from the Savings account. In our little story we seem to have neglected the greengrocer who wants to cash Wilma's check for \$30. So here the SQL command would be:

```
UPDATE checking SET balance = 70 WHERE id=600
```

But wait a minute, both of these transactions running at the same time will cause a problem – “Lost Update”. What should we do in order to maximize concurrency?

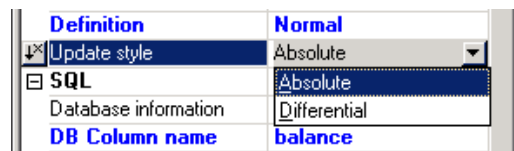
Let us welcome a new property: **Update Style**

Numeric Update Style

This property is only valid for a Numeric field and is defined in the table's column or in the Select Properties dialog box of the task. It has the following options:

- Absolute
- Differential

The default of this property is Absolute.



How can this property assist our dilemma? When the Update Style property is defined as *Differential*, then instead of an absolute value being set, i.e. a constant value, the difference is sent instead. This may be better understood with our example. Let us take Wilma's update and the greengrocer's update and look at them again.

```
Wilma: UPDATE checking SET balance = 140 WHERE id=600
```

```
Grocer: UPDATE checking SET balance = 70 WHERE id=600
```

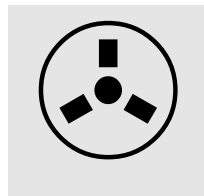
Now let us send these transactions as a Differential update style:

```
Wilma: UPDATE checking SET balance = balance + 40 WHERE id=600
```

```
Grocer: UPDATE checking SET balance = balance - 30 WHERE id=600
```

So what is the end result here? It is not important in this case which operation is performed first, the end result will be that there is \$110 in the Checking account! This is the correct amount that should be in the database.

Referential Integrity – Parent-Child Situation

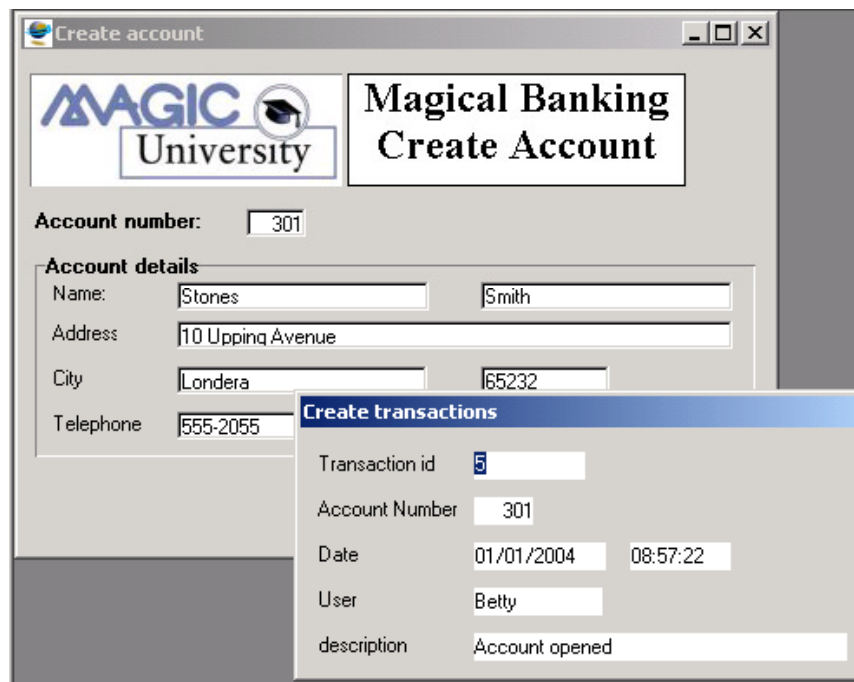


In a typical RDBMS database, some tables are defined in which they are related to one another, such as a Header Order and Header Line relationship. Database rules are created that link these tables together. Deferred transactions are advantageous in a certain situation. In order to explain the problem and the solution, we will use our banking application that we have come to love so much. We have our Savings account, which holds the current balance. But what often happens is that we would need to have a history file which will show all transactions that were made. You wouldn't like to phone your bank and ask how much money you have (or don't have) and be told \$100 without any method of finding out how they came by this figure.

This history file is a table that is linked to the account table. What would be the meaning of a history file without the account file? If we had the history of account number 921 in the history file, but no account 921 in the accounts table, the history would be meaningless and unreachable.

This will be a parent-child table relationship. When defining this kind of relationship in the RDBMS using referential integrity, the RDBMS assists us and does not allow a situation to exist in which there is an entry for an account in the history table but no corresponding entry in the account table. By the same token, it would not allow us to delete a parent entity (account) where there is still data linked to it.

Let us look at this in a typical scenario, the same scenario that we have come to know and love. But now, we will slightly change the scenario and say that Fred and Wilma want to create a Savings account for their darling daughter. The bank database requires that when an account is opened, an entry is written to the history file with a description of "Account opened".



What will happen here? The program creating the account will call the program that creates the transaction. Let us have a look at the resulting SQL commands (ignoring the problem of date and time conversions):

```
INSERT INTO history (transid, id, transdate, transtime, clerk, desc) VALUES (5, 301, '2004-01-01', '085722', 'Betty', 'Account opened')
```

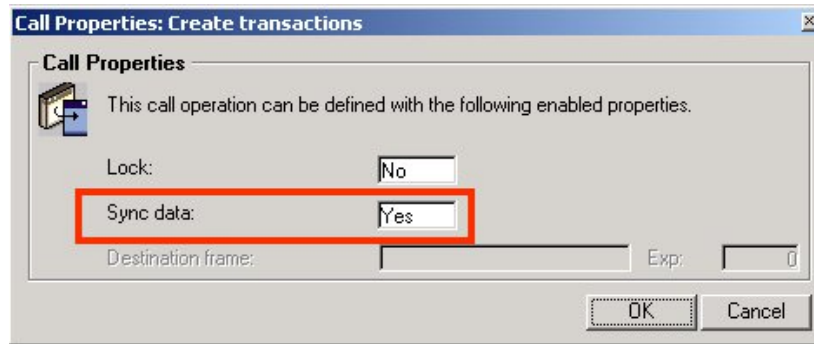
```
INSERT INTO savings (id, first_name, surname, address, city, zip, telnum) VALUES (301, 'Stones', 'Smith', '10 Upping Avenue', 'Londera', '65232', '555-2055')
```

Here, we can now see the problem. The entry for the **history** table is being written to the database before the entry in the **savings** table. This will cause a conflict with the database referential constraint rules. In this case, the transaction will fail.

So what is the solution to this problem? The solution is obvious; the **history** record has to be written to the database after the **savings** record has already been written (referred to as flushed) to the database. The question remains as to how to do this in an application. Well, we can utilize the fact that all the data manipulation commands are held in the transaction cache before they are committed to the database itself. We now introduce the use of a new property: **Sync Data**.

Sync Data

This property may be found in the properties of a Call Task, Program or Public operation. It is a logical value that may have an expression. When this property evaluates to True, the *sync data* flushes the parent program's record to the database before flushing the child's records.



How will this help us in our scenario? By changing the Sync Data value to True, the record of the parent program, **Create Account**, in which we want to flush the record in order to add a new entry to the **accounts** table, is actually flushed to the physical database before the record of the second program, **Create Transactions**.

This functionality is made possible because eDeveloper has all the manipulation information stored in cache, waiting for the final OK.

Error Handling

What do we mean by an error? My programs do not have errors!

There is a well known proverb that says “To err is human”. But we all know that to totally foul things up takes a computer. But here we are not referring to programmatic errors per se. Here we are referring to database errors. These errors are generally as a result of an attempt to perform something that goes against the database constraints. What do we mean by this? Well remember we spoke about referential integrity? We had the accounts table that had a transaction history table. If we were to try and delete a record from the accounts table and there were still records for that account in the history table, the RDBMS would stop us: “Hey, you can’t do that”. We would have to first delete the records from the history table and only then would we be able to delete the record from the accounts table.

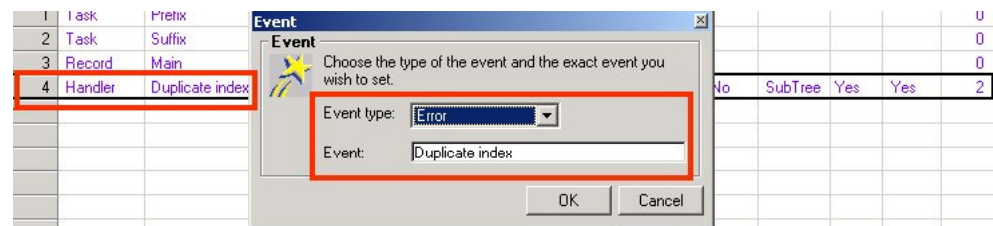
Error Handling – the Concept



eDeveloper has an error handling mechanism designed to take the headache of handling errors away from the developer. There are a few levels at which database errors are handled by eDeveloper. The first and foremost level is handled by the engine itself or the eDeveloper internal gateway for that DBMS. Let us take our example of Wilma trying to create a new Savings account for her daughter. The application enables the client to select which account number they would like to have from a list of available numbers. So the bank clerk displays this lovely list to Wilma and she sees that account number 301 is available. Wow, what luck! Her account is 300, easy enough to remember. So she chooses this number and the clerk enters the details. But, since they are working in a multi-user environment, someone else was quicker and chose that number. When the transaction is committed, the bank clerk will get an error message. The error message that will be seen is “**Duplicate Index, Table: accounts**”. This is an error generated by eDeveloper. The actual error message from the DBMS depends on the DBMS itself. For instance, an error message from MSSQL 2000 may be something like: “Cannot insert duplicate key row in object ‘history’ with unique index ‘trans’”. For Oracle, it will be totally different. Within eDeveloper, the message will be the same for all databases.

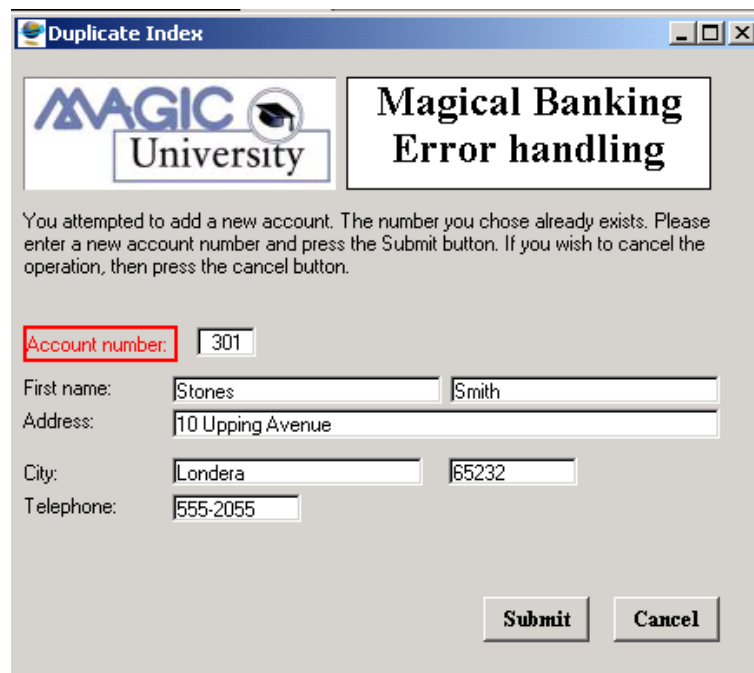
Looking at this, how does it help us? In most cases, this is more than enough. But suppose for a moment that you had a long transaction continuing through a few tasks and when you committed this transaction, you received the error. What do you do now? It would be nice to be able to provide your own screen that will enable the user to fix the error, or in our case to provide a new account number. As a result, eDeveloper enables the developers to provide their own developer-defined handling of this error. As George Washington once said “**To err is natural; to rectify error is glory**”.

The developer-defined error handling mechanism works in the same way as eDeveloper’s internal event handling mechanism. You define a handler of type Error.



So how does this work? The engine encounters a DBMS error; it raises the Error event, for example our “Duplicate Index” error event. The engine now looks for a handler to handle this error. It starts searching in the current task, along the runtime task tree until it reaches the Main Program. If it does not find a handler there, it will revert to the default error handling mechanism that gives the message. In order to understand more about the event handling mechanism, you should read the concept paper called “Event-Driven Architecture” which explains this mechanism in depth.

Let us use the Duplicate Index situation as an example of what can be done. We can create an error handler for Duplicate Index. Within that handler, we can call a program of our own that displays the error and enables the user, in this case the clerk, to fix the error. We can add as much information as we want, to ensure that the clerk will know what to do.



The Cancel button, in the above example, may simply perform a Rollback of all the work that resulted in this error.

eDeveloper's Built-in Behavior

The question remains as to what will happen after the engine encounters a handler and executes it. After the engine has executed all the error handlers, or its own internal default handler, it will then perform what is known as an “**engine directive**” for that specific error. The directive works hand-in-hand with the Error Behavior Strategy defined in the program itself. This may be either Abort or Recover. This strategy defines how the task will behave when it encounters a database error.

Each error will have a different behavior under the Error Behavior Strategy. As an example, let us look at our scenario in which we received the Duplicate Index message. If the Error Behavior Strategy is defined as Recover, the engine directive for this error will be User Retry; meaning that the user can re-enter the data. On the other hand, if the Error Behavior Strategy was defined as Abort, the task will be aborted. As a different example, we can look at the Maximum Connections Exceeded error. If the Error Behavior Strategy is defined as Recover, then the eDeveloper engine directive will be Auto Retry without input from the user.

The developer can override these by re-defining the requested directive in the handler and giving a new directive. Let us take the Duplicate Index example again. The developer may define that when a duplicate index is encountered, the error is serious enough that the task must be aborted and all the work rolled back. The developer will then set the engine directive to *Abort task* in the handler.

Note

Some of the engine directive options, such as "Ignore" are not valid for some of the errors. Some of the handlers, such as "Duplicate Index", are only valid for deferred transactions. Please refer to the *eDeveloper Reference Guide* for more information.

Important

The engine directive may be set for each error handler. However, the resulting directive that will be executed is defined at the level of the first handler that is executed.

Programming Considerations

What should be taken into consideration when programming with databases?

We now know how to work with transactions, especially deferred transactions and we also know how to handle errors that may arise, but what else should be taken into consideration? This is what will be dealt with in this chapter.

Developing for the Web



As mentioned in earlier chapters, in order to have concurrency, the lock and the transaction need to be as short as possible. A Web application can serve thousands of users at the same time all over the globe, and locking a resource can make the application unavailable to some users. The client in this kind of an environment is disconnected from the server and, as a result, from the database server. When the user will actually commit the transaction is unknown. The user may also exit the browser by unconventional means, such as clicking the browser's Close button or by moving to a different Web page. In this case, the transaction would be kept open until the eDeveloper context times out.

As a result, the recommended transaction mode of a Browser task is a deferred transaction.

Transaction Considerations

You should carefully consider when the transaction should be opened. There are no hard and fast rules, but we will try and provide some basic guidelines:

Menu Tasks

Browser tasks that are opened from a menu task are usually expected to open an independent transaction, which is why the Transaction mode property of the menu task should be set to **None**.

Handling Each Record Separately

Set the Transaction begin task property to **Before Record Prefix** if you want to have every record committed immediately when the record is exited or, alternatively, to roll back each record modification separately.

Important

Take into consideration that in a record level transaction, the browser client connects to the remote server for every record that is modified.

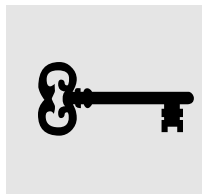
Handling All Task Records as a Single Unit

Set the Transaction begin task property to **Before Task Prefix** if you wish to have all the modified records committed only when the task is closed or, alternatively, to roll back all the modifications of the task records as a single unit.

Note

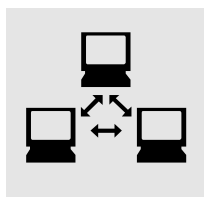
Setting the transaction as task level will reduce the number of times that the browser client will contact the server.

Logical Serialization



Quite often there are processes which are important enough to the integrity of the system, that only one user should be allowed to access it at any given time. Any other user who tries to access this process will have to wait their turn. Let us take our happy couple as an example. They were discussing the fact that perhaps they needed a line of credit for their Checking account. So while Wilma was at the bank, she went to see the bank manager and discussed this with him. The bank manager agreed to provide \$1000 in credit and began to update the system. At the same time, Fred is at another branch (or even at the same branch) and is discussing matters with the Credit Manager who agrees to provide only \$750. If both of these transactions were allowed to go through, the actual line of credit would be \$1750. In order to overcome this problem, you can use the eDeveloper internal function *Lock* for this program. By using this function in the Task Prefix of the Credit Line program (and *UnLock* in the Task Suffix), then once the Bank Manager loads the program, the Credit Manager would not be able to load it until the Bank Manager finishes.

Cross-Database Programming



Sometimes it is necessary to create an application in which the customer determines on which RDBMS it will run. For example, the software house may have developed their banking application on MSSQL and sold it to Stones Savings Bank who has an Oracle database. The same application may have also been sold to Stones Loans Bank, which runs on a DB2 database.

There are some guidelines that may make this transition as smooth as possible.

Default Storage

The Default Storage property is defined per column in the table. It takes into account the inherent defaults that eDeveloper has for each data type. This is mainly relevant for tables that are created via the application itself and not via the DBMS. In this case, it is the eDeveloper DBMS gateway that defines the storage that will be created in the database.

Note

The default storage is defined in the gateway and cannot be changed by the developer.

As an example, let us look at a numeric field. In some DBMSs this would be defined as INTEGER and in others it may be NUMBER or PACKED DECIMAL. The eDeveloper gateway for that specific DBMS will handle this difference internally.

Magic SQL Clause

Sometimes the developer may need to add SQL statements to the WHERE clause sent by eDeveloper to the DBMS. These are entered as a DB SQL range and sent as entered to the DBMS. When writing cross-database applications, you will often find differences in some of the functions. For example, the function to fetch a substring of a certain string is *Substr* in DB2 and Oracle, and *Substring* in MSSQL. This obviously causes a problem.

You can use the Magic SQL range, so that eDeveloper performs a syntax check, and regular eDeveloper functions can also be used. As an example we can take the above example of a subset of a string. Instead of battling with the various nuances of different DBMSs, we simply use the eDeveloper internal function: *MID*.

Note

The internal functions that are available in the Magic SQL range are only a part of the eDeveloper functions. Please refer to the *eDeveloper Reference Guide* to find out what functions are available.

Summary

A few words before we end.

We hope this document provided you with sufficient information about eDeveloper database programming. We hope that after reading this document, you are more familiar with the usage of deferred transactions and error handling.

We are confident that using eDeveloper's inherent capabilities, the developer can create more complete and robust applications that work well in a concurrent deployment environment and also one in which the nuances of each RDBMS are minimized.