Transactions and concurrency

Introduction

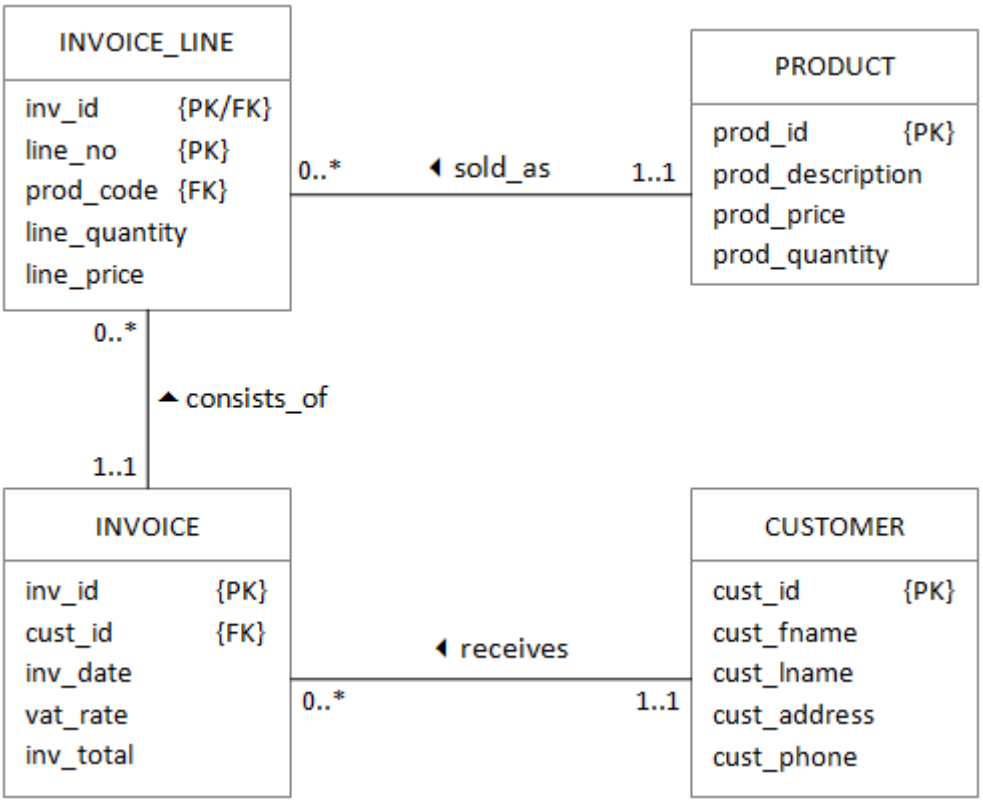
One of the principal purposes of databases is to support many simultaneous users. However, it is perfectly possible that different users may need to update the same data at the same time. The DBMS therefore needs ways to avoid conflicts in such situations. The problem is further complicated because it is sometimes necessary to update several different tables in the database as part of the same task.

This week we are looking at the main mechanisms that the DBMS users to maintain the integrity of the database contents in a multi-user environment. As always, there is much more detail on the subject than we can cover in this module. You can use the embedded links to find out more.

Transactions

The database operations you have seen so far involve only a single table. Even when you have been inserting related rows in separate tables, each operation is essentially independent. Frequently, though, in real database systems a single task may involve updating several tables, and all updates must be successful in order for the overall task to succeed.

As an example, consider a database that supports an e-commerce Web site. The company needs to maintain a stock of products, and when a sale is made, the quantity of product available needs to be reduced. A customer may buy more than one product during a single visit to the site, and a confirmation receipt is sent by email when the sale is complete. The ER diagram below shows the tables needed to represent this information.



To explain the diagram a little: one **CUSTOMER** can make many purchases, and each purchase is represented by a record in the **INVOICE** table. Each **INVOICE** may have several **LINES**, and each **LINE** represents a **PRODUCT** that the **CUSTOMER** is buying. The **LINE** indicates how many units of the **PRODUCT** the customer is buying, and maintains a record of the unit price. This is needed because the price of the **PRODUCT** may change over time, but the **LINE** price must remain the same. VAT is calculated for the whole **INVOICE**, and therefore the total cost of the purchase including tax is held on the**INVOICE** record. The **PRODUCT** record shows how many units are available in the PROD_QUANTITY column, and this value is reduced whenever some units are sold.

When a purchase is made, several updates must be made:

- An **INVOICE** record must be inserted with a new INV_ID
- A **LINE** record must be inserted for each different **PRODUCT** using the same INV_ID
- Each **PRODUCT** record must be updated to reduce the PROD_QUANTITY
- Once all the **LINES** have been inserted, the **INVOICE** record must be updated with the total value including tax

If any one of these steps fails for some reason, then the overall task of recording the sale is incomplete and can cause problems. For example, if the PROD_QUANTITY value is not updated on the **PRODUCT** record, it will look as if there is more of a product available than there actually is. If the final step is missed, the **CUSTOMER** will not be charged the VAT.

These operations must either all succeed of all fail to leave the database in a consistent state. They therefore represent a *logical unit of work* which includes a set of database operations, and that is the definition of a database *transaction*.

Another way of saying that a transaction cannot be split into smaller parts would be to say it is *atomic*.

Transaction control

The operations that make up a transaction do not all happen simultaneously. While they are being carried out therefore, a second user can query the data. If that happens, the second user will get an incorrect impression of the state of the database. For example if User2's query happens after all records have been inserted but before the **INVOICE** record has been updated with the tax, the **CUSTOMER**'s bill will appear smaller than it actually is.

It is therefore important that before the entire transaction is complete, none of the updates are visible to other database users. To accomplish this, the DBMS effective creates a copy of the database objects that are affected by the transaction. The changes are made in the copy, and only when they are all complete are the changes made permanent in the database. Performing the operations in a protected copy also means that if any one operation should fail, all update can be abandoned.

This behaviour can be demonstrated by using two independent connections to the same database to simulate two different users. This can be done in Oracle by logging into SQL Developer as the HR user, and logging in as HR through SQL*Plus at the same time.

With a small amount of investigation, you will find that there are 25 records in the **COUNTRIES** table, but it does not contain a record for Spain. In SQL*Plus, type the following:

```
INSERT INTO countries (country_id, country_name, region_id)
              VALUES ('ES', 'Spain', 1);
```

Now the two connections show different states of the database. Try the following query in both SQL*Plus and SQL Developer:

```
SELECT COUNT(*) FROM countries;
```

To make the change permanent, you need to tell the database that the transaction is complete by issuing the command:

```
COMMIT;
```

Try it now, and then check that both connections now show the same number of countries.

If for some reason you do not want the transaction to go ahead after all, you can undo the changes by issuing the command:

```
ROLLBACK;
```

This exercise demonstrates the property of *isolation* by which other users are protected from a transaction until it is complete.

Consistency is another property of transactions which ensures that a transaction returns a database to a consistent state once it is complete. A consistent state in this context is one in which no constraints are violated. By default, Oracle validates each constraint immediately, even during a transaction. Using the appropriate commands, however, a database programmer can defer constraint checking until the end of the transaction. In this situation records can be inserted even if foreign key values do not match primary keys elsewhere. As long as all the data is in place by the end of the transaction, everything completes as normal. On the other hand, if the COMMIT command is issue to terminate the transaction and some constraints are still violated, the whole transaction is rolled back.

The transaction log

All of the details of a transaction - the tables involved and the data - are save into a *transaction log*. Essentially this operates just like any other table. The importance of the transaction log is that it can be used to return the database to a previous consistent state (after a system failure, for example). It is actually integral in creating the illusion of the "private copy" that we have just seen. The elements of the transaction are queued in the transaction log and are then executed as a set when the commit command is issued. The operations can be just as easily rolled back since they have not yet been made permanent in the database.

The transaction log contains all of the database operations that have been performed since the last database backup. This means that it can also be used to bring the database up to date in case of some kind of serious system failure. This use of the transaction log will be covered in more detail in week 11.

All the main database platform manufacturers manage transaction log in slightly different ways, and for that reason they often use their own terminology. Oracle for example uses the term *redo log* while Microsoft SQL Server uses the standard *transaction log*.

Concurrency

Concurrency is the term used to mean the simultaneous processing of independent operations in a multi-user environment.

We have already considered the problems that could occur if a second user has access to the changes made by a transaction before the entire transaction is complete. This is one type of concurrency problem known as an *inconsistent retrieval*.

Things can get even worse if instead of just querying the data the second user tries to update the same records. In this situation there are two simultaneous transactions competing for the same resources. The next two sections outline specific problems that can occur using the example of the e-commerce Web site example.

Lost updates

Imagine two different customers buying quantities of the same product at the same time. Customer1 orders 10 units of the product and Customer2 orders 50, and the PRODUCT record shows that there are currently 100 units available. At the end of each transaction, the remaining quantity of the product needs to be updated. To see how problems can arise, we need to think about the individual step in the update of the product quantity as shown below:

1. Read current product quantity
2. Calculate new product quantity
3. Store new value

If both transactions are running concurrently, it would be entirely possible for the steps of the two transactions to be interleaved as shown below:

Step	Transaction	Operation	Stored value
1	1	Read quantity	100
2	2	Read quantity	100
3	1	Calculate: new value = 100 - 10	100
4	2	Calculate: new value = 100 - 50	100
5	1	Store new value	90
6	2	Store new value	50

Notice that after both transactions are complete the value stored in the table is 50 when in fact it should be 40 (50 + 10). The update at step 5 has been lost because both transactions used the original value of 100 as the basis of their calculation.

Uncommitted data

If a transaction T2 has access to the data stored by another transaction T1, a further problem can occur if T1 is rolled back. Again the problem arises because of the interleaving of the low-level steps of the updates associated with the two transactions. The table below illustrate the situation:

Step	Transaction	Operation	Stored value
1	1	Read quantity	100
2	1	Calculate: new value = 100 - 10	100
3	1	Store new value	90
4	2	Read quantity (uncommitted)	90
5	2	Calculate: new value = 90 - 50	90
6	1	ROLLBACK	100
7	2	Store new value	40

Because the calculation in T2 is based on uncommitted data, the final value stored in the table is 40 when in fact it should be 50.

Locking

To prevent the problems identified in the previous section, the DBMS isolates each transaction. That is to say that it ensures that the interleaving of low level steps cannot happen and that one transaction must be complete before another one can make use of the same data. This is achieved by *locking* the data that is in used and releasing the locks again once the transaction has been committed or rolled back.

Locks can operate at different levels. For example, when a user starts a transaction, the entire database could be locked until the transaction is complete. However, in a busy environment this would cause a significant decrease in performance while transactions are held in a queue. A less drastic lock would be one which locks only the affected table in the database leaving the others available for other transactions. In real systems such as out e-commerce example, this would still cause performance issues because the transactions would be concentrated on a small number of tables. The effect would therefore be similar to locking the entire database.

A much more realistic approach is to lock individual rows within tables. *Row-level locking* allows multiple users to perform transactions on the same tables concurrently, and only a small number would actually conflict with each other. The small proportion of conflicts would have a negligible effect on performance.

Of course, the logical extension of row-level locking would be *field-level locking* where different users can update different fields in the same record concurrently. The benefits of this would be very small, however, and the management overhead would be very large. Field-level locking is therefore rarely used.

A special DBMS process called the *scheduler* manages the locking process thereby keeping transactions isolated from each other. We can see how locking resolves the problems illustrated earlier by revisiting the same examples.

Lost updates

Step	Transaction	Operation	Stored value
1	1	Obtain locks	100
2	1	Read quantity	100
3	2	Wait	100
4	1	Calculate: new value = 100 - 10	100
5	2	Wait	100
6	1	Store new value	90
7	1	COMMIT (release locks)	90
8	2	Obtain locks	
9	2	Read quantity	90
10	2	Calculate: new value = 90 - 50	90
11	2	Store new value	40
12	2	COMMIT (release locks)	40

Because the row in the PRODUCT table is locked by transaction 1, transaction 2 is forced to wait until transaction 1 is complete. This ensures that the correct value is stored in the database at the end of both transactions.

Uncommitted data

Step	Transaction	Operation	Stored value
1	1	Obtain locks	100
2	1	Read quantity	100
3	1	Calculate: new value = 100 - 10	100
4	1	Store new value	90
5	2	Wait	90
6	1	ROLLBACK (release locks)	100
7	2	Obtain locks	100
8	2	Read quantity	100
9	2	Calculate: new value = 100 - 50	100
10	2	Store new value	50
11	2	COMMIT (release locks)	50

Again, because transaction 2 is obliged to wait, the final value stored in the database is the correct one.

Deadlocks

Although locking resolves the major concurrency problems illustrated earlier, it can give rise to a problem of its own. Imagine there are two concurrent transactions, T1 and T2, and that they both need to update the same rows in tables A and B. However, T1 updates table A first, and T2 updates table B first. This means that T1 could obtain a lock on table A and T2 obtains a lock on table B, but neither transaction can obtain the second lock because it has to wait for the other. This situation is known as a *deadlock*, and is illustrated in the table below.

Step	Transaction	Operation	DBMS reply	Table A	Table B
1	-	-		Unlocked	Unlocked
2	T1	Lock A	OK	Locked T1	Unlocked
3	T2	Lock B	OK	Locked T1	Locked T2
4	T1	Lock B	Wait	Locked T1	Locked T2
5	T2	Lock A	Wait	Locked T1	Locked T2
6	T1	Lock B	Wait	Locked T1	Locked T2
7	T2	Lock A	Wait	Locked T1	Locked T2

From step 4 onwards, neither transaction can proceed because it is waiting for the other to release its locks. This is a deadlock.

There are several ways that the DBMS can resolve the deadlock problem, either by ensuring that transactions obtain all their locks at once, or by restarting one of the conflicting transactions. Usually the transaction that started first will be allowed to proceed while the later transaction will be forced to restart. This is done transparently to the database user.

Explicit locking

SELECT FOR UPDATE

Complex transaction management is usually only necessary within database applications where the SQL statements are embedded into structured code such as Java or C#. In an application program there may be several processing operations between the retrieval of a value from the database and the modification of the records. For this reason, it is important that there is a way to tell the DBMS that you intend to update certain rows in a table at some point in the future. This is done with an additional clause in the SQL SELECT statement.

An explicit lock can be obtained on all of the rows returned by a query by adding a FOR UPDATE clause at the end of the query. The example below illustrates how to lock the row in the HR.COUNTRIES table that we created earlier:

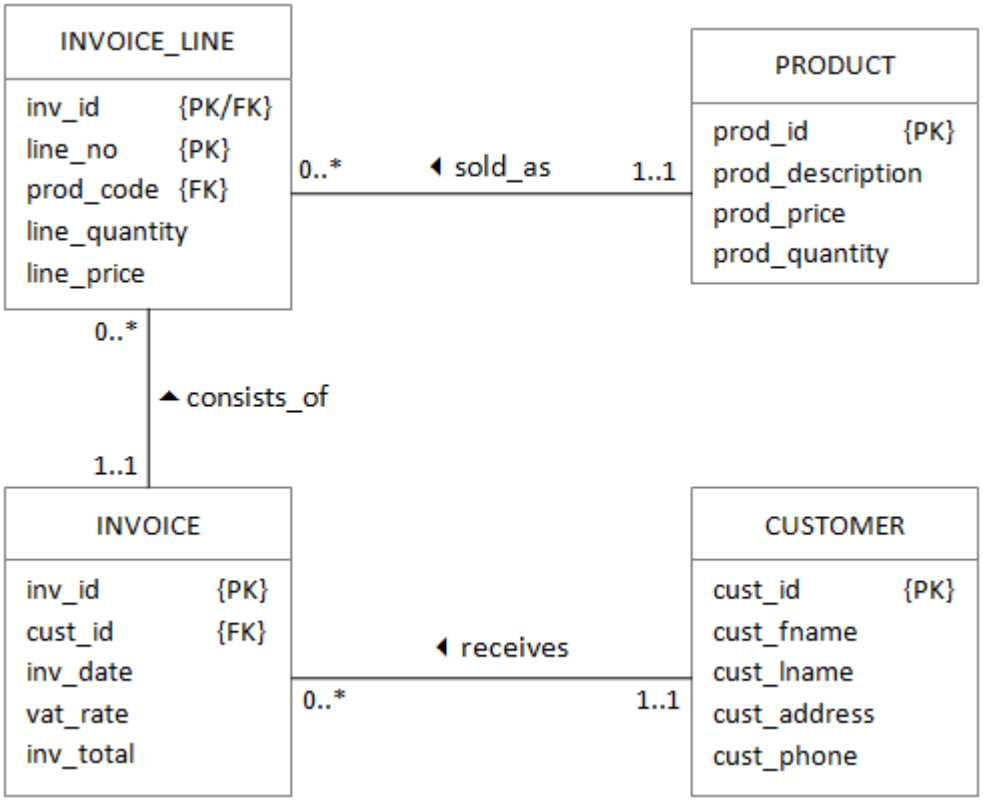
```
SELECT *
FROM   countries
WHERE  country_id = 'ES'
FOR UPDATE;
```

Log into as the HR user in bot SQL*Plus and APEX as before, and run this statement in SQL*Plus. You will see the results of the query as usual, but the difference is that you have now locked that row of the table. You can see the effect of this by running the following statement in the APEX interface:

```
UPDATE countries
SET    country_name = 'Espana'
WHERE  country_id = 'ES';
```

Release the lock in SQL*Plus by issuing either the COMMIT or ROLLBACK statement, then try the update again.

LOCK TABLE



If you are not selecting from the table with a view to updating it later, you can use the LOCK TABLE command. This has exactly the same effect of giving you exclusive control over the locked tables until a COMMIT or ROLLBACK statement is issued. Applied to the e-commerce example, assuming an order with two lines, the full transaction would look like this:

```
LOCK TABLE invoice;
LOCK TABLE product;

INSERT INTO invoice …

INSERT INTO invoice_line …
UPDATE product SET prod_quantity = …

INSERT INTO invoice_line …
UPDATE product SET prod_quantity = …

UPDATE invoice SET inv_total = …

COMMIT;
```