Security and administration

Introduction

The database administrator is responsible for the overall effective operation of the database. This includes among other things security, user management and performance. You have already seen some of the activities needed in each of these areas, and this week we will be bringing them together and adding a little more detail.

In this set of notes, we will be using some technical terms from the security and administration domains, and it is useful to have the definitions in advance:

Term	Definition
Authentication	The process of identifying a user
Authorisation	The process of allocating appropriate access rights to a user
Compromise	General term for exposing data to unauthorised access or other damage
Countermeasure	Action taken to prevent or resolve a security threat
Credentials	The security details used to gain access - typically username and password
Privilege	A specific access permissions that can be allocated to a user
Threat	A situation which has the potential to damage the database system of its contents

Threats

Like any other system, a database is exposed to a range of common security threats. Some of them affect all types of system, but because databases make use of very specific technology, and because they contain large amounts of data, there are some threats that are of more concern than others.

The first scenario that may come to mind when thinking about database security is deliberate intrusion by an unauthorised user and the theft of information. However, threats come in many other forms and it is useful to categorise them in some way. Categorisation provides an overview of the variety of ways in which the system may be vulnerable, and like any other hierarchy it can act as a checklist during strategic planning.

Although threats can be categorised in many different ways, the table below is sufficient for the purposes of this module. It is not intended to be comprehensive, and students on security programmes will go on to develop a much more detailed picture.

	Deliberate	Accidental
Human	Abuse of privilege Spoofing	Operational error
Software	SQL injection Database rootkit Denial of Service Worms/viruses	Incompatibility
Hardware	Sabotage	Power failure Disk failure
Environment	Terrorist attack	Natural disaster

Some threats such as denial of service (DoS), worms and viruses are common to all systems that are connected to the internet. Because they are not specific to databases, we will not be discussing the details here. Suffice it to say that they should form part of the company's overall security strategy.

Database rootkits are a special category of software threat which are designed to take control of the database application itself and provide administrator privileges. Although they affect the database specifically, they can only be used if the underlying operating system is compromised, and therefore concern operating system vulnerabilities more than those of the database system. Again, the details will not be considered here.

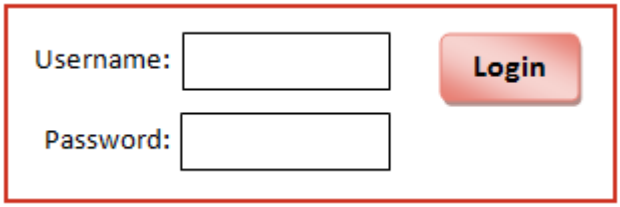
Threats in the hardware and environment categories are most effectively addressed through the appropriate planning and operation of backup strategies which are discussed in a later section of these notes.

Although the table divides other threats into the human and software categories, it is immediately obvious that those in the software category also involve humans beings when the threat is deliberately applied. The purpose of separating the two categories is to highlight the possibility of abuse of privilege by those people who have been specifically granted access to the database. This includes legitimate users taking advantage of their access rights to steal or otherwise compromise the contents of the database, as well as users who have been given access to parts of the data that they do not need to see or use in their working roles. These issues will be discussed under the heading of users and schemas, and measures to restrict access are described in the section on row-level security.

One of the common issues with all of the threats in the deliberate column is that the database security strategy must be able to detect when any of these situations occurs. Cases of spoofing, for example, in which an unauthorised user tries to gain access by using someone else's credentials can be very difficult to detect at the time, but in a later investigation, the existence of a record of all logins can be crucial in identifying them since the real user will be elsewhere at the time. Control over user accounts and the privileges they have are therefore one side of the equation, while an accurate audit trail of system access is another. These measures have a technical side which is the responsibility of the DBA, and they also have a procedural side which is typically set at a higher level of management. The security policies and procedures define how, when and how often user details are reviewed and what records are kept of account changes and system access. In this module, we are more concerned with the practical aspects of database administration, and these will be discussed in the section on users and schemas. The policy and procedural side will be left for later modules in relevant programmes of study.

One very specific threat to database security is SQL injection in which an attacker uses technical knowledge about the syntax of SQL and the architecture of database applications to gain unauthorised access to data. To understand fully how SQL injection works and how to guard against it requires an understanding of application architecture which not all students have at this point.

Briefly, SQL injection means forcing an application to execute a series of SQL statements rather than the single one that was planned by the application programmer. The following example will illustrate:



Let us say that an application requires users to log in, and presents the usual dialog as shown on the right. The user enters their credentials and clicks on the *Login* button. In order to verify the user's identity (authentication) and allocate the appropriate privileges (authorisation), the application code must embed the credential values into an SQL statement such as the one below. You have already seen the use of substitution variables where the variable name is replaced by the supplied value when the statement is executed. When the correct values are used, the statement below works as planned.

```
SELECT   access_level
FROM     users
WHERE    username = '&username'
AND      password = '&password';
```

However, let's say that instead of entering a simple password into the field on the screen, the user enters the string:

```
rubbish'; SELECT * FROM users;
```

Because of the careful placement of the single quote and semicolon characters, when the string is substituted for the variable *&password*, the result is actually two SQL statements and one statement fragment as shown below. When this short script is executed, the first statement returns no rows, but the second one displays all of the data in the USERS table. This allows the attacker to select a set of genuine login credentials and spoof a real identity. The remaining code fragment simply produces an error.

Statement 1	`SELECT access_level`
	`FROM users`
	`WHERE username = 'NoName'`
	`AND password = 'rubbish';`

Statement 2	`SELECT *` `FROM users;`
Fragment	`';`

Countermeasures against SQL injection can be built into the application code, and are therefore left for later application development modules.

Another threat which can be best countered by the design of the application is operational error. Although it is impossible to completely eliminate human error, an application can include verification checks and other constraints on the data that can be entered by a user. To a certain extent, user training and company procedures can also be effective against this threat, but again they is beyond the scope of this module.

Finally, because the DBMS is a software application that is installed alongside others and depends on the services provided by the operating system, it is possible for incompatibilities to arise. This can happen for example when the operating system or some component of the operating system is upgraded and the existing DBMS is only compatible with the earlier version. The only effective way to guard against this type of threat is for the DBA and other systems administrators to be systematic and rigorous in the way they plan and implement upgrades. As well as paper research with the software release notes, the planning process can involve a trial run of the upgrade on a copy of the operational system followed by a period of testing to ensure that the new configuration is stable. These issues however start to delve into the details of the DBA's work that are well outside the scope of this module.

Authentication and authorisation

Authentication

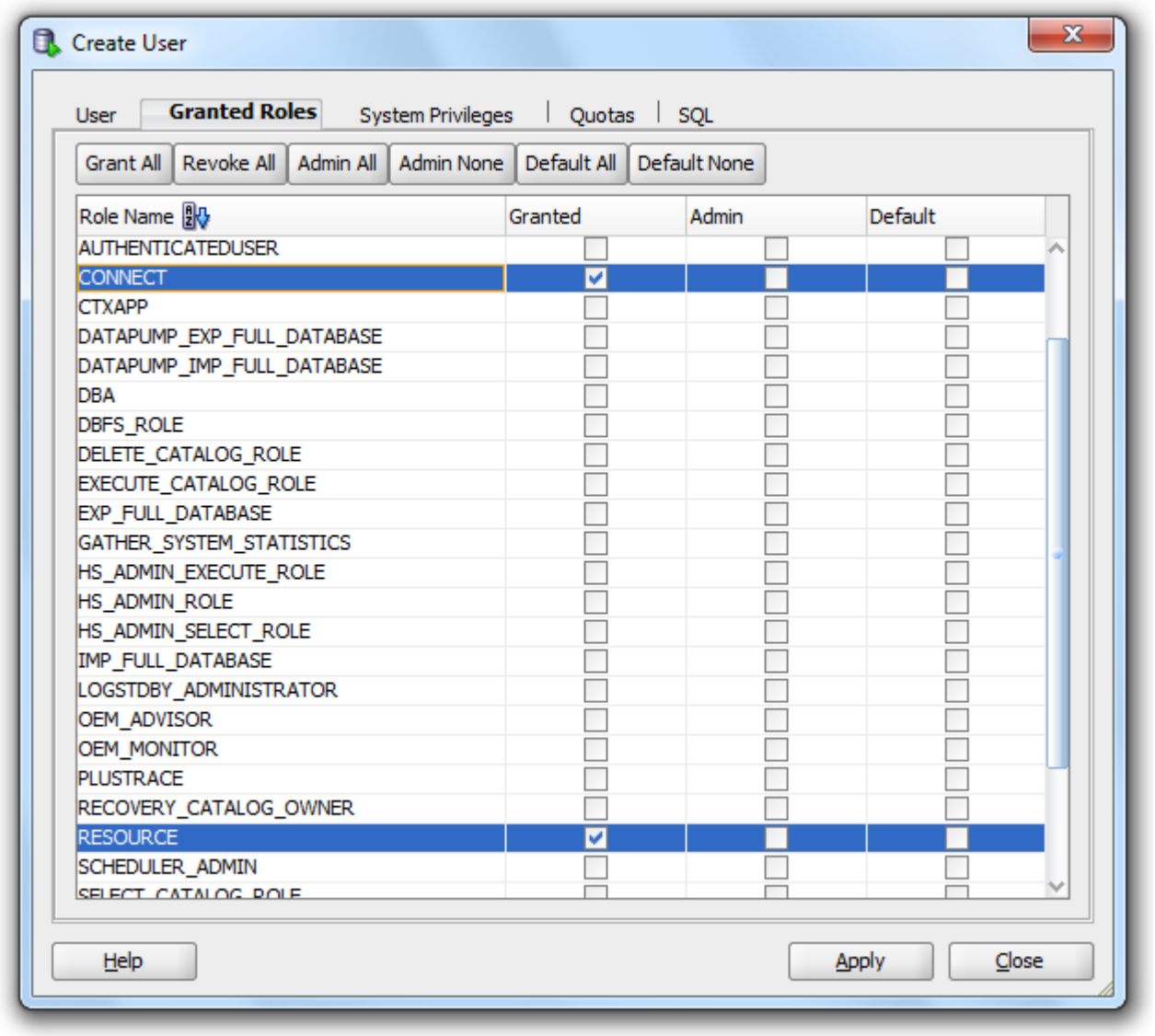
The standard way to authenticate a user is to check a standard set of credentials such as a username and password against those held by the system. If the system recognises the password supplied by the user, we assume that the user is who they say they are. This is the end of the authentication process, however. Authentication in itself does not confer any particular privileges to the user; it simply succeeds or fails to identify them.

It is tempting when discussing authentication to digress into alternatives to the standard username and password. Systems can, for example, include physical tokens or biometric measurements which can be used for authentication, and the relative effectiveness of these methods is worthy of consideration if security is a particular concern. However, these subjects are not appropriate in a database module at this level simply because they are components of the hardware system, and therefore independent of the DBMS. In any case, the general model of authentication is the same for all the methods mentioned: a piece of information supplied by the user (password, fingerprint, etc.) is compared to a known value, and if it matches the system confirms the user's identity.

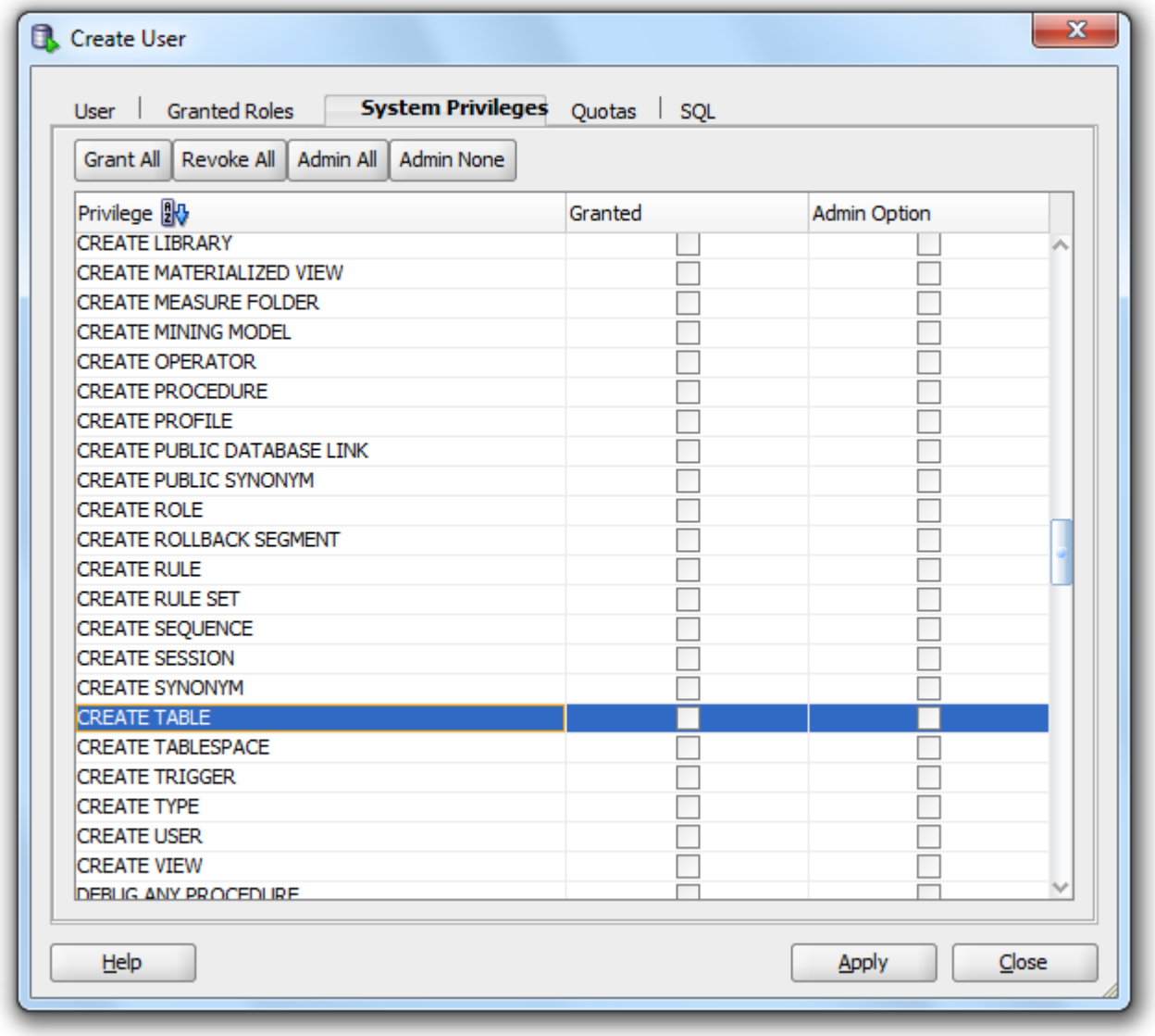
Authorisation

Once a user has been authenticated, the appropriate level of access is determined with reference to information stored about the authenticated user account. In most relational databases, access is controlled by a set of standard privileges which can be granted to a user. Privileges are maintained independently of the user account, so that an account can exist with no privileges at all, and privileges can be added or removed as required.

The most basic privilege is to allow the user to establish a session with the database. Thereafter the user's ability to perform basic functions can be controlled by adding further privileges such as being allowed to create tables. When you created new user accounts earlier in the module, you assigned them the two user roles CONNECT and RESOURCE as shown below.



In Oracle, roles provide a way to assign several privileges to a user in group rather than assigning them separately. The RESOURCE role includes, for example, the privilege to create and drop tables and to allow other users to access owned tables. The list of individual privileges can be seen by activating the *System Privileges* tab in the *Create User* dialog as shown below.



Each privilege is very specific which give the DBA a high degree of control over what users are permitted to do.

The dialogs in SQL Developer simply provide a GUI for the standard method for adding or removing privileges on a user account which is to use the SQL keywords GRANT and REVOKE. The table below shows some example SQL statements using these keywords and their interpretations.

Statement	Meaning
GRANT CREATE SESSION TO bob;	Allow the user bob to start a session with the database
GRANT DROP USER TO bob;	Allow the user bob to drop user accounts
REVOKE CREATE TRIGGER FROM bob;	Remove the ability to create triggers from user bob

A full list of Oracle system privileges can also be found under the entry for GRANT in the SQL Language Reference.

The GRANT and REVOKE commands can be used with roles as well as with privileges which makes command line administration simpler. The example below shows the creation of a user account using the GRANT command with the additional INDENTIFIED BY clause to supply a password. This same statement can be used to change the password for an existing user. Note that single quotes are not required for the password value.

```
GRANT CONNECT to bob IDENTIFIED BY b0b123;
```

Users and schemas

So far you have seen that there is a distinction between user accounts with administrative functions and rights and other database users. For example, you have used the SYSTEM account to create new users and to perform some administrative operations such as exporting a schema. To create a schema (ie a coherent set of tables), you have used an account such as HR or SPECIES.

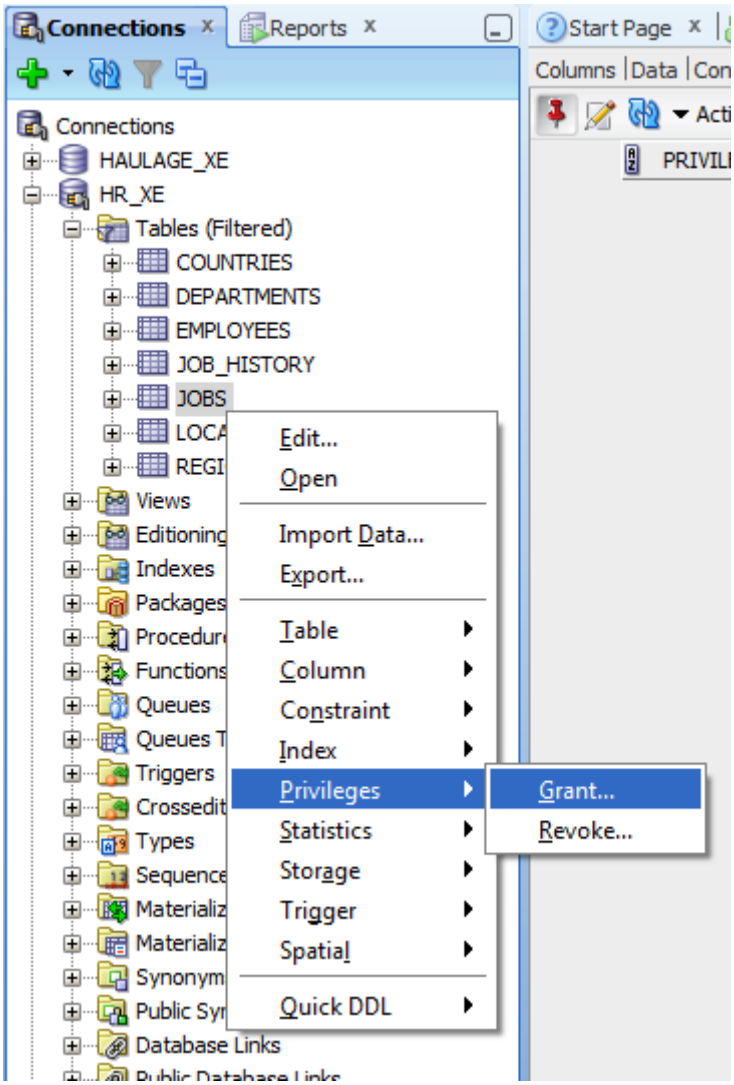
The user account which owns a set of tables defines the schema: it collects the set of related tables together so that it can be treated as a single object. However, databases are designed to support multiple simultaneous users who access the same schema via an application interface. There is therefore a third category of user who does not own any tables, but accesses table owned by the schema account. This is the reason for the three standard roles in Oracle: DBA accounts such as SYSTEM are granted the DBA role, schema owners such as HR have the RESOURCE role, and other users of the HR application have the CONNECT role.

The system level privileges are not sufficient on their own however to provide access to the HR tables for the ordinary user. This level of security is delegated to the schema user account. Thus the HR user will be responsible for granting or revoking privileges on the database objects within the schema.

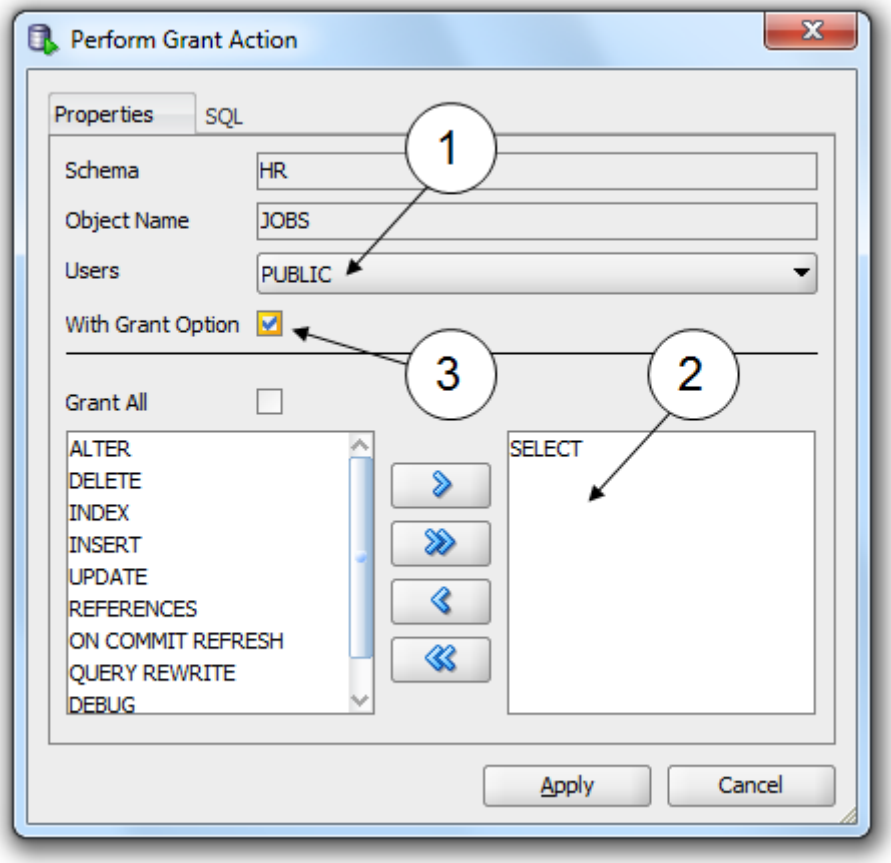
For example, say a new member of staff joins the HR department. The DBA will log into the database using the SYSTEM account to grant overall access a the new account with the CONNECT role and to provide a password. Then either the DBA or another administrator will log into the database as the HR user and grant access to all of the objects that the new account needs to successfully operate the HR application. The table below illustrates some of the object-level operations that could be performed.

Statement	Interpretation
GRANT SELECT ON employees TO bob;	Allow user bob to query the EMPLOYEES table
GRANT SELECT, UPDATE ON job_history TO bob;	Allow user bob to query and update the JOB_HISTORY table
GRANT ALL ON departments TO bob;	Grant all privileges on the DEPARTMENTS table to user bob
REVOKE UPDATE ON departments FROM bob;	Remove the privilege to update the DEPARTMENTS table from user bob

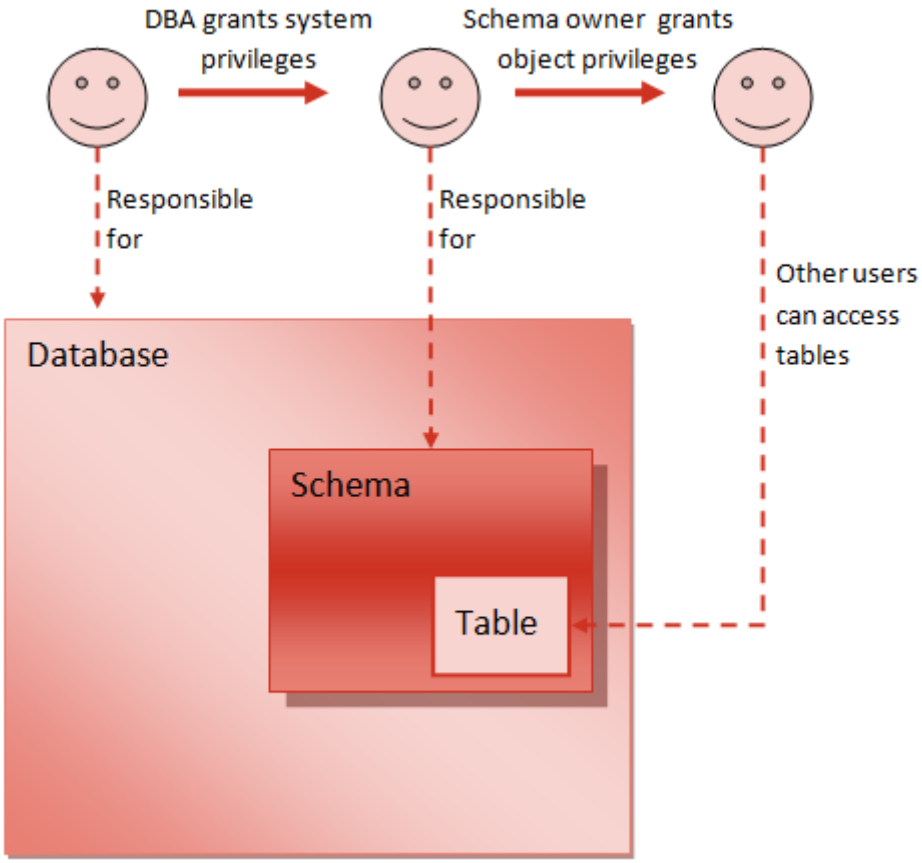
SQL Developer allows a user to manage the security of its objects by choosing the *Grants* option in the object browser as show below.



The graphic below shows the dialog used to grant privileges to the JOBS table. Note that the *grantee* is the user the privilege is given to; the schema owner is known as the *grantor*. In this example, the grantee is PUBLIC (1) which means "any user". The privileges selected from the left-hand pane can be seen in the right-hand pane (2) and these will be assigned to the selected user when the *Apply* button is clicked. *With Grant Option* (3) allows the grantee to grant access on this object to other users. This can be useful for delegating the management of parts of a large application to lower level managers, but opens up the possibility of additional security risks.



The two levels of security and the three different categories of user can be summarised in the following diagram. You should note that this is the model used by Oracle to manage access to schema objects. The mechanisms used by other database platforms such as SQL Server have variations of this arrangement.



Row-level security

The system of privileges that are provided by the DBMS provide adequate security at the system level and at the level of the individual database object. Thus the DBA can prevent or allow access to the system, and schema owners can prevent or allow access to tables, views and so on. Different levels that involve increasingly small differences are referred to as *granularity* - the smaller the object, the lower or finer the level of granularity.

There are situations though where access control over individuals objects does not provide a sufficiently fine level of granularity. Take for example a hospital records system where consultants are only allowed to see the records for their own registered patients in order to protect the privacy of others. Here it is not sufficient to allow or deny access to the patient records table because it contains all records. A finer level of granularity is required whereby a user can have access to certain rows in a table but not others.

The DBMS provides a solution to this problem through the use of views. A view can be created which extracts only the required records from a table. When access privileges are granted on a view, they are independent of any access rights on the underlying table. This means that a user can have access to that subset of records in the view without having any rights at all on the rest of the table. The following example illustrates this.

Assume that the hospital system contains the PATIENT table shown on the right. The column *assigned_consultant* contains the identifier for the consultant. If we further assume that the consultant identifier, as well as being the primary key in the CONSULTANT table, is also the username that the consultant uses to log into the database, we can user that information to select the appropriate columns from PATIENT.

patient
patient_id
first_name
last_name
gender
date_of_birth
last_admission_date
assigned_consultant
date_of_death

Oracle and other database platforms always provide some way to identify the current user. In Oracle it is done using a *pseudo-column* caller USER. A pseudo-column is a context dependent value which behaves like a column with respect to SQL statements, but which is not actually stored in a table. SYSDATE is another example.

If it is referenced, USER evaluates to the username of the current user. If we include it in the definition of a view therefore, the contents of the view can be made to vary depending on which user is performing the query.

In the hospital example, we could create a view called ASSIGNED_PATIENTS using the following statement:

```
CREATE VIEW assigned_patients AS
SELECT *
FROM   patient
WHERE  assigned_consultant = USER;
```

Backups

When faced with a situation in which the entire database has been rendered unavailable, such as a major act of sabotage or terrorism, or a more likely natural disaster or hardware failure, there is no alternative but to restore the database from a backup. This highlights the needs however, for there to be a backup and recovery strategy to be in place, and for the relevant procedures to be followed thoroughly.

Ideally, every database transaction should be applied to parallel systems so that if one system fails, the other is still operational. Relational database products are sufficiently advanced to do this; however, there are significant barriers. Parallel running is a very expensive option. Database suppliers realise that parallel operations brings benefits in terms of data security, and the licence costs are therefore typically double. In addition, there is a resource overhead to making multiple updates. On busy systems, a small delay associated with each transaction may be sufficient to reduce the overall performance of the whole system. This might mean further costs for more powerful hardware. Finally, even having two parallel systems does not guard against the possibility of both systems failing at the same time, or the second system failing soon after the first which amounts to the same thing. In this case, a backup strategy is still required.

In practice very few organisations can afford to maintain parallel systems just to protect their data, and most rely on carefully maintained backup and recovery procedures for the security of the data.

Types of backup

The simplest and most obvious type of backup is the full database backup. The complete contents of the database is extracted, compressed and stored in an offline file which can be placed in secure storage until it is needed for recovery. Although this sounds like it should be sufficient protection, there are some serious practical issues to take into consideration.

A full database backup can take a long time, and in order to maintain the integrity of the data while the backup is taking place, it is common to prevent users from making any transactions. This can be a

major inconvenience for 24-hour operations like e-commerce companies where there is a strong business pressure to minimise the time that the system is unavailable (downtime).

A further disadvantage of full backups is their size. Because each one contains the entire contents of the database, holding several of them can take up a significant amount of storage space, and they are difficult and time-consuming to move from one location to another. The reason for keeping several backups at a time is discussed below.

Many companies employ storage specialists to store backup files in secure locations on their behalf. If the files are large, this may increase the cost of using these facilities.

An alternative to the full backup is the incremental backup. Whereas the full backup contains all data, the incremental backup only contains the contents of the transaction log (introduced in week 9). The transaction log is flushed after each backup, so as long as there is one recent full backup, further incremental backups can be taken which contain only the more recent changes, and which are significantly smaller and easier to handle. If the transaction log is implemented as an external file, the time taken to make an incremental backup could be reduced to the time taken to copy it to another location.

As part of a system recovery, the final step would be to recover the most recent database changes from the current transaction log. If used correctly, this ensures that in the event of a full system failure, the only data that is lost consists of those transactions that were actually in progress at the time, and which had not yet been committed to the database.

Backup strategies

The disadvantage of using an incremental backup is that it extends the time required to perform a database recovery. If we have a full backup from day 1, for example, and incremental backups from the next two days and there is a system failure on the third day, there are three steps required to recover the system. First, the system must be recovered using the full backup, and then the incremental backups must both be applied in the correct order.

A further consideration is that backup files themselves may suffer damage and become unusable. It is therefore common practice to maintain a series of backups so that it is possible to recover the system from the most recent usable backup file.

Planning a backup strategy is therefore a question of balancing the advantages and disadvantages of the different types of backup against the company's priorities. A range of solutions is possible, and the examples below illustrate some of the possibilities.

Example 1: Mission-critical 24-hour operation

The priority in this case is to minimise downtime as much as possible. Given sufficient budget, the system could use parallel databases, but a last-resort backup strategy would still be required. Because a full backup is time-consuming and the risk of needing to perform a full system recovery would be significantly reduced through the use of parallel databases, the time between full backups could be quite long. The assumption is that any disaster large enough to cause both database servers to fail is likely to affect the company in other ways too, and a general return to business will therefore take a long time anyway. The time required to restore the system under these circumstances would therefore not be an issue.

We might therefore decide in this case to perform a full backup once a month at the least busy time, and to take incremental backups once a day in the intervening period. Although a full system recovery might be very time consuming with many incremental steps, we assume that this would be acceptable in exceptional circumstances. An external contractor will be used to store backup files in a secure location.

Example 2: University

Although the database systems are central to the administration of the university, the core business does not depend on them, and extended periods of downtime are therefore acceptable. Backup files could be stored with an external contractor, but given the need to keep costs as low as possible, it might be better to reach a cooperative arrangement with another university so that the backup files for one partner are stored on the premises of the other. The assumption is that most disasters will only affect one site, and that any event which compromises both sites is likely to have far wider impact that just the administrative databases. The risk to the backup files is therefore deemed acceptable. Whether an external contractor or a partner university undertakes to store the backup files, their size and volume may still be an issue, and therefore a balance needs to be found between full and incremental backups.

In this case, we might decide to take a full backup once a week at the weekend, and an incremental backup every day in the intervening period. Two sets of backup files could be maintained for extra resilience covering the most recent two-week period.

Example 3: Small business

Assuming that the business depends on the database systems only for administration, and that the staff are capable of keeping alternative records if necessary, the time needed to perform backup and recovery operations is not a major issue. In this case the database is also likely to be quire small, and a full backup is unlikely to take very long. It is still good practice to store at least one recent backup in a secure location, so a minimal contract with a storage provider would be a good idea.

Here we might decide to take a full backup after the close of business each evening. The backup from the last working day of the week could be entrusted to a secure storage specialist to provide a recovery route in case all other backups are damaged. The maximum data that could be lost in this situation would be one week's worth of transactions, but given the option to re-enter that data, we judge that the costs saved on the minimal backup strategy outweigh the risk.

Performance

Another aspect of the DBA's responsibilities is to maintain the performance of the database. Although the structural design is determined by a systematic and logical design process, practical issue may arise later that prevent it from responding as quickly as it should. The most likely reason for a degradation in performance over time is the accumulation of large amounts of data. Financial institutions, stock trading platforms and e-commerce firms all process thousands of transactions a day, and the volume of data can quickly escalate. Ideally, these issues should be addressed at the design stage, but occasionally the problem only surfaces later.

An obvious approach to resolving performance issues is to invest in more powerful hardware. This can be expensive however, and significant time will be necessary to install the new system, configure it appropriately, install the required software, configure the DBMS, migrate the data from the old system to the new system, and parallel run with the old system until any remaining issues have been identified and resolved. It is not cheap, not quick, and may even fail to solve the problem. Until the precise cause of the performance drop is identified, putting money into new hardware is just guessing.

Archiving

Another obvious route for the DBA to take is to reduce the amount of data in the database. It is good practice to archive old data on a regular basis in any case, and there may be further advantages to doing so. For example, the company may wish to analyse database transactions over a long period of time. This will often provide useful strategic information to business managers; however, such analyses consume a lot of computing resource and are not normally possible while the system is in operation. Old data is therefore often extracted and loaded into a second database known as a *data warehouse*. Apart from the occasions when new data is added, the contents of the data warehouse is

fixed. This differentiates it from the operational database which is known as a online transaction processing (OLTP) system. The structure of the data warehouse, or online analytical process (OLAP) system, can therefore be optimised for fast data retrieval rather than to accommodate the storage and updating of records.

Notice that data is rarely ever completely deleted from a database system. Even if the old data is not required for loading into a data warehouse, it will typically be maintained in offline storage of some kind for a particular length of time. For certain types of data such as financial accounts, electoral results and records of company formation there are legal requirements on how long data must be maintained. This paper by Watson Hall information security specialists neatly summarises the UK legal regulations on data retention.

For practical reasons, it may not always be desirable to remove old data from the online system. In a system where existing records can act as references for new transactions such as a helpdesk or medical records database, removing data may reduce the overall effectiveness of operation.

Indexes

As mentioned in week 7, the speed of a query can sometimes be improved by adding and index. When considering this option, the DBA needs to identify a query that is too slow and examine the WHERE clause to determine whether an index would be of use. Oracle and several other database platforms automatically create indexes for primary and unique keys, but other indexes must be added explicitly. If the WHERE clause of the query uses a non-key column to filter the table contents, an index on that column could help.

patient
patient_id
first_name
last_name
gender
date_of_birth
last_admission_date
assigned_consultant
date_of_death

For example, in the hospital example from earlier, the *patient_id* column is the primary key. A query which filters patient records by *last_admission_date*however may begin to run slowly once a large number of records accumulates. This issue may have been identified and addressed at design time, but if it was not, a full table scan will be required every time the query is run. Creating an index on *last_admission_date* would therefore improve performance.

Designing an appropriate index is not always that simple, since several columns can be included in a single index and success may depend on getting the right combination; however, further detail in this area is beyond the scope of the module.

In the simple case discussed here, the index could be created using the following SQL statement:

```
CREATE INDEX p_index1 ON
patient(last_admission_date);
```