Database Facilities

One of the main benefits from centralising the implementation data model of a DBMS is that a number of critical facilities can be programmed once against this model and thus be available to all databases using the DBMS.

This section provides a discussion of:

- Providing Security in Data Access
- Query Optimisation
- Concurrent Access to Databases
- Recovery From System Crashes
- Distributed Databases

Security Mechanisms I

The database authorisation subsystem controls who may or may not access particular information in the database.

There may be up to six mechanisms used:

1/ User Accounts
   - Just like any complex computer system, the user must be authorised to use a DBMS, by being issued with a login and a password
   - The DBMS will have its own login and password, but insist that the user must be registered by a system administrator before they can use it
     • this information is held in a table which holds all the user information
   - Microsoft Access can have a password system or just allow anyone logged in to use the system

Security Mechanisms II

2/ Privileges

Given an account, the system can then restrict access to particular parts of the data.

Oracle provides two kinds of privileges:
• System Privileges - the right to perform a particular action on all data:
  e.g. to create tablespaces or delete records anywhere
• Object Privileges - the right to perform a particular action on a specific table or view
  e.g. to read data from a particular table

The creator of a relation is its owner & can grant access to others, e.g.:

grant privilege on relation to all/userLogin/role/group

where privilege is one of select, update, delete, insert or all

This allows either everyone or one user to access the relation in the specified way

There is also a revoke command to remove a privilege

Security Mechanisms III

3/ Logical Users

DBMS provide two additional constructs to provide security checks which are not tied to particular users (e.g. mail aliases):
• roles - indicate a logical user (e.g. manager) rather than a specific person
• groups - indicate a set of users who all have the same rights - not Oracle

4/ Views

To restrict information further, a view may be used.

For instance to restrict access to reading CS student date:

create view CS as

select * from STUDENT where Dept = 'CS'

grant select on CS to rich

5/ Profiles

A profile is a named set of resource limits, assigned to a user or role:
 e.g. cpu usage per session, number of reads, etc.

6/ Statistical Access

(Not available in Oracle)

To permit a user to access statistical information but not individual details, a user may be restricted to use only COUNT, SUM, AVG, MAX, MIN, etc.
**Auditing**

Auditing is the monitoring and recording of selected user database activities.

Auditing is used for the following three reasons:

i) to gather statistics from which a system can be optimised
ii) to gather information on data use which is legally required
iii) to investigate suspicious activity

The auditing system maintains an *audit trail* which is a series of time-stamped records which describe user activity.

Oracle provides the following kinds of auditing:

- *Statement auditing* - e.g. use of delete statement
- *Privilege auditing* - use of system privileges - e.g. create table
- *Object auditing* - use of a particular table

Particular roles and users can be audited and an audit command can be made specific to successful or unsuccessful use.

---

**Query Processing and Optimisation**

NB – this is why the Relational Algebra is important!

When responding to a query, the DBMS knows:

(i) what the *query* is
(ii) *statistical information* about the database (how many rows/columns)
and (iii) the *storage structures used* (i.e. are there any indexes, etc.)

From these it can estimate the cost of performing the queries in various ways and implement it accordingly. The strategy is as follows:

(i) Decompose the query into (relational algebra) components
(ii) Estimate the cost (i.e. time to execute) of each component
(iii) Re-organise the components into equivalent forms that are probably faster
  - using equivalences between algebraic expressions
(iv) Estimate the new version and use it if it seems better

---

**Example**

```sql
select member, name from book, loan, member
where book. title = "Dracula" and member. ticket = loan.loaned To
and loan.loaned Book = book.access#
```

There are two equivalent Relational Algebra strategies for processing this:

```
PROJECT( name )
(1 record)

select( title = "Dracula" and )
(100 records)
product BOOK with loanBook
(30,000,000 records)
MEMBER
(100 records)

PROJECT( name )
(1 record)

select( ticket# = loanedTo)
(100 records)
product MEMBER with loanDracula
(300,000 records)

MEMBER
(100 records)

PROJECT( name )
(1 record)

select( access# = loanedBook)
(500 records)
product LOAN with BOOKMEMBER
(300 records)

MEMBER
(100 records)

PROJECT( name )
(1 record)

select( title = "Dracula" )
(1 record)
product LOAN with Dracula
(1,000 records)

LOAN
(300 records)

PROJECT( name )
(1 record)

select( title = "Dracula" )
(1 record)
product BOOK with Dracula
(1,000 records)

BOOK
(1,000 records)
```

---

**Supporting Concurrent Multi-User Access**

Many applications require a lot of users to access the data simultaneously (e.g. airline booking systems)

Uncontrolled simultaneous access can result in chaos, so some controlling mechanism is required.

We introduce the notion of the *transaction* to aid the discussion.

A transaction is a *logical* unit of work which takes the DB from one consistent state to another, i.e. obeying constraints.

It will probably be made up of smaller operations which temporarily cause inconsistency.
The transaction to transfer £27 from the University account to RC's account is made up of two updates:

UniversityAC.balance = UniversityAC.balance - 27 \quad U_1

and \quad RCAC.balance = RCAC.balance + 27 \quad U_2

The DBMS ensures that even if the system crashes or someone asks for the sum of all balances between U_1 and U_2, then it never appears that only U_1 has been executed.

That is the transaction is either wholly completed or fails - transactions are **atomic**!

The transaction gives a single user the illusion of being the sole user of the database.

A **Transaction Processing Monitor (TP)** accepts transactions and integrates their effects on the database.

The four ACID properties, which transactions must respect, are:

- **Atomicity** - all components of a transaction must be performed or none at all
- **Consistency** - the database must be consistent at the beginning and the end of a transaction, where consistency means that the integrity constraints and enterprise rules all hold
- **Isolation** - a transaction must not reveal the effect of updates to other transactions until it completes
- **Durability** - once a transaction and its changes are made permanent, these changes must never be lost

Transactions are used for three purposes in DBMS:

- to determine when **integrity constraint checks** should occur (only at the end of transactions)
- to control **concurrent access**
- to manage **recovery** from system crashes

In introducing many users, we can either **serialise** their transactions or **interleave** them.

We wish to do the latter as we want to use the processor to perform other work while one transaction waits for a disc access.

However, we must **not** allow the transactions to conflict with each other.

This is what is meant by **Isolation**.

Conflict may occur when two transactions are trying to use the same piece of data and at least one of them is trying to change it.
Potential Problems with Interleaved Transactions I

Lost Updates
Consider two transactions A and B which add 10 and 20 respectively to a value V
A and B both take a copy of the original value of V
They both change the value in memory
A puts back its new value first and then B puts back its new value which immediately overwrites A’s change
A’s update is lost!

Transaction A

- get V
- add 10
- put V

Transaction B

- get V
- add 20
- put V

Value of V

5
5
15
25

Key Slide

Potential Problems with Interleaved Transactions II

Temporary Update
Transaction A updates V
A aborts and V's old value is restored
B uses A's updated value
B continues with erroneous value!

Transaction A

- store 20 in V

Transaction B

- get V

Value of V

5
20
5

Key Slide

Potential Problems with Interleaved Transactions III

Incorrect Summary
A updates all the values in a set V
B calculates an average while A is half-way through
B uses inconsistent data

Transaction A

- update V₁
- update V₂
- .......
- update Vₙ

Transaction B

- read the Vₛ
- calculate average

The solution to these problems is to use locks

Key Slide

Locks I

Every time a transaction makes use of a piece of data it notifies the DBMS of this and acquires a lock on that item
This gives it certain access rights, usually one of two types:
- an exclusive Lock (X-lock) means that no-one else can use it
- a shared Lock (S-lock) means that anyone else can also have an S-lock but not an X-lock
  - (NB - Oracle has more than this)
"One writer or many readers"
- When updating, the transaction needs an X-lock
- When retrieving, the transaction only needs an S-lock
Locks II

If a transaction tries to acquire a lock but someone else already holds an incompatible lock, the transaction must wait.

The database system might provide locks at different levels of granularity:
- e.g. locking a cell, a record or the whole table
- the bigger the locking unit the more the system will be slowed down by blocked transactions
- the smaller the locking unit the more lock management needs to be done

Solving the Problems I

Lost Updates

<table>
<thead>
<tr>
<th>Transaction A</th>
<th>Transaction B</th>
<th>Value of V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request X-lock on V</td>
<td>Request X-lock on V</td>
<td>5</td>
</tr>
<tr>
<td>Acquire X-lock on V</td>
<td>Wait</td>
<td>5</td>
</tr>
<tr>
<td>Get V</td>
<td>....</td>
<td>5</td>
</tr>
<tr>
<td>Update V</td>
<td>Wait</td>
<td>15</td>
</tr>
<tr>
<td>Release X-lock on V</td>
<td>Acquire X-lock on V</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Get V</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Update V</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Release X-lock on V</td>
<td>35</td>
</tr>
</tbody>
</table>

Solving the Problems II

Temporary Update

<table>
<thead>
<tr>
<th>Transaction A</th>
<th>Transaction B</th>
<th>Value of V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request X-lock on V</td>
<td>Request S-lock on V</td>
<td>5</td>
</tr>
<tr>
<td>Acquire X-lock on V</td>
<td>Wait</td>
<td>5</td>
</tr>
<tr>
<td>Set V to 20</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Crash</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Roll back</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Release lock</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Acquire an S-lock on V</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Get V</td>
<td>5</td>
</tr>
</tbody>
</table>

Solving the Problems III

Incorrect Summary

<table>
<thead>
<tr>
<th>Transaction A</th>
<th>Transaction B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request X-lock on V1</td>
<td>Request S-lock on V1</td>
</tr>
<tr>
<td>Acquire X-lock on V1</td>
<td>Wait</td>
</tr>
<tr>
<td>Update V1</td>
<td></td>
</tr>
<tr>
<td>Request X-lock on V2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Acquire S-lock on V1 etc.</td>
</tr>
</tbody>
</table>
Three New Problems

1) Which transaction should we start next?
If you make each TX acquire all of its locks before releasing any (two phase or pessimistic locking), you can ensure that you know which can precede which
• this is known as serialising the transactions
Alternatively, you might assume there will be few clashes (optimistic locking) and cope with problems as they occur. A copy is taken (check-out), changes are made and, if all is well, the copy is merged back into the database (check-in)

2) If we have the situation shown left, where two transactions are waiting on each other we have deadlock
The DBMS must carry out deadlock detection and roll back of one of the transactions

3) Support for co-operative work and long transactions
There is a need for more sophisticated locks!

Transaction Management Commands

The user of a DBMS needs mechanisms for grouping updates into transactions. Five different facilities are usually provided:
begin transaction - this starts a sequence of queries which will be treated atomically
commit - end the transaction making all changes permanent and public. This may or may not also automatically be the start a new transaction
abort or rollback - undo all the changes of the current transaction. This may also start a new transaction automatically
checkpoint, validate or savepoint SpointName - mark the current point as a potential point to roll back to
rollback SpointName - roll back only as far as the check point

In some systems, transactions may be nested - i.e. you can start one transaction inside another

Important note - the goal of atomicity is achieved by making transaction commit a single write operation:
i.e. recording a "transaction committed" record in the log (see below)

Transactions in Oracle

There are two possible set ups for transaction management in Oracle.
They are controlled by the variable autocommit:
set autocommit on makes every SQL query a transaction which automatically commits - the default in Aqua Data Studio
set autocommit off provides the following:
starting a session begins the first transaction automatically
commit - end the transaction making all changes permanent and public - this start a new transaction
rollback - undo all the changes of the current transaction and start a new transaction
savepoint SpointName - mark the current point as a potential point to roll back to
rollback to SpointName - roll back only as far as the save point on quitting the user may either commit or rollback

Oracle also allows explicit locking of tables:
lock table Employee in Share mode

Isolation Levels

In fact, it is possible to use the DBMS without being strict about isolation
– The isolation level is an indication of how tough you want to be in isolating transaction
The American National Standards Institute defines four levels:
– read uncommitted - Tx reads data without a read lock
  • reading uncommitted data (called dirty reads) is possible
– read committed - Tx locks reads but releases lock immediately and the same read later might follow a Tx which changes the data
  • reads may be unrepeatable - i.e. two reads of the same data can give different values
– repeatable read – you always get the same value of existing data but it is still possible to work with a set of records into which other transactions might add new records part way through (these are called phantoms)
– serializable – everything locked – as above

SQL has
– SET TRANSACTION [ { READ ONLY | READ WRITE } ]
  [ { ISOLATION LEVEL READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE } ]
Reasons for Transaction Failure

A transaction fails (and rolls back) if:
- the user invokes rollback
- after a user-specified timeout period (INGRES and possibly Oracle)
- the user quits and doesn’t commit (rollback in INGRES and Oracle)
- an abnormal process termination
- the DBMS detects deadlock
- the log fills up
- the system crashes - see recovery - next

Recovery

If the computer crashes while a DBMS is running, each database must be returned to a consistent state when a computer starts up again
- This is achieved by a recovery algorithm
- The ACID property Durability summarises this requirement

A DBMS use a number of techniques for achieving this:
- explicit backup - e.g. Oracle allows the user to backup all or part of a database
- using an undo or redo log - all changes are kept in a separate file which is faster to write to and which can be used to undo or redo work
- shadow pages - an automatic copy of the changed part of the data:
  • users either work with the copy which gets merged in at the end - the after image
  • or with the actual database and the copy can be used to undo work - the before image
- Oracle provides rollback segments - which can be used to undo work in an aborted transaction

Logging Techniques

The DBMS keeps a log or journal of all work, which will be faster to update than the database
- In particular, it records every time:
  • a transaction starts, commits or aborts
  • a piece of data is changed, recording the old and new values
  • a piece of data is read

When a crashed system restarts, each transaction will have either
(i) completed successfully and commit all of its changes - i.e. make them permanent
or (ii) failed in the middle and rollback to the checkpoint

In some systems, the log is used to undo work that is not complete
- In Oracle, all the work in successfully completed transactions is re-written to the store

In fact, the person specifying the transaction can perform a checkpoint in the middle if they are sure this will not cause an inconsistent state - this saves undoing work that has completed successfully
Distributing Processing and Data

There is an increasing requirement to create database systems which are distributed over remote computer systems connected by network.

Reasons:
- the users are distributed
- the data is inherently distributed
- reliability - if one machine is down another may still work
- controlling who can share your data - you keep your data locally, but others can use it
- improving performance - by having many small DB's

It is important to distinguish two kinds of distribution:
- keeping the data centralised, but distributing processing so that many users can access the data simultaneously
  * i.e. client-server systems
- distributing the data - so that many database systems are together providing what looks like a single database

Client Server Architecture

Early DBMS assumed that most computation was carried out on a large mainframe machine with the users sitting at "dumb" terminals

- Commands from a keyboard were sent to the DBMS for processing

Now the processing is distributed among a variety of machines of differing processing power, although the data remains centralised

Client server architectures exploit that by distinguishing two kinds of process:
- a server process is connected to disk holding the data and deals with all aspects of recovering the data from the files, selecting data and transmitting the selected data to the client
- a client process controls the interaction with the user, generates the queries and presents the results

Some different flavours of client server architecture:
- there are usually many clients, but there may be one or more servers
- computation with the data may occur either at the server or the client end
- there may be intermediary processes which migrate – N-tier architectures

Distributed Databases

A distributed db is one in which the data resides in more than one physical database but can be made to look like just one database

The database systems may either be all be the same - homogeneous databases - e.g. all Oracle or can be different - heterogeneous databases - a mixture

Two techniques are combined:
- making the database systems all look the same – e.g. using ODBC or JDBC
- using SQL queries as pipelines

Some issues:
- transparency - a transparent system does not require the user to know where the data being used is.
- fragmentation - how is the data divided?
  * vertically - i.e. keep the whole of each column together.
  * horizontally - keep the whole of each row together.
- replication - is the same data stored in more than one place?

Two Phase Commit

The most difficult task for a distributed system is committing data updates which might be on different machines

Remember the transaction must be committed all or nothing!

Oracle provides a technique called Two Phase Commit to try to help with this, i.e. reduce each commit to a single write at each point, which works as follows:

Prepare Phase: The application on the co-ordinating node sends to all participating nodes a message asking them to prepare to commit

They check that have their redo-log and locks in place and answer:
- "prepared" - ready to commit
- "read-only" - don't need to commit or "abort" - cannot commit

Commit Phase - the co-ordinator gets the participants to write "transaction committed" to the log