SOFT20101

Information and Database Engineering

Revision Guide

Matthew Robinson

SQL

Theory: Introduction to SQL

What is a relation? 7
What is Structured Query Language (SQL)? 8
Data types 10
Creating databases 11
Table operations 12
Primary keys 15
Constraints 17
Foreign keys 19
Oracle administration 22

Practical: Introduction to SQL

Creating tables 23
Describing tables 24
System tables 25
Inserting data 26

Theory: Joins and Functions

`SELECT` statements 28
`SELECT DISTINCT` statements 29
Set operations 30
Inner joins 33
Outer joins 34
Single table self-join 37
Built-in functions 38

Practical: Joins and Functions

Set operations 39
Joins 40
Built-in functions 41

Theory: Subqueries and Views

Subqueries using `IN` and `NOT IN` clauses 42
Expressing subqueries as joins 45
Subqueries using `ANY` and `ALL` clauses 46
Subqueries using `EXISTS` and `NOT EXISTS` clauses 47
Operations using subqueries 49
Nested subqueries 50
The `ORDER BY` clause 51
The `GROUP BY` clause 52
The `HAVING` clause 54
Views 55

Practical: Subqueries and Views

Subqueries using and joins 57
Clauses 58
Views 59

Theory: PL/SQL and Stored Procedures

Introduction to PL/SQL....60
Blocks 61
Stored procedures....62
PL/SQL control and variables 65
PL/SQL implicit cursors, output and explicit cursors 69
Stored functions....72

Practical: PL/SQL and Stored Procedures

Stored procedures....74
Stored functions....76

Theory: PL/SQL and Triggers

Introduction to triggers....77
Table-level triggers....78
Row-level triggers: new and old 80
Row-level triggers: the `WHEN` clause 82
Evaluation....84
SQL extensions 85

Practical: PL/SQL and Triggers

Table-level triggers....86
Row-level triggers 88

DBMS

Theory: Introduction to DBMS

What is a DBMS?....92
Role of a DBMS....93
DBMS schemas....94
Conceptual level: tablespaces and datafiles....98
Internal level: pages 99
Structure of a DBMS....101
Types of DBMS 105
RDBMS....108

Practical: Introduction to DBMS

Referential integrity in DBMS 110

Theory: Query Optimisation

Physical design 113
Query processing 114
Query speed 117
Query optimisation strategies 118
Indexing....123
Evaluation....126

Practical: Query Optimisation

Indexes 127
The DATE data type 128
Functions in INSERT and UPDATE operations 129
Functions and formatting in output from SELECT operations 130

Theory: Data Administration and Security

Data dictionary 131
Core administration tasks 132
Database security 133
Database level security: user authorisation/authentication, audit trails and encryption 137
Database level security: users and profiles 138
Database level security: controlling user access 139

Practical: Data Administration and Security

Controlling user access with object-level privileges 145
Controlling user access with views 146

Theory: Transactions and recovery

Introduction to transactions and recovery 147
Properties of transactions 148
Operation of the database and recovery 151
Log files 153
Failure of transactions 156

Practical: Transactions and recovery

Updating data 159

Theory: Concurrency control

Introduction to concurrency control 160
Locks 161
Serialisability and two-phase locking (2PL) 166
Deadlock 169
Timestamp protocol 170
Evaluation of protocols 171

Practical: Concurrency control

Complex queries 172

Theory: OODBMS & ORDBMS 1

Why use an object-oriented approach? 175
RDB spatial databases 177
OODBMS 179
Strategies for OODB development 182
ORDBMS 183

Practical: OODBMS & ORDBMS 1

Object types 189
Object types in tables 190

Theory: OODBMS & ORDBMS 2

Inheritance and polymorphism 191
Inheritance in an RDB spatial database 192
Inheritance in an OODBMS 194
Inheritance in an ORDBMS 197

Practical: OODBMS & ORDBMS 3

Subtypes 201
Subtypes in tables 203

SQL

Theory: Introduction to SQL

What is a relation?

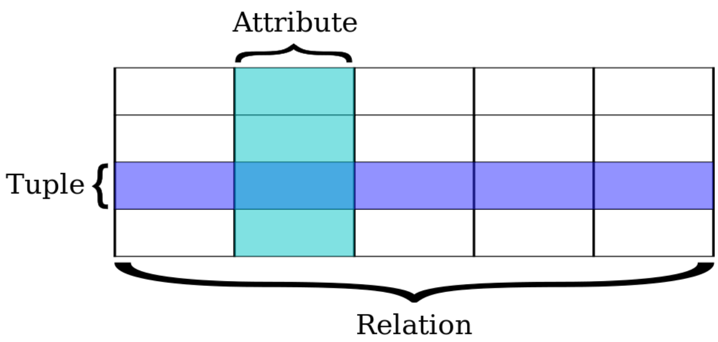
Definition

A **relation** is a two-dimensional table that contains a set of tuples (`d1`, `d2`, ..., d_n), where each element `dj` is a member of `Dj`, a data domain.

How is data organised?

A relational database consists of one or more named relations and relations act as the main structure for a database. The connections and relationships in a relation are represented by values of data.

Diagram: A relation



This shows that a relation in a set of columns, which contain tuples, and rows, which contain attributes.

The order of the rows is non-important as they are identified by content and not by position.
The order of the columns is non-important as they are identified by name, not by position.

Each row is distinct, and each column has a set of allowed values (type and range) or domain.

For example, a relation named `Lecture` could contain data about different lectures taking place in a university.

Example: `Lecture` relation

name	roomNo	occurs	Lecturer
IDBE	128	12-10-14	FitzGerald
SDI	307	13-10-14	Hibberd
IDBE	128	19-10-14	FitzGerald
SDI	307	20-10-14	Hibberd

In this example:

- the name of the relation (table) is `Lecture`;
- the attributes are the named columns – `name`, `roomNo`, `occurs` and `Lecturer`;
- the tuples are the rows of data; and
- each cell contains a single data value.

The database management system (DBMS) provides a layer of abstraction on top of the file management system. As a result, database managers need not be concerned with how exactly the data is stored in order to interact with a database. Instead, a knowledge of Structed Query Language (SQL) is required.

Theory: Introduction to SQL

What is Structured Query Language (SQL)?

Definition

Structured Query Language (SQL) is a declarative language and is both a data definition language (DDL) and data manipulation language (DML).

SQL as a DDL	SQL as a DML
• Create tables. • Alter tables. • Drop tables.	• Perform queries. • Insert data. • Update data. • Delete data.

SQL is mostly a non-procedural language that has been derived from relational model and tuple calculus.

Some other types of databases have different variations of SQL and other databases have distinct boundaries between DDL and DML, however SQL covers both.

Abstraction

The database management system (DBMS) provides a layer of abstraction on top of the file management system. As a result, database managers need not be concerned with how exactly the data is stored in order to interact with a database. Instead, a knowledge of Structed Query Language (SQL) is required.

This means that the database manager can specify what data is required and the action to take, rather than specifying how the action must be taken.

History

SQL was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce after learning about the relational model from Ted Codd in the early 1970s.

Relational algebra is a family of algebras with a well-founded semantics used for modelling the data stored in relational databases. DML operations are based on the eight basic operations from relational algebra that act on relations.

The ANSI SQL standard was released in 1986 (SQL-86) and the ISO SQL standard was released in 1987. In addition, SQL has remained standardised (SQL2 or SQL-92, SQL3 or SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016).

Theory: Introduction to SQL

Information: SQL timeline

Standard Revision	Enhancement	Functionality
SQL:1999	Triggers	Programs that respond to events relating to a given table.
SQL:1999	Stored procedures	Predefined sequences of SQL statements. This includes a language that can be used alongside SQL which behaves more like a procedural programming language.
SQL:1999	New data types: Object-oriented features	Able to store classes and objects.
SQL:2003/2006	XML	Able to store XML files and introduced instructions for searching XML files, similar to the capabilities available in a native XML database.
SQL v12	JSON	Able to store JSON files.
	Application Programming Interface (API)	ODBC and JDBC provided standard set of DDL/DML functions. These allowed support for programming languages, such as C++, C#, Java and PHP etc., to connect to a database and interact with data in tables and therefore support storage of files.

This SQL timeline shows that the ISO standards are responsive to changes in technology.

Theory: Introduction to SQL

Data types

Definition

Data types are predefined types of data that are supported by a language.

Principal SQL data types

Data Type	Keyword(s)	Description	Example
Character	• `CHARACTER(6)` • `CHAR(6)`	Character string of fixed length `n`.	“John ”
Varying Character	• `CHARACTER VARYING(6)` • `VARCHAR(6)`	Character string of variable length and maximum length of `n`.	“John”
Integer	• `INTEGER` • `INT`	Whole number.	12
Decimal	• `DECIMAL(<precision>,<scale>)` • `NUMERIC(<precision>,<scale>)`	Decimal number with fixed precision. Two parameters are defined. • `precision` – Digits before decimal. • `scale` – Digits after decimal.	12.345
Float	• `FLOAT(p)` • `REAL` • `DOUBLE PRECISION`	Decimal number with variable precision. The decimal number will be rounded if necessary.	12.345
Date	• `DATE`	Stores dates in the format `DD-MONTH-YY`.	27-NOV-04

Theory: Introduction to SQL

Creating databases

ISO standard

The ISO standard for SQL does not specify how databases are created. This is because the process for creating a database can differ between products.

Set-up overheads

In MySQL, it is possibly to easily create large numbers of simple databases. However, in Oracle, creating a database can be complex due to a large number of user-set parameters.

Whilst it is possible to create a database by setting a minimal number of parameters, it is generally good practice to not leave these parameter values as default.

This means that an Oracle database is often more complex and has larger set-up overheads, such as a schema.

A **schema** is a collection of database objects (tables, indices, assertions, views and privileges) with one owner. Users can create more database objects which only they see. In each database there may be several schemas.

Theory: Introduction to SQL

Table operations

Creating tables

Format

In Oracle SQL:

- all columns are enclosed by brackets (`()`) and each column is a column separated value; and
- each column is listed with the attribute name and data type.

Format: Creating a table in Oracle SQL

```
CREATE TABLE <table name> (
      <column name> <data type>,
      <column name> <data type>,
      …
      <column name> <data type>
);
```

It is possible to create a table without defining a primary key as the DBMS assumes that a primary key will be added later using an `ALTER TABLE` statement. However, this is not good practice as all tables should always have a primary key – this is discussed further on page 12.

Example

Example: Creating the table `Lecture` in Oracle SQL

Create a table named `Lecture` with a `name` column to take strings up to length `10`, a `roomNo` column to take integer values, a date column `occurs` and a `lecturer` column to take strings up to length `10`.

```
CREATE TABLE Lecture (
      name VARCHAR(10),
      roomNo INT,
      occurs DATE,
      lecturer VARCHAR(10)
);
```

Theory: Introduction to SQL

Altering tables

A database manager may wish to alter a table in a database in order to change its characteristics without the need to delete and re-create the table. This may be necessary to prevent loss of already entered data in the database or to prevent downtime as access to the database is needed by users.

General format

Format: Generic `ALTER TABLE` statement
`ALTER TABLE <table name> <option> <parameters>;`

Adding a column

A database manager may wish to add a column if a new piece of data is to be stored about the rows in a table.

Format: `ALTER TABLE` statement to add a column
`ALTER TABLE <table name> ADD <column name> <data type>;`

Example: Adding a duration column to the `Lecture` table
Alter the table `Lecturer` and add the column `duration` to take a decimal number with a precision of 4 and scale of 2.
`ALTER TABLE Lecture ADD duration DECIMAL(4,2);`

Removing a column

A database manager may wish to remove a column if the data stored in that column is no longer necessary in the table.

Format: `ALTER TABLE` statement to remove a column
`ALTER TABLE <table name> DROP COLUMN <column name>;`

Example: Adding a duration column to the `Lecture` table
Alter the table `Lecturer` and remove the column `duration`.
`ALTER TABLE Lecture DROP COLUMN duration;`

Renaming a column

A database manager may wish to rename a column if a mistake was made during the creation of the database. Renaming a column allows the existing name of a column to be overridden by a new name.

Format: `ALTER TABLE` statement to rename a column

```
ALTER TABLE <table name> RENAME COLUMN <current name> to <new name>;
```

Example: Adding a duration column to the `Lecture` table

Alter the table `Lecturer` and rename the column `duration` to `lectureDuration`.

```
ALTER TABLE Lecture RENAME COLUMN duration to lectureDuration.
```

Deleting tables

A database manager may wish to delete a table if the relation becomes redundant or the data stored is no longer required.

Format

The `DROP` command allows the removal of database components.

Format: Deleting a table

```
DROP TABLE <table name>;
```

Example

Example: Deleting the `Lecture` table

Delete the table `Lecturer`.

```
DROP TABLE Lecture;
```

Additional options

`RESTRICT` – Specifies that the command must not be carried out if the component being deleted is not empty. This is the default action for a `DROP` statement.

`CASCADE CONSTRAINTS` – Specifies that the component must be removed together with the components that have dependencies on the component.
This will ensure that deleting a component with a primary key will also delete the components with related foreign keys. A DBMS will not allow a component with a primary key which has records with related foreign keys to be deleted therefore it is necessary to perform a cascading delete. A cascading delete can be used to remove a component with a primary key while maintaining the referential integrity of the database.

Theory: Introduction to SQL

Primary keys

Definition

A **primary key** is a set of columns that together makes each row have a distinct set of column values. A primary key can uniquely identify a row in a database and is indexed for quick searching and retrieval.

A primary key can be:

- a simple key – consists of a single attribute which uniquely identifies a record;
- a composite key – consists of more than one attribute, where all attributes are a simple key in their own right, which uniquely identifies a record; or
- a compound key – consists of more than one attribute, where one or more of the attributes are not a simple key in their own right, which uniquely identifies a record.

This means that, in a relational database:

- it should be possible to write a statement to select data that returns only one row;
- duplicate rows are forbidden as no two rows should have the same value of the primary key;
- every table should have a primary key, even if it is possible to create a table without a primary key; and
- a primary key should not contain a NULL value as then it may not be possible to identify some rows as more than one row could end up having a NULL value primary key making the rows non-unique.

Using IDs

Some database managers often use identification numbers, such as "Book ID" to identify books or "Customer ID" to identify customers.

Instead, it is considered best practice to instead try to find a combination of columns (attributes) that can be combined in to a composite or compound key in order to create a unique identifier for a given row.

In some cases, an ID is appropriate to use as it contains an actual piece of useful data anyway. An example of this is student numbers often used to identify students at education institutions such as universities. Although, an ID should not be added unnecessarily as this creates additional storage requirements.

One column primary key

If a table is being created with a single column primary key, it is possible to append the keyword "`PRIMARY KEY`" after the attribute name and data type for the row.

Format: Creating a table in Oracle SQL

```
CREATE TABLE <table name> (
      <column name> <data type> PRIMARY KEY,
      <column name> <data type>,
      …
      <column name> <data type>
);
```

Theory: Introduction to SQL

Example: Creating the `Tutor` table with a one column primary key

Create a table named `Tutor` with two columns: a `name` column to take strings up to length 20 and a `roomNo` column to take integer values. The name column values will be unique. Add a primary key.

```
CREATE TABLE Tutor (
      name VARCHAR(20) PRIMARY KEY,
      roomNo INT
);
```

In this example, it is possible to create a primary key that only consists of one column because the `name` column already contains a piece of data that is unique.

Although, this method of defining a primary key is non-standard and is seen as a "shortcut" method. It is good practice to define all constraints in the same manner, as seen in the section below.

Single column, composite or compound primary key

If a table is being created with a composite or compound primary key, it must be defined outside of the column comma separated row values. The keyword "`PRIMARY KEY`" is followed by a comma separated list of column names enclosed by brackets (`()`). This is also the standard way of defining all primary keys, including single column primary keys.

Format: Creating a table in Oracle SQL with any type of primary key

```
CREATE TABLE <table name> (
      <column name> <data type>,
      <column name> <data type>,
      …
      <column name> <data type>,
      PRIMARY KEY (<column name>, <column name>, …, <column name>)
);
```

Example: Creating the table `Lecture` in Oracle SQL

Create a table named `Lecture` with a `name` column to take strings up to length 10, a `roomNo` column to take integer values, a date column `occurs` and a `lecturer` column to take strings up to length 10. Only one lecture will occur in a given room on a given date. Add a primary key.

```
CREATE TABLE Lecture (
      name VARCHAR(10),
      roomNo INT,
      occurs DATE,
      lecturer VARCHAR(10),
      PRIMARY KEY (roomNo, occurs)
);
```

In this example, it is possible to create a primary key consisting of the `roomNo` column and the `occurs` column as these data together will provide a unique value. This is because it is given that only one lecture will occur in a given room on a given date.

Theory: Introduction to SQL

Constraints

Definition

A **constraint** is used to specify rules for data in a table. They define certain properties that the data in a database must comply with.

Naming constraints

Constraints can be named with a custom string defined by a database manager:

```
CONSTRAINT <constraint name> <constraint>
```

It is generally considered good practice to name constraints as, in the absence of a defined constraint name, the DBMS will automatically generate a name for the constraint. Automatically generated constraint names make decoding error messages difficult as they are often ambiguous. As a result, good naming of constraints allows errors to be more easily identified.

`DEFAULT` and `CHECK`

Definitions

A **`DEFAULT` constraint** sets the value for a cell should a value not be specified by the user. This helps to prevent NULL values.

A **`CHECK` constraint** limits the values that are accepted as valid for a cell.

Format

Format: Creating a table that has `DEFAULT` and `CHECK` constraints on a column

```
CREATE TABLE <table name> (
      <column name> <data type> DEFAULT <DATA>
      CONSTRAINT <constraint name> CHECK (<column name> IN (<data>, <data>, …, <data>))
      …
);
```

The `DEFAULT` constraint is set to a given value and the `CHECK` constraint is applied to the column and limits the accepted values for the cell to those present in the comma separated values enclosed by brackets (`()`).

Theory: Introduction to SQL

Example

Example: Creating `Person` table with `Gender` column

Create a table named `Person` with a `name` column to take strings up to length 10 and a `gender` column that can only have the values ‘`M`’, ‘`F`’, or ‘`U`’, and a default value of ‘`U`’.

```
CREATE TABLE Person (
      name VARCHAR(10),
      gender CHAR(1) DEFAULT 'U',
      CONSTRAINT <constraint name> CHECK (gender IN ('M', 'F', 'U')
      …
);
```

Using constraints for Boolean data

Whilst a Boolean data type does exist in some variations of SQL, it does not exist in all and has not always existed in some. As a result, in order to maintain backwards compatibility, it is often good practice to use a different and widely-adopted data type to represent Boolean data.

The integer data type can be used in conjunction with a `CHECK` constraint in order to simulate the functionality of a Boolean data type.

Example: Creating a table with an integer column that simulates a Boolean data type

```
CREATE TABLE Question (
      questionNo INT,
      answer INT,
      CONSTRAINT <constraint name> CHECK (answer IN ('1', '0')
      …
);
```

In this example:

- `1` represents True; and
- `0` represents false.

It is also possible to simulate the functionality of a Boolean data type using the character data type.

Example: Creating a table with an character column that simulates a Boolean data type

```
CREATE TABLE Question (
      questionNo INT,
      answer CHAR(1),
      CONSTRAINT <constraint name> CHECK (answer IN ('T', 'F')
      …
);
```

In this example:

- `T` represents True; and
- `F` represents false.

Theory: Introduction to SQL

Foreign keys

Definition

A **foreign key** is an inter-relational constraint in which values in one set of table columns can only occur if:

- they match a similar set of table columns in another table; or
- they are NULL values.

For example, if `B` references `A`, `A` must exist. If not, the DBMS will prevent the data from being entered in to the database.

Foreign keys are used to link tables together and create a relationship between the tables.

Format

A foreign key can be defined inside the column comma separated values:

```
<column name> <data type> <constraint name> REFERENCES <table name>
```

Format: Creating a foreign key constraint that references a single column

```
CREATE TABLE <table name> (
    …
    <column name> <data type> <constraint name> REFERENCES <table name>
    …
);
```

This will allow a column, given by `<column name>`, to act as a foreign key by linking to a column (usually a primary key) in another table, given by `<table name>`.

By default, this constraint will reference the column name in another table with the same column name as the foreign key.

Similar to defining a primary key, this method of defining a foreign key is non-standard and is seen as a “shortcut” method. It is good practice to define all constraints in the same manner, as seen in below.

In order to reference single or multiple column names (usually a single, composite or compound key), the constraint must be defined outside of the column comma separated values.

Format: Creating a foreign key constraint that references single or multiple columns

```
CREATE TABLE <table name> (
    …
    CONSTRAINT <constraint name> FOREIGN KEY (<column name>, <column name>, …, <column name>)
        REFERENCES <table name>(<column name>, <column name>, …, <column name>)
    …
);
```

Theory: Introduction to SQL

Example

Example: Tables in a university database

Create a table named `Tutor` with a `name` column to take strings up to length 10 and a `gender` column that can only have the values ‘`M`’, ‘`F`’, or ‘`U`’, and a default value of ‘`U`’.

Create a table named `Lecture` with a `name` column to take strings up to length 10 and a `gender` column that can only have the values ‘`M`’, ‘`F`’, or ‘`U`’, and a default value of ‘`U`’.

Create a table named `Student` with a `name` column to take strings up to length 10 and a `gender` column that can only have the values ‘`M`’, ‘`F`’, or ‘`U`’, and a default value of ‘`U`’.

```
CREATE TABLE Tutor (
      name VARCHAR(20),
      roomNo INT,
      PRIMARY KEY (name),
      CHECK (roomNo > 0)
);

CREATE TABLE Lecture (
      name VARCHAR(20),
      roomNo INT,
      occurs DATE,
      lecturer VARCHAR(20),
      PRIMARY KEY (roomNo, occurs),
      CHECK (roomNo > 0)
);

CREATE TABLE Student (
      studentNo INT,
      studentName VARCHAR(20),
      roomNo INT,
      occurs DATE,
      PRIMARY KEY (studentNo),
      name CHAR(20) CONSTRAINT stName
            REFERENCES Tutor,
      CONSTRAINT stLec
            FOREIGN KEY (roomNo, occurs)
            REFERENCES Lecture(roomNo, occurs)
);
```

Theory: Introduction to SQL

Referential action

If a row with a foreign key is updated or deleted, the referential constraints can prevent the change from happening.

There are four response modes that can be given to the foreign key.

Response Mode	Syntax	Description
CASCADE	CONSTRAINT <constraint name> FOREIGN KEY (<column(s)>) REFERENCES <table name> (<column name>, …, <column name>) ON DELETE CASCADE	When referenced data in the parent table is deleted, all rows in the child table that depend on those values in the parent table have are also deleted.
SET NULL	CONSTRAINT <constraint name> FOREIGN KEY (<column(s)>) REFERENCES <table name> (<column name>, …, <column name>) ON DELETE SET NULL	When referenced data in the parent table is deleted, all rows in the child table that depend on those values in the parent table have their foreign keys set to null.
SET DEFAULT	CONSTRAINT <constraint name> FOREIGN KEY (<column(s)>) REFERENCES <table name>(<column name>, …, <column name>) ON DELETE SET DEFAULT	??
No action	CONSTRAINT <constraint name> FOREIGN KEY ((<column(s)>) REFERENCES <table name>(<column name>, …, <column name>)	No action will occur.

Theory: Introduction to SQL

Oracle administration

System views and useful columns

These include:

- `USER_TABLES (TABLE_NAME, NUM_ROWS*, ….)`
- `USER_TAB_COLUMNS (TABLE_NAME, COLUMN_NAME, DATA_TYPE, …)`
- `USER_CONSTRAINTS (TABLE_NAME, CONSTRAINT_NAME, CONSTRAINT_TYPE, INDEX_NAME, …)`
- `USER_INDEXES (TABLE_NAME, INDEX_NAME, INDEX_TYPE, NUM_ROWS, …)`

There are other views for all database objects.

Practical: Introduction to SQL

Creating tables

Example: Creating tables

Create the following four tables:

- **Customer (<u>customerName</u>, street, customerCity)**
- **Deposit (customerName, branchName, <u>accountNumber</u>, balance)**
- **Loan (customerName, branchName, <u>loanNumber</u>, amount)**
- **Branch (<u>branchName</u>, branchCity, Assets)**

In the Deposit table:

- **branchName is a foreign key to the Branch table; and**
- **customerName is a foreign key to the Customer table.**

In the Loan table:

- **branchName is a foreign key to the Branch table;**
- **customerName is a foreign key to the Customer table.**

Please note:

- **the primary keys are underlined;**
- **the foreign keys should be written so that if the primary key row is deleted, then the corresponding foreign key row is also deleted;**
- **the foreign keys should be given constraint names;**
- **accountNumber and loanNumber should be integers;**
- **balance, amount and assets should be decimals with two decimal places and a default value of zero; and**
- **suitable SQL data types should be chosen for the other columns.**

Statement	Output

`CREATE TABLE Branch (`
`branchName VARCHAR(20),`
`Assets DECIMAL(10,2) DEFAULT 0.00,`
`branchCity VARCHAR(20),`
`PRIMARY KEY (branchName)`
`);`
	Table created.

`CREATE TABLE Customer (`
`customerName VARCHAR(20),`
`street VARCHAR(20),`
`customerCity VARCHAR(20),`
`PRIMARY KEY (customerName)`
`);`
	Table created.

`CREATE TABLE Deposit (`
`customerName VARCHAR(20),`
`branchName VARCHAR(20),`
`accountNumber INTEGER PRIMARY KEY,`
`balance DECIMAL(10,2) DEFAULT 0.00,`
`CONSTRAINT DepToCust FOREIGN KEY (customerName)`
`REFERENCES Customer(CustomerName)ON DELETE CASCADE,`
`CONSTRAINT DepToBr FOREIGN KEY (branchName)`
`REFERENCES Branch(branchName) ON DELETE CASCADE`
`);`
	Table created.

`CREATE TABLE Loan (`
`customerName VARCHAR(20),`
`branchName VARCHAR(20),`
`loanNumber INTEGER PRIMARY KEY,`
`amount DECIMAL(10,2) DEFAULT 0.00,`
`CONSTRAINT BorToCust FOREIGN KEY (customerName)`
`REFERENCES Customer(customerName) ON DELETE CASCADE,`
`CONSTRAINT BorToBr FOREIGN KEY (branchName)`
`REFERENCES Branch(branchName) ON DELETE CASCADE`
`);`
	Table created.

Practical: Introduction to SQL

Describing tables

Example: Describing tables

Use the DESCRIBE (or DESC) command to perform a simple check of the table columns created.

Statement	Output

`DESC Branch;`	<pre>Name Null? Type
	--------------------- -------- ----------------------
	BRANCHNAME NOT NULL VARCHAR2(20)
	ASSETS NUMBER(10,2)
	BRANCHCITY VARCHAR2(20)</pre>

`DESC Customer;`	<pre>Name Null? Type
	--------------------- -------- ----------------------
	CUSTOMERNAME NOT NULL VARCHAR2(20)
	STREET VARCHAR2(20)
	CUSTOMERCITY VARCHAR2(20)</pre>

`DESC Deposit;`	<pre>Name Null? Type
	--------------------- -------- ----------------------
	CUSTOMERNAME VARCHAR2(20)
	BRANCHNAME VARCHAR2(20)
	ACCOUNTNUMBER NOT NULL NUMBER(38)
	BALANCE NUMBER(10, 2)</pre>

`DESC Loan;`	<pre>Name Null? Type
	--------------------- -------- ----------------------
	CUSTOMERNAME VARCHAR2(20)
	BRANCHNAME VARCHAR2(20)
	LOANNUMBER NOT NULL NUMBER(38)
	AMOUNT NUMBER(10,2)</pre>

Practical: Introduction to SQL

System tables

Definitions

System tables are implemented by the DBMS to store information about the database.

Oracle SQL includes the system tables `USER_TABLES` and `USER_TAB_COLUMNS`:

- **`USER_TABLES`** describes the relational tables owned by the current user – Its columns (except for `OWNER`) are the same as those in `ALL_TABLES`. To gather statistics for this view, use the `ANALYZE SQL` statement.
- **`USER_TAB_COLUMNS`** describes the columns of the tables, views, and clusters owned by the current user. Its columns (except for `OWNER`) are the same as those in "`ALL_TAB_COLUMNS`". To gather statistics for this view, use the `ANALYZE SQL` statement.

Practical: Introduction to SQL

Inserting data

Example: Inserting data

Using `INSERT` statements, add the following rows of data to the database.

Branch		
branchName	**Assets**	**branchCity**
Yorkshire	10000	Nottingham
Midlands	20000	Nottingham
RoyalBank	25000	Nottingham
HFE	15000	Derby
Southern	30000	Derby

Customer		
customerName	**street**	**customerCity**
Jones	Victoria	Nottingham
Patel	Church	Nottingham
Smith	Derby	Leicester
Ahmed	Church	Derby
Braun	Alfred	Derby
Chan	Victoria	Nottingham

Deposit			
customerName	**branchName**	**accountNumber**	**balance**
Jones	Yorkshite	1	100
Braun	Midlands	20	150
Ahmed	RoyalBank	30	480
Smith	Midlands	21	600
Patel	RoyalBank	31	450
Patel	Midlands	22	70
Braun	Southern	41	2000
Jones	HFE	42	4100

Loan			
customerName	**branchName**	**accountNumber**	**balance**
Jones	Yorkshire	11	3000
Chan	Yorkshire	12	2500
Ahmed	Yorkshire	13	1800
Smith	Midlands	50	5000
Smith	RoyalBank	6	500
Patel	Midlands	51	1000
Jones	Midlands	61	2000

Statement	Output

`INSERT INTO Branch VALUES ('Yorkshire', 10000, 'Nottingham');`
`INSERT INTO Branch VALUES ('Midlands', 20000, 'Nottingham');`
`INSERT INTO Branch VALUES ('RoyalBank', 25000, 'Nottingham');`
`INSERT INTO Branch VALUES ('HFE', 15000, 'Derby');`
`INSERT INTO Branch VALUES ('Southern', 30000, 'Derby');`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`

`INSERT INTO Customer VALUES ('Jones', 'Victoria', 'Nottingham');`
`INSERT INTO Customer VALUES ('Patel', 'Church', 'Nottingham');`
`INSERT INTO Customer VALUES ('Smith', 'Derby', 'Leicester');`
`INSERT INTO Customer VALUES ('Ahmed', 'Church', 'Derby');`
`INSERT INTO Customer VALUES ('Braun', 'Alfred', 'Derby');`
`INSERT INTO Customer VALUES ('Chan', 'Victoria', 'Nottingham');`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`

`INSERT INTO Deposit VALUES ('Jones', 'Yorkshire', 1, 100);`
`INSERT INTO Deposit VALUES ('Braun', 'Midlands', 20, 150);`
`INSERT INTO Deposit VALUES ('Ahmed', 'RoyalBank', 30, 480);`
`INSERT INTO Deposit VALUES ('Smith', 'Midlands', 21, 600);`
`INSERT INTO Deposit VALUES ('Patel', 'RoyalBank', 31, 450);`
`INSERT INTO Deposit VALUES ('Patel', 'Midlands', 22, 70);`
`INSERT INTO Deposit VALUES ('Braun', 'Southern', 41, 2000);`
`INSERT INTO Deposit VALUES ('Jones', 'HFE', 42, 4100);`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`

`INSERT INTO Loan VALUES ('Jones', 'Yorkshire', 11, 3000);`
`INSERT INTO Loan VALUES ('Chan', 'Yorkshire', 12, 2500);`
`INSERT INTO Loan VALUES ('Ahmed', 'Yorkshire', 13, 1800);`
`INSERT INTO Loan VALUES ('Smith', 'Midlands', 50, 5000);`
`INSERT INTO Loan VALUES ('Smith', 'RoyalBank', 6, 500);`
`INSERT INTO Loan VALUES ('Patel', 'Midlands', 51, 1000);`
`INSERT INTO Loan VALUES ('Jones', 'Midlands', 61, 2000);`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`
	`1 row created.`

Practical: Introduction to SQL

Example: Checking inserted data

Using `SELECT` statements, verify the validity of the entered data in each of the tables `Branch`, `Customer`, `Deposit` and `Loan`.

Statement	Output

`SELECT * FROM Branch;`	<pre>BRANCHNAME ASSETS BRANCHCITY
	-------------------- ---------- --------------------
	Yorkshire 10000 Nottingham
	Midlands 20000 Nottingham
	RoyalBank 25000 Nottingham
	HFE 15000 Derby
	Southern 30000 Derby
	
	5 rows selected.</pre>

`SELECT * FROM Customer;`	<pre>CUSTOMERNAME STREET CUSTOMERCITY
	-------------------- -------------------- --------------------
	Jones Victoria Nottingham
	Patel Church Nottingham
	Smith Derby Leicester
	Ahmed Church Derby
	Braun Alfred Derby
	Chan Victoria Nottingham
	
	6 rows selected.</pre>

`SELECT * FROM Deposit;`	<pre>CUSTOMERNAME BRANCHNAME ACCOUNTNUMBER BALANCE
	-------------------- -------------------- ------------- ----------
	Jones Yorkshire 1 100
	Braun Midlands 20 150
	Ahmed RoyalBank 30 480
	Smith Midlands 21 600
	Patel RoyalBank 31 450
	Patel Midlands 22 70
	Braun Southern 41 2000
	Jones HFE 42 4100
	
	8 rows selected.</pre>

`SELECT * FROM Loan;`	<pre>CUSTOMERNAME BRANCHNAME LOANNUMBER AMOUNT
	-------------------- -------------------- ---------- ----------
	Jones Yorkshire 11 3000
	Chan Yorkshire 12 2500
	Ahmed Yorkshire 13 1800
	Smith Midlands 50 5000
	Smith RoyalBank 6 500
	Patel Midlands 51 1000
	Jones Midlands 61 2000
	
	7 rows selected.</pre>

Theory: Joins and Functions

SELECT statements

Definition

The **SELECT statement** is used to select data from a database. The data returned is stored in a result table, called the result-set, and can contain rows from one or more tables.

Syntax

Format: SELECT statement

```
SELECT <column name>, <column name>, …, <column name>
FROM <table name>
WHERE <condition>;
```

For a `SELECT` statement to be valid, it must contain:

- one or more column names that will be returned in the result-set;
- a table name that refers to a table that contains all of the columns specified in other parts of the statement.

A `SELECT` statement may contain a `WHERE` clause that specifies the conditions that must be met by a row for it to be included in the result-set.

Example: Selecting data from the table `Customer`

Write a statement to find the street address (`street`) and city (`customerCity`) of all of the customers in the Customer table whose name is Smith.

```
SELECT street, customerCity
FROM Customer
WHERE customerName = 'Smith';
```

Theory: Joins and Functions

SELECT DISTINCT statements

Definition

The **SELECT DISTINCT statement** is used to select distinct data from a database. The data returned is stored in a result table, called the result-set, and can contain rows from one or more tables, but can only contain values that are different as the values must be distinct.

Syntax

Format: SELECT DISTINCT statement

```
SELECT DISTINCT <column name>, <column name>, …, <column name>
FROM <table name>
WHERE <condition>;
```

For a SELECT DISTINCT statement to be valid, it must contain:

- one or more column names that will be returned in the result-set;
- a table name that refers to a table that contains all of the columns specified in other parts of the statement.

A SELECT DISTINCT statement may contain a WHERE clause that specifies the conditions that must be met by a row for it to be included in the result-set.

Example: Selecting data from the table Deposit

Write a statement to find the branch names (branchName) for each deposit account (Deposit) of the customers Smith and Chen that have a balance greater than 200 or less than 100.

```
SELECT DISTINCT branchName
FROM DEPOSIT
WHERE
(customerName = 'Smith' OR customerName = 'Chen')
AND
(balance > 200 OR balance < 100);
```

Brackets are required around the clauses in order to ensure that the conditional operations (AND and OR) are completed in the correct order. According to the order of precedence for conditional operators, an AND operator takes precedence, and therefore would be completed first, over an OR operator. However, in this example, the intent is for the OR operator to be completed first before the AND operator and therefore brackets as operations within brackets take precedence over all other operations.

In this example, A SELECT DISTINCT statement is used rather than a normal SELECT statement to prevent the branch names being repeated. It is likely that there are multiple rows that meet the conditions of having the customer name Smith or Chen and have a balance that is greater than 200 or less than 100. This means that without using the SELECT DISTINCT statement, multiple rows that contain the same branch name would be returned and therefore the result-set would contain redundancy.

Theory: Joins and Functions

Set operations

Definitions

The **UNION operator** is used to return a result-set with all distinct rows selected by either SELECT query.

The **MINUS operator** is used to return a result-set with all distinct rows selected by the first SELECT query but not the second.

The **INTERSECT operator** is used to a result-set with return all distinct rows selected by both SELECT queries.

Syntax

UNION operator

Format: SELECT statement using a UNION operator

```
( SELECT <column name>, <column name>, …, <column name>
FROM <table name>
WHERE <condition> )
UNION
( SELECT <column name>, <column name>, …, <column name>
FROM <table name>
WHERE <condition> );
```

MINUS operator

Format: SELECT statement using a MINUS operator

```
( SELECT <column name>, <column name>, …, <column name>
FROM <table name>
WHERE <condition> )
MINUS
( SELECT <column name>, <column name>, …, <column name>
FROM <table name>
WHERE <condition> );
```

INTERSECT operator

Format: SELECT statement using an INTERSECT operator

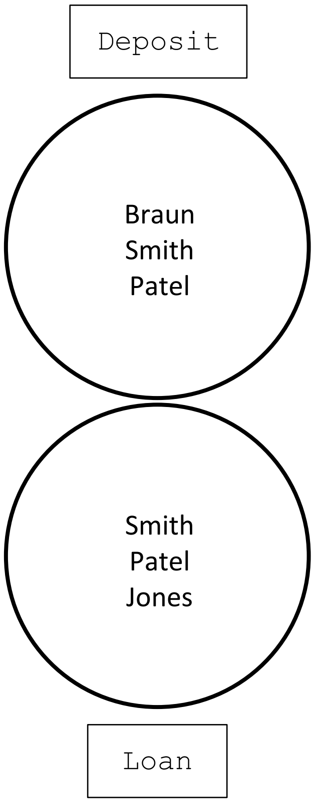
```
( SELECT <column name>, <column name>, …, <column name>
FROM <table name>
WHERE <condition> )
INTERSECT
( SELECT <column name>, <column name>, …, <column name>
FROM <table name>
WHERE <condition> );
```

Theory: Joins and Functions

Examples

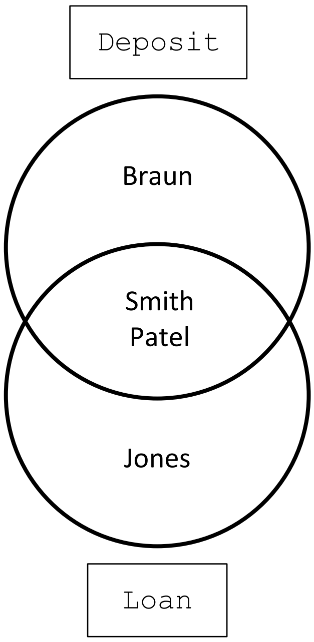
It is given that a database contains two tables, `Deposit` and `Loan`. The names in the `Deposit` table that are at the Midlands branch are Braun, Smith and Patel, while the names in the `Loan` table that are at the midlands branch are Smith, Patel and Jones.

Example: `Deposit` and `Loan` tables



Example: Union of `Deposit` and `Loan` tables

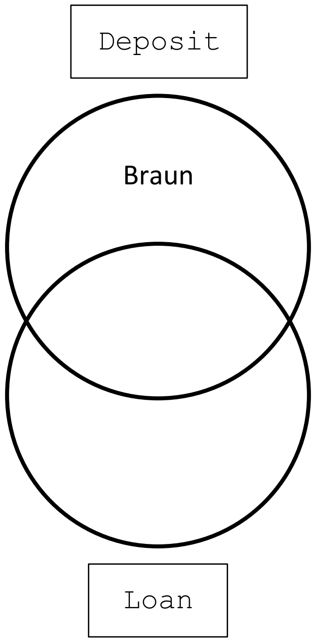
```
( SELECT customerName
FROM Deposit
WHERE branchName = 'Midlands' )
UNION
( SELECT customerName
FROM Loan
WHERE branchName = 'Midlands' );
```



Theory: Joins and Functions

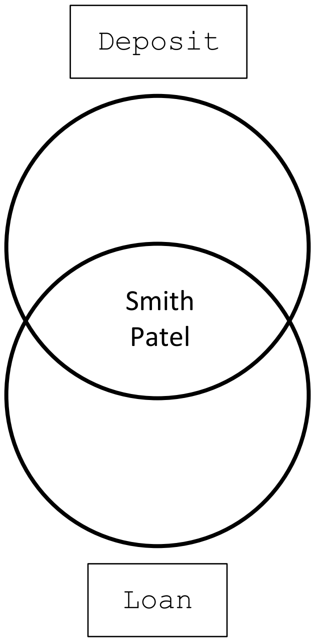
Example: Minus of `Deposit` and `Loan` tables

```
( SELECT customerName
FROM Deposit
WHERE branchName = 'Midlands' )
MINUS
( SELECT customerName
FROM Loan
WHERE branchName = 'Midlands' );
```



Example: Intersect of `Deposit` and `Loan` tables

```
( SELECT customerName
FROM Deposit
WHERE branchName = 'Midlands' )
INTERSECT
( SELECT customerName
FROM Loan
WHERE branchName = 'Midlands' );
```



Theory: Joins and Functions

Inner joins

Definition

An **inner join** selects rows that have matching values in both tables.

Syntax

Format: `SELECT` statement using an inner join

```
SELECT <column name>, <column name>, …, <column name>
FROM <table name> JOIN <table name>
ON <table name>.<column name> = <table name>.<column name>
WHERE <optional condition(s)>;
```

For the `SELECT` statement to be valid when using an `INNER JOIN`, the column names in the `ON` clause must be identical as this is the column on which the tables are joined.

Inner joins can also be expressed using alternative syntax.

Format: Alternative `SELECT` statement using an inner join

```
SELECT <column name>, <column name>, …, <column name>
FROM <table name>, <table name>
WHERE <table name>.<column name> = <table name>.<column name>
AND <optional condition(s)>;
```

When using this alternative syntax, the column names in the first `WHERE` clause must be identical as this is the column on which the tables are joined.

Both syntax approaches will yield valid results, however, it is best practice to use the first syntax option. This is because this syntax is a more standardised approach and is similar to the accepted syntax for different types of joins that will be explored later.

It is important to note that, when using either syntax, omission of the optional condition(s) will result in all of the rows from both tables being part of the result-set.

Example

Example: `SELECT` statement using an `INNER JOIN` keyword

What are the customer loan and account numbers for the customers in the Midlands branch?

```
SELECT loanNumber, accountnumber
FROM Loan JOIN Deposit
ON Loan.customerName = Deposit.customerName
WHERE Loan.branchName = ‘Midlands’
AND Deposit.branchName = ‘Midlands’;
```

```
SELECT loanNumber, accountNumber
FROM Loan, Deposit
WHERE Loan.customerName = Branch.customerName
AND Loan.branchName = ‘Midlands’
AND Deposit.branchName = ‘Midlands’;
```

Theory: Joins and Functions

Outer joins

Definitions

An **outer join** preserves unmatched rows from one or both tables:

A **left outer join** preserves unmatched rows from the left table.

A **right outer join** preserves unmatched rows from the right table.

A **full outer join** preserves unmatched rows from both tables.

Syntax

Left outer join

Format: `SELECT` statement using a left outer join

```
SELECT <column name>, <column name>, …, <column name>
FROM <table name> LEFT JOIN <table name>
ON <table name>.<column name> = <table name>.<column name>
WHERE <optional condition(s)>;
```

For the `SELECT` statement to be valid when using a `LEFT JOIN`, the column names in the `ON` clause must be identical as this is the column on which the tables are joined.

Right outer join

Format: `SELECT` statement using a right outer join

```
SELECT <column name>, <column name>, …, <column name>
FROM <table name> RIGHT JOIN <table name>
ON <table name>.<column name> = <table name>.<column name>
WHERE <optional condition(s)>;
```

Full outer join

Format: `SELECT` statement using a full outer join

```
SELECT <column name>, <column name>, …, <column name>
FROM <table name> FULL JOIN <table name>
ON <table name>.<column name> = <table name>.<column name>
WHERE <optional condition(s)>;
```

Theory: Joins and Functions

Examples

It is given that a database contains two tables, `Loan` and `Deposit`. The `Loan` table contains the customers Smith and Jones and the `Deposit` table contains the customers Smith and Braun.

Data: `Loan` and `Deposit` tables

Loan	
customerName	**loanNumber**
Smith	50
Jones	61

Deposit	
customerName	**accountNumber**
Smith	21
Braun	20

Left outer join

Example: Left outer join of `Loan` and `Deposit` tables

```
SELECT loanNumber, accountNumber
FROM Loan LEFT JOIN Deposit
ON Loan.customerName = Deposit.customerName
WHERE Loan.branchName = 'Midlands'
AND Deposit.branchName = 'Midlands';
```

This statement will preserve the unmatched rows from the `Loan` table but not preserve the unmatched rows from the `Deposit` table.

Loan	
customerName	**loanNumber**
Smith	50
Jones	61

Deposit	
customerName	**accountNumber**
Smith	21
Braun	20

Therefore, the result will be as follows:

Loan.customerName	Loan.loanNumber	Deposit.customerName	Deposit.accountNumber
Smith	50	Smith	21
Jones	61	NULL	NULL

Theory: Joins and Functions

Right outer join

Example: Right outer join of `Loan` and `Deposit` tables

```
SELECT loanNumber, accountNumber
FROM Loan RIGHT JOIN Deposit
ON Loan.customerName = Deposit.customerName
WHERE Loan.branchName = 'Midlands'
AND Deposit.branchName = 'Midlands';
```

This statement will preserve the unmatched rows from the `Deposit` table but not preserve the unmatched rows from the `Loan` table.

Loan	
customerName	**loanNumber**
Smith	50
Jones	61

Deposit	
customerName	**accountNumber**
Smith	21
Braun	20

Therefore, the result will be as follows:

Loan.customerName	Loan.loanNumber	Deposit.customerName	Deposit.accountNumber
Smith	50	Smith	21
NULL	NULL	Braun	20

Full outer join

Example: Full outer join of `Loan` and `Deposit` tables

```
SELECT loanNumber, accountNumber
FROM Loan FULL JOIN Deposit
ON Loan.customerName = Deposit.customerName
WHERE Loan.branchName = 'Midlands'
AND Deposit.branchName = 'Midlands';
```

This statement will preserve the unmatched rows from both the `Deposit` table and the `Loan` table.

Loan	
customerName	**loanNumber**
Smith	50
Jones	61

Deposit	
customerName	**accountNumber**
Smith	21
Braun	20

Therefore, the result will be as follows:

Loan.customerName	Loan.loanNumber	Deposit.customerName	Deposit.accountNumber
Smith	50	Smith	21
NULL	NULL	Braun	20
Jones	61	NULL	NULL

Theory: Joins and Functions

Single table self-join

Definition

A **single table self-join** is a query in which a table is joined to itself to allow values in a column to be compared with values in another column in the same table.

How it works

The single table self-join allows columns from the same table to be joined by creating a "copy" of the table.

Format: `SELECT` statement using a single table self-join

```
SELECT <alias>.<table name>, <alias>.<table name>
FROM <table name> <alias>, <table name> <alias>  // the same table name in both
WHERE <alias>.<column name> = <alias>.<column name>
AND <optional condition(s)>;
```

Example

Example: `SELECT` statement using a single table self-join

Find all the customers who have an account at some branch at which `Patel` has an account (also output the branch names).

```
SELECT T.customerName, T.branchName
FROM Deposit S, Deposit T
WHERE S.branchName = T.branchName
AND S.customerName = 'Patel';
```

This statement will preserve the match rows where the `branchName` is either `Midlands` or `RoyalBank` as these are branches at which `Patel` has an account, including those rows where `Patel` is the customer.

Deposit

`customerName`	`branchName`
Jones	Yorkshire
Braun	Midlands
Ahmed	RoyalBank
Smith	Midlands
Patel	RoyalBank
Patel	Midlands

Deposit

`customerName`	`branchName`
Jones	Yorkshire
Braun	Midlands
Ahmed	RoyalBank
Smith	Midlands
Patel	RoyalBank
Patel	Midlands

Therefore, the result will be as follows:

`Deposit.customerName`	`Deposit.branchName`
Braun	Midlands
Ahmed	RoyalBank
Smith	Midlands
Patel	RoyalBank
Patel	Midlands

Theory: Joins and Functions

Built-in functions

Definition

Built-in functions are functions that are available from the DBMS. They operate on the set of values in a column of a relation and return a single value.

Available functions

The available built-in functions include:

- SUM(<column name>) – sum of values;
- AVG(<column name>) – average of values;
- MIN(<column name>) – minimum value;
- MAX(<column name>) – maximum value;
- COUNT(<column name>) – number of values, excluding NULL values;
- COUNT(*) – number of values, including NULL values;
- TO_CHAR(<date column name>), 'DD-MON-YYYY HH24:MI:SS) – for date selection; and
- SYSDATE() – returns current date and time.

Apart from the COUNT functions, all functions eliminate NULL values first.

Examples

Example: SUM function

Find the total sum of the balances from all deposit accounts for branch name 'RoyalBank'.

```
SELECT SUM(balance)
FROM Deposit
WHERE branchName = 'RoyalBank';
```

Example: COUNT function

Find the number of rows in the Customer table.

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

Practical: Joins and Functions

Set operations

Example: `UNION` operator

Write a SQL statements to find the names of all the customers who have a loan, or a deposit, or both a loan and deposit at the 'Midlands' branch.

Statement	Output
(SELECT CustomerName FROM Deposit WHERE BranchName = 'Midlands') UNION (SELECT CustomerName FROM Loan WHERE BranchName = 'Midlands');	CUSTOMERNAME -------------------- Braun Jones Patel Smith

Practical: Joins and Functions

Joins

Example: INNER JOIN

Write a SQL statement to retrieve the customer name and branch name for all customers who have a loan (i.e. appear in the loan table) and who have the customer city 'Nottingham'.

Statement	Output

`SELECT branchName, Customer.customerName`
`FROM Customer JOIN Loan`
`ON Customer.customerName = Loan.customerName`
`WHERE customerCity = 'Nottingham';`
	`BRANCHNAME CUSTOMERNAME`
	`-------------------- --------------------`
	`Yorkshire Jones`
	`Yorkshire Chan`
	`Midlands Patel`
	`Midlands Jones`

`SELECT branchName, Customer.customerName`
`FROM Customer, Loan`
`WHERE CustomerCity = 'Nottingham'`
`AND Customer.customerName = Loan.customerName;`
	`BRANCHNAME CUSTOMERNAME`
	`-------------------- --------------------`
	`Yorkshire Jones`
	`Yorkshire Chan`
	`Midlands Patel`
	`Midlands Jones`

Example: LEFT JOIN

Write a SQL statement using an outer join to retrieve all the customer names from the loan table who have a loan with only the 'Yorkshire' branch, together with their amount loaned and any balance they have in the deposit table. Each customer can have a deposit at any branch in the deposit table.

Statement	Output

`SELECT Loan.customerName, amount, balance`
`FROM Loan LEFT JOIN Deposit`
`ON Deposit.customerName = Loan.customerName`
`WHERE Loan.branchName = 'Yorkshire';`
	`CUSTOMERNAME AMOUNT BALANCE`
	`-------------------- ---------- ----------`
	`Jones 3000 100`
	`Ahmed 1800 480`
	`Jones 3000 4100`
	`Chan 2500`

Example: Single table self-join

Write a SQL statement using an inner join to find the names of all customers who have a loan at the same branch at which 'Smith' has a loan.

Statement	Output

`SELECT t.CustomerName`
`FROM Loan t, Loan s`
`WHERE s.CustomerName = 'Smith'`
`AND s.BranchName = t.BranchName`
`AND t.CustomerName <> 'Smith'; // <> means is not equal to`
	`CUSTOMERNAME`
	`--------------------`
	`Patel`
	`Jones`

Practical: Joins and Functions

Built-in functions

Example: MAX, MIN and AVG

Write a SQL statement using functions to find the minimum, maximum, average balance of deposits in the Midlands branch.

Statement	Output

`SELECT MAX(balance), MIN(balance), AVG(balance)`
`FROM Deposit`
`WHERE branchName='Midlands';`
	`MAX(BALANCE)  MIN(BALANCE)  AVG(BALANCE)`
	`------------  ------------  ------------`
	`600           70            273.333333`

Theory: Subqueries and Views

Subqueries using `IN` and `NOT IN` clauses

Definition

A **subquery** is a query (SELECT) that appears in the WHERE clause of another SQL statement. This is possible as a SELECT statement can return a table of values that can be used in another query.

The `IN` clause

The **IN clause** returns TRUE if the provided value is found in a set of values.

Format: Using `IN` clause on a distinct set of values

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <condition>
      AND <column name> IN (<value>, <value>, …, <value>)
```

Example: Using `IN` clause on a distinct set of values

Write a SQL statement to find all of the customer names who have both a loan at the `Midlands` branch and are `Patel`, `Smith` or `Braun`.

Statement	Output

`SELECT customerName`
`FROM Loan`
`WHERE branchName = 'Midlands'`
`AND customerName IN ('Patel', 'Smith', 'Braun');`
	`CUSTOMERNAME`
	`--------------------`
	`Smith`
	`Patel`

In this example, each row found from the Loan table where the branchName is Midlands will only be output if the customer name is in the set of values.

However, it is not good practice to "hard code" values in SELECT queries if those values are likely to change. Instead, a subquery may use the IN clause where a SELECT statement specifies that a field must be present in the results from the subquery. The IN clause returns TRUE if the provided value is found in the values returned by the subquery.

In this case, it may be appropriate to use IN rather than EQUALS because the results from the first query may not match the results from the subquery exactly. Instead, a result from the first query must exist in a set of results from the subquery.

Format: Subquery using the `IN` clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> IN
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)> );
```

The subquery restricts the output of the query to those results that also appear in the output of the subquery.

Theory: Subqueries and Views

Example: Subquery using the `IN` clause

Write a SQL statement to find all of the customer names who have both a loan and deposit at the `Midlands` branch.

Statement

```
SELECT customerName
      FROM Loan
      WHERE branchName = 'Midlands'
      AND customerName IN
      ( SELECT customerName
        FROM Deposit
        WHERE branchName = 'Midlands' );
```

Output

```
CUSTOMERNAME
--------------------
Smith
Patel
```

In this example, there are three results from the first query returned from the `Loan` table and three results from the subquery returned from the `Deposit` table.

Loan
customerName
Smith
Patel
Jones

Deposit
customerName
Braun
Smith
Patel

The subquery restricts the output of the query to those customers who also appear in the `Deposit` table where the `branchName` is `Midlands`. This means that only `Smith` and `Patel` are output as these are the only rows whose `customerName` appears in both query results.

The `NOT IN` clause

The **`NOT IN` clause** returns `TRUE` if the provided value is not found in a set of values.

This clause works in a similar way to the `IN` clause and, when using subqueries, it may be appropriate to use `NOT IN` rather than `NOT EQUAL`. This is because the results from the first query may not completely differ from the results from the subquery exactly. Instead, a result from the first query must not exist in a set of results from the subquery.

Format: Subquery using the `NOT IN` clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> NOT IN
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)> );
```

Example: Subquery using the `NOT IN` clause	
Write a SQL statement to find all of the customer names who do not have a deposit at the `Midlands` branch.	
Statement	**Output**

`SELECT customerName`
`FROM Customer`
`WHERE customerName NOT IN`
`( SELECT customerName`
`FROM Deposit`
`WHERE branchName = 'Midlands' );`
	`CUSTOMERNAME`
	`--------------------`
	`Chan`
	`Jones`
	`Ahmed`

In this example, there are sixth results from the first query returned from the `Customer` table and three results from the subquery returned from the `Deposit` table.

Customer
customerName
Jones
Patel
Smith
Ahmed
Braun
Chan

Deposit
customerName
Braun
Smith
Patel

The subquery restricts the output of the query to those customers who also appear in the `Deposit` table where the `branchName` is not `Midlands`. This means that only `Chan`, `Jones` and `Ahmed` are output as these are the only rows whose `customerName` appears in the first query but not the subquery.

Theory: Subqueries and Views

Expressing subqueries as joins

Example

It is possible to express a subquery as a join and achieve the same output.

Example: Subquery using the `IN` clause

Write a SQL statement using a join to find all of the customer names who have both a loan and deposit at the `Midlands` branch.

Statement	Output

SELECT Loan.customerName
    FROM Loan, Depsoit
    WHERE Loan.customerName = Deposit.customerName
    AND Loan.branchName = 'Midlands'
    AND Deposit.branchName = 'Midlands';
	CUSTOMERNAME
	-------------------- 	
	Smith
	Patel

In this example, the `Deposit` and `Loan` tables are joined using an `INNER JOIN` on `customerName`. This would give the same result as using the subquery shown on the previous page.

Loan	
customerName	**branchName**
Smith	Midlands
Patel	Midlands

Deposit	
customerName	**branchName**
Smith	Midlands
Patel	Midlands

The join matches rows with the same `customerName` and `branchName` that appear in both the `Loan` and `Deposit` table. This means that only `Smith` and `Patel` are output.

Theory: Subqueries and Views

Subqueries using the ANY and ALL clauses

The ANY clause

The **ANY clause** returns TRUE if the provided value is greater than one of the values in a set of values.

Format: Subquery using the ANY clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> ANY
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)> );
```

Example: Subquery using the ANY clause

Write a SQL statement to find all branches that have greater assets than at least one branch located in Nottingham.

Statement	Output

`SELECT branchName`
`FROM Branch`
`WHERE assets > ANY`
`( SELECT assets`
`FROM Branch`
`WHERE branchCity = 'Nottingham' );`
	BRANCHNAME
	-------------------- 	
	Midlands
	RoyalBank
	HFE
	Southern

The ALL clause

The **ALL clause** returns TRUE if the provided value is greater than all of the values in a set of values.

Format: Subquery using the ALL clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> ALL
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)> );
```

Example: Subquery using the ALL clause

Write a SQL statement to find all branches that have greater assets than all branches located in Nottingham.

Statement	Output

`SELECT branchName`
`FROM Branch`
`WHERE assets > ALL`
`( SELECT assets`
`FROM Branch`
`WHERE branchCity = 'Nottingham' );`
	BRANCHNAME
	-------------------- 	
	Southern

Theory: Subqueries and Views

Subqueries using the `EXISTS` and `NOT EXISTS` clauses

The `EXISTS` clause

The **`EXISTS` clause** returns `TRUE` if a value is present in a set of values.

Format: Subquery using the `EXISTS` clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> EXISTS
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)> );
```

Example: Subquery using the `EXISTS` clause

Write a SQL statement to find all customer names who have a deposit.

Statement

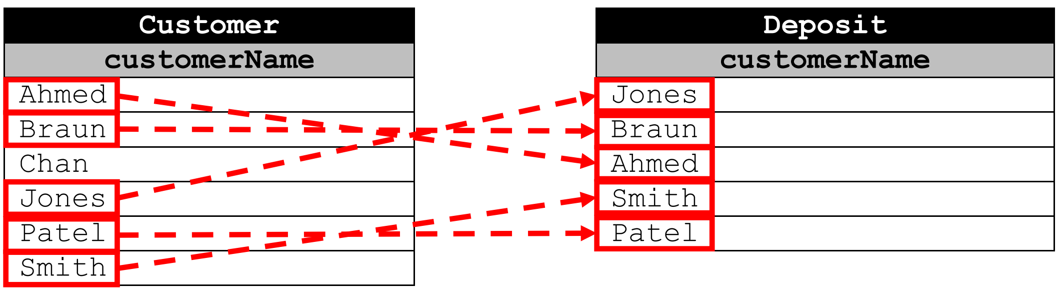
```
SELECT Customer.customerName
      FROM Customer
      WHERE EXISTS
            ( SELECT Deposit.customerName
              FROM Deposit
              WHERE Customer.customerName = Deposit.customerName);
```

Output

```
BRANCHNAME
--------------------
Ahmed
Braun
Jones
Patel
Smith
```

In this example, there are six results from the first query returned from the `Customer` table. In addition, there are five distinct values of `customerName` in the `Deposit` table (obtained by `SELECT customerName FROM Deposit`).

Each `customerName` from the `Customer` table is put in to the `WHERE` clause in the subquery as to restrict the output to the instances where `customerName` exists in both tables.



The subquery restricts the output of the query to those customers who also appear in the `Customer` table and the `Deposit` table. This means that only `Ahmed`, `Braun`, `Jones`, `Patel` and `Smith` are output as these are the only rows whose `customerName` appears in both query results.

As seen before, this could also be written using an `INNER JOIN`.

Theory: Subqueries and Views

The NOT EXISTS clause

The **NOT EXISTS clause** returns TRUE if a value is absent from a set of values.

Format: Subquery using the NOT EISTS clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> NOT EXISTS
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)> );
```

Example: Subquery using the NOT EXISTS clause

Write a SQL statement to find all customer names who do not have a deposit.

Statement	Output

SELECT Customer.customerName
      FROM Customer
      WHERE EXISTS
            (SELECT Deposit.customerName
              FROM Deposit
              WHERE Customer.customerName = Deposit.customerName);
	BRANCHNAME
	-------------------- 	
	Chan

In this example, there are six results from the first query returned from the Customer table. In addition, there are five distinct values of customerName in the Deposit table (obtained by SELECT customerName FROM Deposit).

Each customerName from the Customer table is put in to the WHERE clause in the subquery as to restrict the output to the instances where customerName does not exist in both tables.

Customer
customerName
Ahmed
Braun
Chan
Jones
Patel
Smith

Deposit
customerName
Jones
Braun
Ahmed
Smith
Patel

The subquery restricts the output of the query to those customers who also appear in the Customer table and the Deposit table. This means that only Chan is output as this is the only row whose customerName appears in the first query only.

Theory: Subqueries and Views

Operations using subqueries

Deletion

Format: Using a subquery to perform a `DELETE` statement

```
DELETE FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> <clause>
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)> );
```

Example: Using a subquery to perform a `DELETE` statement

Write a SQL statement to delete all deposits at branches with branch city in Derby.

Statement

```
DELETE FROM Deposit
      WHERE branchName IN
      ( SELECT branchName
        FROM Branch
        WHERE branchCity = 'Derby' );
```

This example shows that it is possible to delete the branches with branch city in Derby without prior knowledge of the branch names themselves. This means that time can be saved as there is no need to find the branch names first.

Update

Format: Using a subquery to perform an `UPDATE` statement

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> <clause>
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)> );
```

Example: Using a subquery to perform an `UPDATE` statement

Write a SQL statement to update the balance of all deposits to 100 whose branch has branch city in Nottingham.

Statement

```
UPDATE Deposit
      SET balanace = 100
      WHERE branchName IN
      ( SELECT branchName
        FROM Branch
        WHERE banchCity = 'Nottingham' );
```

This example shows that it is possible to update the deposits with whose branch has branch city in Nottingham without prior knowledge of which branches have branch city in Nottingham. This means that time can be saved as there is no need to find the branches with branch city in Nottingham first.

Theory: Subqueries and Views

Nested subqueries

Definition

A **nested subquery** is one that is comprised of multiple subqueries that depend on each other to perform the overall query.

Usage

Format: Nested subqueries

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      AND <column name> <clause>
            ( SELECT <column name>
              FROM <table name>
              WHERE <optional condition(s)>
              AND <column name> <clause>
              ( SELECT <column name>
                FROM <table name>
                WHERE <optional condition(s)> ));
```

Example: Nested subqueries

Write a SQL statement to find all the customer cities for any customer who has a deposit in a branch located in Nottingham.

Statement	Output

`SELECT customerCity`
`      FROM Customer`
`      WHERE customerName IN`
`      ( SELECT customerName`
`        FROM Deposit`
`        WHERE branchName IN`
`            ( SELECT branchName`
`              FROM Branch`
`              WHERE branchCity = 'Nottingham' ) );`
	`CUSTOMERCITY`
	`--------------------`
	`Nottingham`
	`Derby`
	`Derby`
	`Leicester`
	`Nottingham`

This could also be achieved using a three table join.

Theory: Subqueries and Views

The ORDER BY clause

Definition

The **ORDER BY clause** specifies that a SELECT statement returns a result set with the rows being sorted by the values of one or more columns.

Usage

Format: ORDER BY clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      ORDER BY
            <column name> <ASC/DESC>,
            <column name> <ASC/DESC>,
            …
            <column name> <ASC/DESC>;
```

The default is ascending and therefore ASC may be omitted and perform the same function.

Example: ORDER BY clause

Write a SQL statement to list the customer names and branch names for loans in descending order of `branchName` and ascending order of `customerName`.

Statement	Output

<pre>SELECT customerName, branchName
      FROM Loan
      ORDER BY
            branchName DESC,
            customerName;</pre>
	<pre>CUSTOMERNAME BRANCHNAME
	-------------------- --------------------
	Ahmed Yorkshire
	Chan Yorkshire
	Jones Yorkshire
	Smith RoyalBank
	Jones Midlands
	Patel Midlands
	Smith Midlands</pre>

Theory: Subqueries and Views

The GROUP BY clause

Definition

The **GROUP BY clause** is used to group the results of a SELECT statement based on one or more columns such that groups of rows are formed with the same column value.

Usage

Format: GROUP BY clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      GROUP BY
            <column name>,
            <column name>,
            …
            <column name>;
```

Example: GROUP BY clause

Write a SQL statement to find the branch names and average deposit balance for each branch.

Statement	Output
SELECT branchName, AVG(balance) FROM Deposit GROUP BY branchName;	BRANCHNAME AVG(BALANCE) -------------------- -------------------- Yorkshire 121.55 Midlands 273.333333 RoyalBank 465 HFE 4100 Southern 2000

In this example, GROUP BY allows each branchName to be grouped together.

branchName	balance
Yorkshire	121.55
Midlands	150
RoyalBank	480
Midlands	600
RoyalBank	450
Midlands	70
Southern	2000
HFE	4100

branchName	balance
HFE	4100
Midlands	150
Midlands	600
Midlands	70
RoyalBank	480
RoyalBank	450
Southern	2000
Yorkshire	121.55

Theory: Subqueries and Views

Subsequently, the average balance is calculated for each group of branches.

branchName	balance
HFE	4100
Midlands	150
Midlands	600
Midlands	70
RoyalBank	480
RoyalBank	450
Southern	2000
Yorkshire	121.55

branchName	AVG(balance)
Yorkshire	121.55
Midlands	273.333333
RoyalBank	465
HFE	4100
Southern	2000

This example shows that `GROUP BY` can be used to display aggregate values; the aggregate average deposit balance grouped by each branch name.

Multiple outputs

The example shown above contains a `SELECT` statement that outputs the grouped branch names and the average deposit balance for each of those groups. It is not possible to simply add another column name, such as the customer name, to output without special consideration first.

In a `SELECT` statement, it is not possible to mix row outputs and group outputs. As a result, in order to add another column to the output, it must also be added to the `GROUP BY` clause.

Example: Multiple `GROUP BY` clause

Write a SQL statement to find the branch names, customer names and average deposit balance for each branch.

Statement

```
SELECT branchName, customerName, AVG(balance)
      FROM Deposit
      GROUP BY
            branchName,
            customerName;
```

Theory: Subqueries and Views

The HAVING clause

Definition

The **HAVING clause** specifies that a SELECT statement should only return rows where aggregate values meet the specified conditions. It acts as a filter for groups, similar to how the WHERE clause acts as a filter for rows.

The HAVING clause can only reference grouping columns or other columns from the table list to which an aggregate can be applied. Predicates in the HAVING clause are applied AFTER the formation of groups that might use the WHERE clause.

Usage

Format: HAVING clause

```
SELECT <column name>, <column name>, …, <column name>
      FROM <table name>
      WHERE <optional condition(s)>
      GROUP BY
            <column name>,
            <column name>,
            …
            <column name>
      HAVING
            <condition(s)>;
```

Example: HAVING clause

Write a SQL statement to find the branch names of all branches where the average account balance is more than £200.

Statement	Output

`SELECT branchName, AVG(balance)`
`FROM Deposit`
`GROUP BY`
`branchName`
`HAVING`
`AVG(balance) > 200;`
	`BRANCHNAME            AVG(BALANCE)`
	`--------------------  --------------------`
	`Midlands              273.333333`
	`RoyalBank             465`
	`HFE                   4100`
	`Southern              2000`

Theory: Subqueries and Views

Views

Definition

A **view** is a virtual table derived from one or more tables or other views. A view is defined by a `SELECT` query expression.

Usage

Format: Creating a view

```
CREATE VIEW <view name> (<column name>, <column name>, …, <column name>) AS
      SELECT <column name>, <column name>, …, <column name>
      FROM <table name(s)>
      WHERE <optional conditions(s)
```

When creating a view, it is possible to give column names that better describe the column based on the data collected by the `SELECT` statement.

Example: Creating a view

Write a SQL statement to create a view showing branch finance by branch name.

Statement

```
CREATE VIEW Finance (branchName, branchAssets, branchTotalLoans, branchTotalDeposits) AS
      SELECT Branch.branchName, Branch.assets, SUM(Deposit.balance),
             SUM(Loan.amount)
      FROM Branch, Deposit, Loan
      WHERE Deposit.branchName = Branch.branchName
      AND Loan.branchName = Branch.branchName
      GROUP BY
            Branch.branchName,
            Branch.assets;
```

In this example, the column names `branchTotalLoans` and `branchTotalDeposits` give a better description of the data collected by `SUM(Deposit.balance)` and `SUM(Loan.amount)` respectively.

A view can be treated in a similar fashion to a table. As such, performing the statement `SELECT * FROM Finance` would return the table below.

```
BRANCHNAME  BRANCHASSETS  BRANCHTOTALLOANS  BRANCHTOTALDEPOSITS
----------  ------------  ----------------  -------------------
RoyalBank   25000         930               1000
Yorkshire   10000         364.65            7300
Midlands    20000         2460              24000
```

Theory: Subqueries and Views

Evaluation

The use of views can be beneficial as they:

- allow different and independent external representations of the database tables without modification to the tables; and
- provide independence to table and column additions or column re-definition.

Practical: Subqueries and Views

Subqueries and joins

Example: Subqueries

Write an SQL statement that uses a subquery to find the names of all the customers who have a deposit at a branch with a branch city 'Nottingham'.

Statement	Output

`SELECT customerName`
`FROM Deposit`
`WHERE branchName IN`
`( SELECT branchName`
`FROM Branch`
`WHERE branchCity = 'Nottingham' );`
	BRANCHNAME
	---------- 	
	Jones
	Braun
	Ahmed
	Smith
	Patel
	Patel

Example: Joins

Write an SQL statement that uses a join to find the names of all the customers who have a deposit at a branch with a branch city 'Nottingham'.

Statement	Output

`SELECT customerName`
`FROM Deposit JOIN Branch`
`ON Deposit.branchName = Branch.branchName`
`WHERE Branch.branchCity = 'Nottingham'`
 	
OR
 	
`SELECT customerName`
`FROM Deposit D JOIN Branch B`
`ON D.branchName = B.branchName`
`WHERE B.branchCity = 'Nottingham'`
 	
OR
 	
`SELECT customerName`
`FROM Deposit, Branch`
`WHERE Deposit.branchName = Branch.branchName`
`AND Branch.branchCity = 'Nottingham';`
 	
OR
 	
`SELECT customerName`
`FROM Deposit D, Branch B`
`WHERE D.branchName = B.branchName`
`AND B.branchCity = 'Nottingham';`
	BRANCHNAME
	---------- 	
	Jones
	Braun
	Ahmed
	Smith
	Patel
	Patel

Practical: Subqueries and Views

Clauses

Example: GROUP BY clause

Write a single SQL statement to list each branch name together with the sum of all balances from deposits with that branch name (i.e. group the output by branch name).

Statement	Output

`SELECT branchName, SUM(balance)`
`FROM Deposit`
`GROUP BY`
`branchName;`
	`BRANCHNAME   SUM(BALANCE)`
	`----------   ------------`
	`Yorkshire    122.55`
	`Midlands     820`
	`RoyalBank    930`
	`HFE          4100`
	`Southern     2000`

Example: HAVING clause

Write a single SQL statement to list each branch name together with the number of customer deposits with that branch name, where the number of customer deposits with that branch name is more than 1.

Statement	Output

`SELECT branchName, COUNT(*) AS NoOfDeposits`
`FROM Deposit`
`GROUP BY`
`branchName`
`HAVING`
`COUNT(*) > 1;`
	`BRANCHNAME   SUM(BALANCE)`
	`----------   ------------`
	`Midlands     3`
	`RoyalBank    2`

Example: ALL clause

Write a SQL statement to find which customer name with a deposit that has the highest balance out of all the deposits.

Statement	Output

`SELECT customerName`
`FROM Deposit`
`WHERE balance >= ALL`
`( SELECT balance`
`FROM Deposit );`
 	
OR
 	
`SELECT customerName`
`FROM Deposit`
`WHERE balance >= (SELECT MAX(balance) FROM Deposit);`
	`CUSTOMERNAME`
	`------------`
	`Jones`

Practical: Subqueries and Views

Views

Example: Creating a view

Create a view that displays the customer names, customer cities, branch names and branch cities for each customer that has a deposit.

Statement

```
CREATE OR REPLACE VIEW customersDetails AS
      SELECT Deposit.customerName, customerCity, Deposit.branchName, branchCity
      FROM
            Deposit JOIN Branch
            ON Deposit.branchName = Branch.branchName
            Deposit JOIN Customer
            ON Deposit.customerName = Customer.customerName
```

OR

```
CREATE OR REPLACE VIEW customerDetails AS
      SELECT Deposit.customerName, customerCity, Deposit.branchName, branchCity
      FROM Deposit, Branch, Customer
      WHERE Deposit.branchName = Branch.branchName
      AND Deposit.customerName = Customer.customerName;
```

Example: Testing a view

Test the view created above.

Statement

```
SELECT * FROM customersDetails;
```

Output

```
CUSTOMERNAME  CUSTOMERCITY  BRANCHNAME  BRANCHCITY
------------  ------------  ----------  ----------
Jones         Nottingham    HFE         Derby
Jones         Nottingham    Yorkshire   Nottingham
Patel         Nottingham    Midlands    Nottingham
Patel         Nottingham    RoyalBank   Nottingham
Smith         Leicester     Midlands    Nottingham
Ahmed         Derby         RoyalBank   Nottingham
Braun         Derby         Southern    Derby
Braun         Derby         Midlands    Nottingham
```

Theory: PL/SQL and Stored Procedures

Introduction to PL/SQL

What is PL/SQL?

PL/SQL is Oracle's programming language extension of SQL.

Every DBMS provides a programming language in which SQL statements can be embedded, for example:

- Microsoft SQL server has T-SQL;
- Sybase has T-SQL; and
- IBM DB-2 has SQL PL.

These programming languages offer procedural extensions to SQL, including features such as:

- variables;
- loops; and
- conditional statements.

The basic constructs of these programming languages are often transferrable between different programming language extensions of SQL, although the syntax may differ.

Theory: PL/SQL and Stored Procedures

Blocks

Definitions

Blocks are similar to the usage of curly brackets ({ }) that enclose functions/procedures or constructs such as if statements and loops in high level programming languages, such as C++.

A **PL/SQL block** is a set of programming statements extending SQL in PL/SQL.

A **named block** is a set of PL/SQL blocks that are named and stored as database objects on the server. They are compiled and can be called by other SQL statements. Named blocks one of two types:

- a stored procedure: or
- a stored function.

An **anonymous block** is a set of PL/SQL blocks that do not have names assigned to them and are not stored as database objects on the server. They can be compiled and immediately executed and must be created and used in the same session as they are not stored on the server.

Theory: PL/SQL and Stored Procedures

Stored procedures

Definition

Stored procedures are named and pre-compiled PL/SQL blocks that:

- are stored on the server;
- can be called to be executed in SQL statements in a similar manner to SQL system-defined functions, such as `MAX` and `AVG`; and
- are compiled in database and are optimised.

Why use stored procedures?

Stored procedures are compiled on the database on the server-side, they are optimised for that database. This is often faster than trying to complete the same operations in other high-level programming languages, such as C++, that make connection to the database.

In a typical operation on data in a database, a `SELECT` statement must be used to obtain some data. During this operation:

- the `SELECT` statement must be sent from the high-level programming language to the database server;
- the resulting data must be sent back from the database server to the high-level programming language; and
- subsequent operations must make further connections with the database to send the resulting data back to the database server.

Whereas, a stored procedure can complete all of these operations server-side and therefore yield from better optimisation and bypass the potentially slow connections between the database server and the high-level programming language.

Stored procedures with single PL/SQL block

Format: Creating a stored procedure with a single PL/SQL block

```
CREATE PROCEDURE <name>(<argument list>) AS
<code to declare variables>
BEGIN
      <code statements>
EXCEPTION
      <code to deal with exceptions>
END;
/
```

The argument list is formatted using a list of comma-separated values, each of which is comprised of:

`<argument name> <data type>`.

The size of the arguments need not be specified.

There must be code to declare the variables used locally in the procedure as to assign space in memory and name the variable. Code statements can be written after the `BEGIN` keyword. Any code to deal with exceptions/errors in the code can be written after the `EXCEPTION` keyword.

The forward-slash (`/`) is required at the end of the stored procedure as the code statements in the stored procedure are likely to contain semi-colons (`;`) that dictate the end of a statement. As a result, a “stronger” dictation of the end of the stored procedure is required.

Theory: PL/SQL and Stored Procedures

Stored procedures with nested PL/SQL blocks

Format: Creating a stored procedure with nested PL/SQL blocks

```
CREATE PROCEDURE <name>(<argument type list>) AS
<code to declare variables>
BEGIN
<code statements>
      DECLARE
      <code to declare variables>
      BEGIN
      <code statements>
      EXCEPTION
      <code to deal with exceptions>
      END;
<code statements>
EXCEPTION
      <code to deal with exceptions>
END;
/
```

Nested PL/SQL blocks must start with the keyword `DECLARE`.

Each nested PL/SQL block inherits its own local scope and therefore will only access variables that have been created in the block itself. For example, should an integer variable named `x` be declared in the outer block, another integer variable named `x` could be declared in the nested block and act independently. As a result, when leaving the blocks, the memory for those variables is released.

Declaration

The first part of a stored procedure declares its name and arguments.

Example: Declaration of a stored procedure

Write some PL/SQL code to declare a stored procedure named `TransferToCustomerLoan` with the arguments:

- **`theName` `VARCHAR`**
- **`theAccountNo` `INT`**
- **`theLoanNo` `INT`**
- **`theTransfer` `DEC`**

Statement

```
CREATE PROCEDURE TransferToCustomerLoan(theName VARCHAR, theAccountNo INT,
      theLoanNo INT, theTransfer DEC) AS
...
```

The variable names in the example above begin with “`the`”. This is a technique to avoid confusion between variables used in stored procedures and column names in tables. For example, the column `accountNo` may exist in a table in the database and therefore its counterpart variable could be named `theAccountNo`.

No size is required for the arguments, as shown by the omission of the size of the `VARCHAR` and `DEC` data types in the example above. In addition, the keyword `DECLARE` is not required for the first PL/SQL block in the stored procedure.

Theory: PL/SQL and Stored Procedures

Using arguments

Example: Using arguments in a stored procedure
Write some PL/SQL code to create a stored procedure named `TransferToLoan` with the arguments: • **`theAccountNo INT`** • **`theLoanNo INT`** • **`theTransfer DEC`** **The stored procedure should allow a specified amount to be transferred from a specified deposit account to a specified loan account.**

Statement

```
CREATE PROCEDURE TransferToLoan(theAccountNo INT, theLoanNo INT, theTransfer DEC) AS
BEGIN

UPDATE Deposit
SET balance = balance - theTransfer
WHERE accountNumber = theAccountNo;

UPDATE Loan
SET amount = amount - theTransfer
WHERE accountNumber = theLoanNo;

END;
/
```

The example above allows arguments to be passed in to the stored procedure that are used to determine how much should be transferred and the account numbers for the deposit and loan accounts.

This stored procedure could become more sophisticated by adding control, such as IF statements that do not transfers that are too large and may result in invalid negative loan amounts.

Executing and calling stored procedures

When the stored procedure is executed, real values will be put in place of the arguments and used in the SQL statements in the stored procedure.

Environment	Uses	Example
SQL*Plus	The `execute` SQL statement	`execute TransferToLoan(1, 61, 250);`
PL/SQL code	The `CALL` SQL statement	`CALL TransferToLoan(1, 61, 250);`

Application Express (APEX)	PL/SQL code
	
	Runs an anonymous block of PL/SQL code
		`begin`
		`TransferToLoan(1, 61, 250);`
		`end;`
		`/`

When the stored procedure `TransferToLoan` is called, its arguments are set as follows:

```
theAccountNo  =  1
theLoanNo     =  61
theTransfer   =  250
```

Theory: PL/SQL and Stored Procedures

PL/SQL control and variables

Comments in PL/SQL code

It may be useful to comment PL/SQL code in order to increase code readability.

Single-line comments begin with a double hyphen (`--`).
Multi-line comments begin with a slash-asterisk (`/*`), and end with an asterisk-slash (`*/`).

Format: Comments in PL/SQL code

Single-line	Multi-line
`-- this is a comment`	`/* This is` `   a comment */`

Using control statements

IF statements

An **IF statement** provides conditional control.

Format: CASE statement in a PL/SQL block

```
IF <condition> THEN
      <code statements>
ELSIF <condition> THEN
      <code statements>
ELSE
      <code statements>
END IF;
```

CASE statements

The **CASE statement** chooses from a sequence of conditions and executes a corresponding statement. The CASE statement evaluates a single expression and compares it against several potential values or evaluates multiple Boolean expressions and chooses the first one that is `TRUE`.

Format: CASE statement in a PL/SQL block

```
CASE <expression>
      WHEN <condition> THEN
            <code statements>
      WHEN <condition> THEN
            <code statements>
      ...
      WHEN <condition> THEN
            <code statements>
      ELSE
            <code statements>
END
```

Theory: PL/SQL and Stored Procedures

Using loops

Loops

The **LOOP statement** executes a sequence of statements within a PL/SQL code block multiple times.

Format: LOOP in a PL/SQL block

```
LOOP
      IF <condition(s) to exit loop> THEN
            EXIT;
      END IF;
END LOOP;
```

A LOOP requires a method to exit and this can be implemented using an IF statement that runs the `EXIT` statement if specific condition(s) are met.

FOR loops

The **FOR statement** are used to execute a set of SQL statements more than once.

Format: FOR loop in a PL/SQL block

```
FOR i in <start> .. <end> LOOP
      <code statements>
END LOOP;
```

A for loop takes an integer start value and integer end value. For example, a for loop with a start value of `1` and end value of `10` would iterate from 1 to 10.

WHILE loops

A **WHILE loop** repeats a set of SQL statements as long as a specified expression is true. The condition is evaluated immediately before each entry into the loop body.

Format: WHILE loop in a PL/SQL block

```
WHILE <condition(s) to remain in loop) LOOP
      <code statements>
END LOOP;
```

A while loop will remain in execution until specific condition(s) are met.

Theory: PL/SQL and Stored Procedures

Variables and constants

Variables

A **variable** is a named space in memory. It can be defined by specifying its name and data type.

Format: Declaring a variable

```
<variable name> <variable data type>;
```

Example: Declaring a variable

Write some PL/SQL code to declare a variable with name `aCustomerName` and data type `CHAR` with size 3 and a variable with name `anAccountNumber` and data type `INT`.

Statement
`aCustomerName CHAR(3);` `anAccountNumber INT;`

Constants

A **constant** is a named space in memory that remains unchanged throughout the program and is a user-defined literal value. It can be defined by specifying its name, using the keyword `CONTSTANT` and then specifying its data type and assigning a value using the assignment operator.

Format: Declaring a constant

```
<constant name> CONSTANT <constant data type> := <value>
```

Example: Declaring a variable

Write some PL/SQL code to declare a constant with name `anInterestRate`, data type `DECIMAL` with precision 6 and scale 2 and a value of `0.08`.

Statement
`anInterestRate CONSTANT DECIMAL(6,2) := 0.08;`

In a PL/SQL block the keyword `DECLARE` precedes the variables and constants, except those in the first PL/SQL block.

Variable types based on table columns

The variable type can be declared to the as the data type of a column in a table.

Format: Declaring a variable with data type of a table column

```
<variable name> <table name>.<column name>%TYPE;
```

Theory: PL/SQL and Stored Procedures

Example: Declaring a variable with data type of a table column
Write some PL/SQL code to declare a variable with name `aCustomerName` and data type same to that of the `customerName` column in the `Deposit` table.
Statement
`aCustomerName Deposit.customerName%TYPE;`

Variable types based on table rows

The variable type can be declared to the as the data type of a row in a table.

Format: Declaring a variable with data type of a table row

```
<variable name> <table name>%ROWTYPE;
```

Example: Declaring a variable with data type of a table row
Write some PL/SQL code to declare a variable with name `aDepositRecord` and data type same to that of a row in the `Deposit` table.
Statement
`aDepositRecord Deposit%ROWTYPE;`

Declaring a variable with the data type of a table row will result in the variable becoming a one-dimensional (1D) array of each column in the table specified. Subsequently, specifying the variable will return the array or the dot notation can be used to access specific values in the one-dimensional (1D) array.

Format: Accessing specific values in the one-dimensional (1D) array

```
<variable name>.<column name>
```

Example: Accessing specific values in the one-dimensional (1D) array
Write some PL/SQL code to access the value for `branchName` in the `aDepositRecord` array.
Statement
`aDepositRecord.branchName`

These methods of declaring data types for variables are useful as the code need not be updated should the data type of a column or row be changed in the future, nor should the data type of the column or row be remembered when writing the code.

Theory: PL/SQL and Stored Procedures

PL/SQL implicit cursors, output and explicit cursors

Implicit cursors

An **implicit cursor** stores the result of a query using the `SELECT INTO` statement for a single row/value.

The **`SELECT INTO`** statement allows a single row/value to be stored in a variable within the scope of the `SELECT INTO` statement.

Format: Using `SELECT INTO`

```
SELECT <column name> INTO <variable name>
FROM <table name>
WHERE <condition(s) to return only one value>;
```

Example: Using `SELECT INTO`

Write some PL/SQL code to store the `balance` of the account with `accountNumber` 4 from the `Deposit` table in the variable `aBalance`.

Statement

```
SELECT balance INTO aBalance
FROM Deposit
WHERE accountNumber = 4;
```

In this example, the `WHERE` clause uses the primary key `accountNumber` in which there is only one possible row in the table with an account number value of `4`. This is important as, in the case that more than one value is returned, there would be a runtime error as the variable `balance` cannot hold more than one value.

Output to screen

The DBMS offers the function `put_line()` in the `dbms_output` package to output data to the screen.

Format: Outputting to the screen

```
dbms_output.put_line(<data>);
```

Example: Outputting to the screen

Write some PL/SQL code to output the string '`Hello`' to the screen.

Statement

```
dbms_output.put_line('Hello');
```

Different data types can be concatenated in the output by using the concatenation symbol which is two vertical bars (`||`).

Format: Outputting different to the screen

```
dbms_output.put_line(<data>||<data>||...||<data>);
```

Theory: PL/SQL and Stored Procedures

Example: Outputting different to the screen
Write some PL/SQL code to output the string '`Hello `' and the integer 5 to the screen.
Statement
`dbms_output.put_line('Hello '‖5);`

In order to view the output on the client-side, server output must be activated using the following command:

```
set serveroutput on
```

Using implicit cursors and output in a stored procedure

Example: Implicit cursor and output in stored procedure

Write some PL/SQL code to create a stored procedure named `getBalance` with the arguments:

- **`theAccountNumber INT`**

The stored procedure should store the `balance` of the account with the specified `accountNumber` from the `Deposit` table in the variable `aBalance` and output the value to the screen.

Statement

```
CREATE OR REPLACE PROCEDURE getBalance(theAccountNumber INT) AS
      aBalance Deposit.balance%TYPE;
BEGIN
      SELECT balance INTO aBalance
      FROM Deposit
      WHERE accountNumber = theAccountNumber;
      dbms_output.put_line('Balance ='||aBalance);
END;
/
```

Explicit cursors

An **explicit cursor** stores the result of a query for multiple rows/values. They are is used when the result set of a pre-determined `SELECT` statement returns more than one row/value.

There are four stages to using an explicit cursor:

- declare – signals to the DBMS that an explicit cursor is being created, similar to other local variables;
- open – loads the cursor to allow its values to be accessed;
- fetch – uses a loop to fetch the values out of the cursor to be used somewhere else in the code; and
- close – releases the memory back to the user space.

Format: Declare cursor

```
CURSOR <cursor name> IS
      SELECT <column name>
      FROM <table name>
      WHERE <optional condition(s)>;
```

Theory: PL/SQL and Stored Procedures

Format: Open cursor

```
OPEN <cursor name>;
```

Format: Fetch from cursor

```
LOOP
      FETCH <cursor name> INTO <variable name>;
      EXIT WHEN <cursor name>%NOTFOUND;
      <code statements>
END LOOP;
```

The loop first fetches each value one-by-one from the cursor in to the specified variable. The loop will continue to repeat until the `NOTFOUND` flag for the cursor evaluates to True as this means that no further rows are left in the cursor. Otherwise, any code statements can be written that make use of the values fetched from the cursor.

Format: Close cursor

```
CLOSE <cursor name>;
```

Using explicit cursors and output in a stored procedure

Example: Explicit cursor and output in stored procedure

Write some PL/SQL code to create a stored procedure named `outputCustomers`. The stored procedure should store the output the `customerName` of each of the accounts in the `Customers` table.

Statement

```
CREATE OR REPLACE PROCEDURE outputCustomers AS

      CURSOR someCustomerRows IS  -- declare
            SELECT customerName
            FROM Customer;

      aCustomerName    Customer.customerName%TYPE;

BEGIN
      OPEN someCustomerRows;  -- open (loads cursor)

      LOOP
            FETCH someCustomerRows INTO aCustomerName;  -- fetch
            EXIT WHEN someCustomerRows%NOTFOUND;  /* if no more rows left in cursor
                                                     NOTFOUND flag is True */
            dbms_output.put_line('Customer Name = '|| aCustomerName);
      END LOOP;

      CLOSE someCustomerRows;  -- close
END;
/
```

Theory: PL/SQL and Stored Procedures

Stored functions

Definition

Stored functions are named and pre-compiled PL/SQL blocks that:

- are stored on the server;
- can be called to be executed in SQL statements in a similar manner to SQL system-defined functions, such as `MAX` and `AVG`;
- are compiled in database and are optimised; and
- have a return value.

Stored functions share most of the same attributes as stored procedures, other than their ability to return a value.

Stored functions with single PL/SQL block

Format: Creating a stored function with a single PL/SQL block

```
CREATE FUNCTION <name>(<parameter list>) RETURN <data type> IS
<code to declare variables>
BEGIN
      <code statements>
EXCEPTION
      <code to deal with exceptions>
END;
/
```

The argument list is formatted using a list of comma-separated values, each of which is comprised of:

`<argument name> <data type>`.

The size of the arguments need not be specified.

The data type of the return value must be specified after the `RETURN` keyword. The size of the return value need not be specified.

There must be code to declare the variables used locally in the procedure as to assign space in memory and name the variable. Code statements can be written after the `BEGIN` keyword. Any code to deal with exceptions/errors in the code can be written after the `EXCEPTION` keyword.

The forward-slash (`/`) is required at the end of the stored procedure as the code statements in the stored procedure are likely to contain semi-colons (`;`) that dictate the end of a statement. As a result, a “stronger” dictation of the end of the stored procedure is required.

Theory: PL/SQL and Stored Procedures

Example: Creating a stored function with a single PL/SQL block

Write some PL/SQL code to create a stored function named `getBalance` with the arguments:

- **`theAccountNumber  INT`**

and return type `DECIMAL`.

The stored procedure should store the `balance` of the account with the specified `accountNumber` from the `Deposit` table in the variable `aBalance` and return the value.

Statement

```
CREATE OR REPLACE FUNCTION getBalance(theAccountNumber INT) RETURN DECIMAL IS
      aBalance Deposit.balance%TYPE;
BEGIN
      SELECT balance INTO aBalance
      FROM Deposit
      WHERE accountNumber = theAccountNumber;
      RETURN aBalance;
END;
/
```

Executing and calling stored functions

When the stored function is executed, real values will be put in place of the arguments and used in the SQL statements in the stored function and the return value will be assigned to a variable or used in an SQL statement.

Environment	Uses	Example

SQL*Plus	The `execute` SQL statement	`variable x NUMBER`
		`execute :x := getBalance(22) /* returns a DECIMAL value and assigns value to x */`
		`print x`

PL/SQL code	The `CALL` SQL statement	`CALL getBalance(22) INTO :x; /* returns a DECIMAL value and assigns value to x */` `print x`

SQL code	Embedded call	`SELECT getBalance(accountNumber) /* by default, uses argument of each account number one-by-one from the deposit table and returns a DECIMAL value */`
		`FROM Deposit`
		`WHERE accountNumber = 22; /* restricts possible arguments to just one value (22) */`

When the stored function `getBalance` is called, its arguments are set as follows:

```
      theAccountNumber  =  22
```

Practical: PL/SQL and Stored Procedures

Stored procedures

Example: Updating values

Write a single PL/SQL stored procedure that will add interest to all deposits (in table `Deposit`) for a specified branch name and at a specified rate of interest (i.e. enter the branch name and interest rate as arguments in the stored procedure argument list).

Execute the stored procedure from SQL*Plus.

PL/SQL code

```
CREATE OR REPLACE PROCEDURE newBalances(aBranchName VARCHAR, anInterestRate
DECIMAL) AS
BEGIN
      UPDATE Deposit
      SET balance = balance * (100.0 + anInterestRate)/ 100.0
      WHERE branchName = aBranchName;
END;
/
```

Statement	Output
`execute newBalances('Yorkshire', 5.0);`	`PL/SQL procedure successfully completed.`
OR	
`CALL newBalances ('Yorkshire',5.0);`	`Call completed.`

Example: Outputting values

Edit the stored procedure from the example above so that it will output to the screen the customer names and new balances for each deposit updated.

Execute the stored procedure from SQL*Plus.

PL/SQL code

```
CREATE OR REPLACE PROCEDURE newBalances(aBranchName VARCHAR, anInterestRate DECIMAL) AS

      CURSOR someDepositRows IS
            SELECT customerName, balance
            FROM Deposit
            WHERE branchName = aBranchName;

      aCustomerNameRow Deposit.customerName%TYPE;
      aBalanceRow Deposit.balance%TYPE;

BEGIN
      UPDATE Deposit
      SET balance = balance * (100.0 + anInterestRate) / 100.0
      WHERE branchName = aBranchName;

      OPEN someDepositRows;
      dbms_output.put_line('Customer Name - '||'Balance - ' );

      LOOP
            FETCH someDepositRows INTO aCustomerNameRow, aBalanceRow;
            EXIT WHEN someDepositRows%NOTFOUND;
            dbms_output.put_line(aCustomerNameRow|| ' - ' || aBalanceRow);
      END LOOP;

      CLOSE someDepositRows;
END;
/
```

Practical: PL/SQL and Stored Procedures

OR

```
CREATE OR REPLACE PROCEDURE newBalances(aBranchName VARCHAR, anInterestRate DECIMAL) AS

      CURSOR someDepositRows IS
            SELECT * FROM Deposit
            WHERE branchName = aBranchName;
            aDepositRow Deposit%ROWTYPE;

BEGIN
      UPDATE Deposit
      SET balance = balance * (100.0 + anInterestRate) / 100.0
      WHERE branchName = aBranchName;

      OPEN someDepositRows;
      dbms_output.put_line('Customer Name - ' || 'Balance - ' );

      LOOP
            FETCH someDepositRows INTO aDepositRow;
            EXIT WHEN someDepositRows%NOTFOUND;
            dbms_output.put_line(aDepositRow.customerName|| ' -'||aDepositRow.Balance);
      END LOOP;

      CLOSE someDepositRows;
END;
/
```

Statement	Output
`execute newBalances('Yorkshire', 5.0);`	Customer Name - Balance - Jones - 115.76 PL/SQL procedure successfully completed.
OR	
`CALL newBalances ('Yorkshire',5.0);`	Customer Name - Balance - Jones - 115.76 Call completed.

Practical: PL/SQL and Stored Procedures

Stored functions

Example: Outputting values

Write a stored function that will output the total amount loaned by a specified branch name (enter the branch name as an argument in the stored function argument list and return the total amount).

Execute the stored function from SQL*Plus.

PL/SQL code

```
CREATE OR REPLACE FUNCTION getTotalLoaned(aBranchName VARCHAR) RETURN DECIMAL IS
      aTotalLoaned Loan.amount%TYPE;
BEGIN
      SELECT SUM(amount) INTO aTotalLoaned
      FROM Loan
      WHERE branchName = aBranchName;

      RETURN aTotalLoaned;
END;
/
```

Statement	Output

`variable x NUMBER`
`execute :x := getTotalLoaned ('Yorkshire')`
`print x`
	`PL/SQL procedure successfully completed.`
	
	`X`
	`----------`
	`7300`

OR	

`CALL getTotalLoaned('Yorkshire') INTO :x;`
`print x`
	`Call completed.`
	
	`X`
	`----------`
	`7300`

Example: Using stored functions in `SELECT` statements

Use the stored function in a SELECT statement that uses the branch name from the Branch table as the argument.

Statement	Output
`SELECT branchname, getTotalLoaned(branchname) FROM Branch;`	See below

```
BRANCHNAME  GETTOTALLOANED(BRANCHNAME)
----------  --------------------------
HFE
Midlands    8000
RoyalBank    500
Southern
Yorkshire   7300
```

Theory: PL/SQL and Triggers

Introduction to triggers

Definitions

An **active database** is one where some of the data maintenance is performed automatically by the system, such as by triggers.

A **passive database** is one where the data maintenance is initiated solely by the users and implementation of referential integrity.

A **trigger** defines an action that the database should take when some event occurs in the application. A trigger may be used to enforce some referential integrity constraints, to enforce complex constraints, or to audit changes to data.

Types of triggers

Events that could cause the invocation of a trigger include:

- an `INSERT` statement on a specific table (or view, in some cases);
- an `UPDATE` statement on a specific table (or view, in some cases);
- a `DELETE` statement on a specific table (or view, in some cases);
- a `CREATE` statement on any schema object;
- a `CREATE` statement on any schema object;
- an `ALTER` statement on any schema object;
- a `DROP` statement on any schema object;
- a database startup or instance shutdown;
- a specific error message;
- any error message;
- a user logon; or
- a user logoff.

Theory: PL/SQL and Triggers

Syntax

General format

The general format of syntax for a trigger leaves many options available for configuration of how the trigger is invoked and what action it has on the database.

Format: Creating a trigger

```
CREATE TRIGGER <trigger name>
      < BEFORE | AFTER | INSTEAD OF >
      < INSERT | DELETE | UPDATE [ of column list ] >
      ON < table name | view name >
      REFERENCING < OLD AS old-correlation-name | NEW AS new-correlation-name >
      FOR EACH < ROW | STATEMENT >
      WHEN <condition>
      < code statements | CALL procedure/function >
```

Level

A trigger may be either:

- table-level (or statement-level) – identified by the FOR EACH ROW clause in the CREATE TRIGGER command, and triggers for each row in the table;
- row-level – triggers execute once for each transaction no matter the number of rows in the table.

Before and after

A trigger may be a BEFORE trigger or an AFTER trigger.

BEFORE trigger	AFTER trigger
Are executed before the triggering events.	Are executed after the triggering events.
Cannot activate other triggers.	Can activate other triggers.
New values can be modified before entering tables.	New values cannot be modified before entering tables.

Theory: PL/SQL and Triggers

Table-level triggers

On INSERT or DELETE

Example: Trigger on INSERT or DELETE

Write an SQL statement to provide all loan customers in the Midlands branch with a £200 deposit account. Let the loan number serve as the account number for the new deposit account.

Write an SQL statement to create a table named `TotalCustomers` with the columns:

- **`noCustomers` `INTEGER`**
- **`dateRecorded` `DATE`**

The primary key should be a compound key of `noCustomers` and `dateRecorded`.

Write some PL/SQL code to create a trigger named `keepTotalCustomers` that adds a record to the `totalCustomers` table containing the current number of customers in the `Deposit` table and the current date.

SQL Statement

```
INSERT INTO Deposit
      SELECT customerName, branchName, loanNumber, 200
      FROM Loan
      WHERE branchName = 'Midlands';
```

SQL Statement

```
CREATE TABLE TotalCustomers (
      noCustomers INTEGER,
      dateRecorded DATE,
      PRIMARY KEY (noCustomers, dateRecorded)
);
```

PL/SQL Code

```
CREATE OR REPLACE TRIGGER keepTotalCustomers
      AFTER INSERT OR DELETE ON Deposit
      BEGIN
            INSERT INTO TotalCustomers
            ( SELECT COUNT(customerName), SYSDATE
              FROM Deposit );
      END;
      /
```

To demonstrate the functionality of the trigger created in the example above, a row should be deleted from the `Deposit` table to invoke the trigger on `AFTER DELETE ON Deposit`:

```
      DELETE FROM Deposit WHERE accountNumber = '61';
```

Subsequently, the trigger is executed and the `totalCustomers` table is now as shown below.

```
NOCUSTOMERS DATERECORED
----------- -----------
         10   03-JAN-19
```

Adding the deleted row back to the `Deposit` table should invoke the trigger on `AFTER INSERT ON Deposit`:

```
      INSERT INTO Deposit VALUES ('Jones', 'Midlands', 61, 200);
```

Yet again, the trigger is executed and the `totalCustomers` tables is now as shown below.

```
NOCUSTOMERS DATERECORED
----------- -----------
         10   03-JAN-19
         11   03-JAN-19
```

Row-level triggers: new and old

What are new and old values?

	new	old
Relation	Relates to the new row values of the table on which the trigger is created	Relates to the previous row values of the table on which the trigger is created
SQL	`:new.<column name>`	`:old.<column name>`

Using new and old with BEFORE and AFTER

	new		old	
BEFORE	**Read**	**Write**	**Read**	**Write**
	Yes	Yes	Yes	No
AFTER	**Read**	**Write**	**Read**	**Write**
	Yes	No	Yes	No

The table above shows that:

- `BEFORE` can write to the `new` value but not the `old` value;
- `AFTER` cannot write to `new` or `old` values; and
- `BEFORE` and after can read both `new` and `old` values.

Using new and old with INSERT, DELETE and UPDATE

	new	old
INSERT	Yes	No
DELETE	No	Yes
UPDATE	Yes	No

The table above shows that:

- a trigger invoked by an `INSERT` statement has meaningful access to `new` values only, this is because the row is being created by the `INSERT` statement and therefore `old` values will be `NULL`;
- a trigger invoked by a `DELETE` statement has meaningful access to `old` values only, this is because the row no longer exists after the row is deleted and therefore the `new` values will be `NULL` (if trying to modify new values, `ORA-4084` would be raised); and
- a trigger invoked by an `UPDATE` statement has meaningful access to both `new` and `old` values for both `BEFORE` and `AFTER` row triggers.

Theory: PL/SQL and Triggers

Examples

Example: Trigger using new values

Write an SQL statement that adds a column named `noCust` with data type `INT` to the `Deposit` table.

Write an SQL statement that initialises the row values of the `noCust` column in the `Deposit` table to the current number of customers in each branch with a deposit.

Write some PL/SQL code to create a trigger named `addCustomer` that maintains the number of customers in each branch with a deposit when customers are added to the `Deposit` table.

SQL Statement

```
ALTER TABLE Branch ADD (
      noCust INT
);
```

SQL Statement

```
UPDATE Branch
      SET noCust = ( SELECT COUNT(*)
                     FROM Deposit
                     WHERE Deposit.branchName = Branch.branchName )
;
```

PL/SQL Code

```
CREATE TRIGGER addCustomer
      AFTER INSERT ON Deposit
      FOR EACH ROW
      BEGIN
            UPDATE Branch
                  SET noCust = noCust + 1
                  WHERE Branch.branchName = :new.branchName;
      END;
      /
```

Before the trigger is executed the `totalCustomers` table is as shown below.

```
BRANCHNAME ASSETS BRANCHCITY NOCUST
---------- ------ ---------- ------
Yorkshire   10000 Nottingham      1
Midlands    20000 Nottingham      6
RoyalBank   25000 Nottingham      2
HFE         15000      Derby      1
Southern    30000      Derby      1
```

To demonstrate the functionality of the trigger created in the example above, a row should be added to the `Deposit` table to invoke the trigger on `AFTER INSERT ON Deposit`:

```
INSERT INTO Deposit VALUES (‘Jones’, ‘Midlands’, 72, 2000);
```

Subsequently, the trigger is executed and the `totalCustomers` table is now as shown below.

```
BRANCHNAME ASSETS BRANCHCITY NOCUST
---------- ------ ---------- ------
Yorkshire   10000 Nottingham      1
Midlands    20000 Nottingham      7
RoyalBank   25000 Nottingham      2
HFE         15000      Derby      1
Southern    30000      Derby      1
```

As shown above, the `noCust` column for the Midlands branch has been incremented by one to reflect the INSERT on the `Deposit` table.

Using the WHEN clause to distinguish between operations

The WHEN clause can be used in a CASE statement to control which code statements are executed based on the statement that invoked the trigger.

Example: Trigger using WHEN
Write an SQL statement that adds a column named noCust with data type INT to the Deposit table. **Write an SQL statement that initialises the row values of the noCust column in the Deposit table to the current number of customers in each branch with a deposit.** **Write some PL/SQL code to create a trigger named addCustomer that maintains the number of customers in each branch with a deposit when customers are added to or removed from the Deposit table.**

SQL Statement

```
ALTER TABLE Branch ADD (
      noCust INT
);
```

SQL Statement

```
UPDATE Branch
      SET noCust = ( SELECT COUNT(*)
                     FROM Deposit
                     WHERE Deposit.branchName = Branch.branchName )
;
```

PL/SQL Code

```
CREATE OR REPLACE TRIGGER addRemoveCustomer
      AFTER INSERT OR DELETE ON Deposit
      FOR EACH ROW
      BEGIN
            CASE
                  WHEN INSERTING THEN
                        UPDATE Branch
                              SET noCust = noCust + 1
                              WHERE Branch.branchName = :new.branchName;
                  WHEN DELETING THEN
                        UPDATE Branch
                              SET noCust = noCust + 1
                              WHERE Branch.branchName = :new.branchName;
            END CASE;
      END;
      /
```

This improves on the trigger created on page 78 as the noCust column will now be maintained on both events where customers are added and events where customers are removed and therefore the information will be correct.

Theory: PL/SQL and Triggers

Using the WHEN clause to perform checks

The `WHEN` clause can be used to perform checks on `new` and `old` values to enforce rules in the database.

Example: Trigger using WHEN

Write some PL/SQL code to create a trigger named `bigWithdrawal` that enforces the rule that a withdrawal on a deposit account cannot exceed 100.

PL/SQL Code

```
CREATE OR REPLACE TRIGGER bigWithdrawal
      BEFORE UPDATE OF balance ON Deposit
      FOR EACH ROW
      WHEN (old.balance - new.balance > 100)
            BEGIN
                  :new.balance := :old.balance;
            END;
      /
```

Theory: PL/SQL and Triggers

Evaluation

Comparison

	Active Database	Passive Database
Definition	An active database is one where some of the data maintenance is performed automatically by the system, such as by triggers.	A passive database is one where the data maintenance is initiated solely by the users and implementation of referential integrity.
Advantages	Provide very powerful rule-based mechanism.	No risk of excessive use of triggers. This means that there will be less changes of complex interdependencies and therefore increases maintainability in a large applications.
	Useful for automating functionality; triggers can improve the working of a database by removing repetitive error checking and correction.	
Disadvantages	Excessive use of triggers can result in complex interdependencies, which can be difficult to maintain in a large application. For example, when a trigger fires, a SQL statement within its trigger action potentially can fire other triggers, resulting in cascading triggers.	No automation for checking and correction.

Theory: PL/SQL and Triggers

SQL extensions

Extensions to SQL

- Support for multimedia data – Oracle Multimedia
- Support for spatial data – Oracle Spatial
- Addition of object-orientation and user-defined types – SQL:1999
- Support for XML – SQL:2003/2006

Practical: PL/SQL and Triggers

Table-level triggers

Example: Table-level triggers

This example is about adding a new table to store the number of loans in the `Loan` table and adding triggers to automatically update these values.

- **Create a table called `BankStats` that has column '`when`' of type `DATE` (this is the primary key) and column '`numberOfLoans`' of type `INTEGER`.**
- **Insert a row into `BankStats` that contains the current time together with how many loans are in the `Loan` table.**
- **Create a trigger(s) on table `Loan` that will add a new row in `BankStats` whenever a row is inserted into or deleted from the `Loan` table.**
- **Insert and delete a new row into table `Loan` and check that the trigger(s) are working.**

SQL statement

```
CREATE TABLE BankStats (
      when DATE,
      numberOfLoans INT,
      PRIMARY KEY(when,numberOfLoans )
);
```

SQL statement

```
INSERT INTO BankStats
      ( SELECT SYSDATE, COUNT(*)
      FROM Loan );
```

OR

```
INSERT INTO BankStats VALUES (
      SYSDATE,
      ( SELECT COUNT(*)
        FROM Loan )
);
```

SQL statement

```
CREATE OR REPLACE TRIGGER addLoan
      AFTER INSERT ON Loan
      BEGIN
            INSERT INTO BankStats
            ( SELECT SYSDATE, COUNT(*)
              FROM Loan );
      END;
      /

CREATE OR REPLACE TRIGGER removeLoan
      AFTER DELETE ON Loan
      BEGIN
            INSERT INTO BankStats
            ( SELECT SYSDATE, COUNT(*)
              FROM Loan );
      END;
      /
```

OR

Practical: PL/SQL and Triggers

```
CREATE OR REPLACE TRIGGER onChangeLoan
      AFTER INSERT OR DELETE ON Loan
      BEGIN
      INSERT INTO BankStats
            ( SELECT SYSDATE,COUNT(*)
              FROM Loan );
      END;
      /
```

SQL statement	Output
`INSERT INTO Loan VALUES ('Chan', 'Yorkshire', 54, 500);`	

`SELECT * FROM BankStats;`	`WHEN NUMBEROFLOANS`
	`--------- -------------`
	`03-JAN-19 8`

`DELETE FROM Loan WHERE loanNumber = 54;`	

`SELECT * FROM BankStats;`	`WHEN NUMBEROFLOANS`
	`--------- -------------`
	`03-JAN-19 7`

Practical: PL/SQL and Triggers

Row-level triggers

Example: Table-level triggers

This question is about adding a new table to store the number of loans in the `Loan` table for each branch and adding triggers to automatically update these values.

- **Create a table called BankStats2 that has column 'branchName' of type VARCHAR (this is the primary key) and a column '`numberOfLoans`' of type integer.**
- **Enter initial data into `BankStats2` by inserting rows into `BankStats2` that contain the branch names together with how many loans are in the `Loan` table for that branch name.**
- **Create a trigger(s) on table Loan that will add/subtract one to '`numberOfLoans`' in `BankStats2` whenever a row is inserted into or deleted from the `Loan` table for a particular branch name.**
- **Insert and delete a new row into table `Loan` and check that the trigger(s) are working.**

SQL statement

```
CREATE TABLE BankStats2 (
      branchName VARCHAR(20),
      numberOfLoans INT,
      PRIMARY KEY (branchName)
);
```

SQL statement

```
INSERT INTO BankStats2
      ( SELECT Branch.branchName, COUNT(Loan.branchName)
        FROM Branch LEFT JOIN Loan
        ON Branch.branchname = Loan.branchName
        GROUP BY Branch.branchName );
```

SQL statement

```
CREATE OR REPLACE TRIGGER addLoan
      AFTER INSERT ON Loan
      FOR EACH ROW
            BEGIN
                  UPDATE BankStats2
                  SET numberOfLoans = numberOfLoans + 1
                  WHERE BankStats2.branchName = :new.branchName;
            END;
            /

CREATE OR REPLACE TRIGGER removeLoan
      AFTER DELETE ON Loan
      FOR EACH ROW
            BEGIN
                  UPDATE BankStats2
                  SET numberOfLoans = numberOfLoans - 1
                  WHERE BankStats2.branchName = :old.branchName;
            END;
            /
```

OR

Practical: PL/SQL and Triggers

```
CREATE OR REPLACE TRIGGER addOrRemoveLoan
      AFTER INSERT OR DELETE ON Loan
      FOR EACH ROW
            BEGIN
                  CASE
                        WHEN INSERTING THEN
                              UPDATE BankStats2
                              SET numberOfLoans = numberOfLoans + 1
                              WHERE BankStats2.branchName = :new.branchName;
                        WHEN DELETEING THEN
                              UPDATE BankStats2
                              SET numberOfLoans = numberOfLoans - 1
                              WHERE BankStats2.branchName = :old.branchName;
                  END CASE;
            END;
            /
```

OR

```
/* this implementation will insert a row if the branchName does not exist in
   BankStats 2 */

CREATE OR REPLACE TRIGGER addOrRemoveLoan
      AFTER INSERT ON Loan
      FOR EACH ROW
            DECLARE
                  aCount INT;
            BEGIN
                  CASE
                        WHEN INSERTING THEN
                              SELECT COUNT(*) INTO aCount
                              FROM BankStats2
                              WHERE branchName = :new.branchName;
                              IF (aCount=0) THEN
                                    INSERT INTO BankStats2 (branchName, numberOfLoans)
                                    VALUES (:new.branchName, 1);
                              ELSE
                                    UPDATE BankStats2
                                    SET numberOfLoans = numberOfLoans + 1
                                    WHERE BankStats2.branchName = :new.branchName;
                              END IF;
                        WHEN DELETING THEN
                              UPDATE BankStats2
                              SET numberOfLoans = numberOfLoans - 1
                              WHERE BankStats2.branchName = :old.branchName;
                  END CASE;
            END;
            /
```

Practical: PL/SQL and Triggers

SQL statement	Output
`INSERT INTO Loan VALUES ('Chan', 'Yorkshire', 54, 500);`	
`SELECT * FROM BankStats2;`	BRANCHNAME NUMBEROFLOANS ---------- ------------- HFE 0 Midlands 3 RoyalBank 1 Southern 0 Yorkshire 4
`DELETE FROM Loan WHERE loanNumber = 54;`	
`SELECT * FROM BankStats2;`	BRANCHNAME NUMBEROFLOANS ---------- ------------- HFE 0 Midlands 3 RoyalBank 1 Southern 0 Yorkshire 3

DBMS

Theory: Introduction to DBMS

What is a DBMS?

Definitions

A **database** is a shared collection of logically related data.

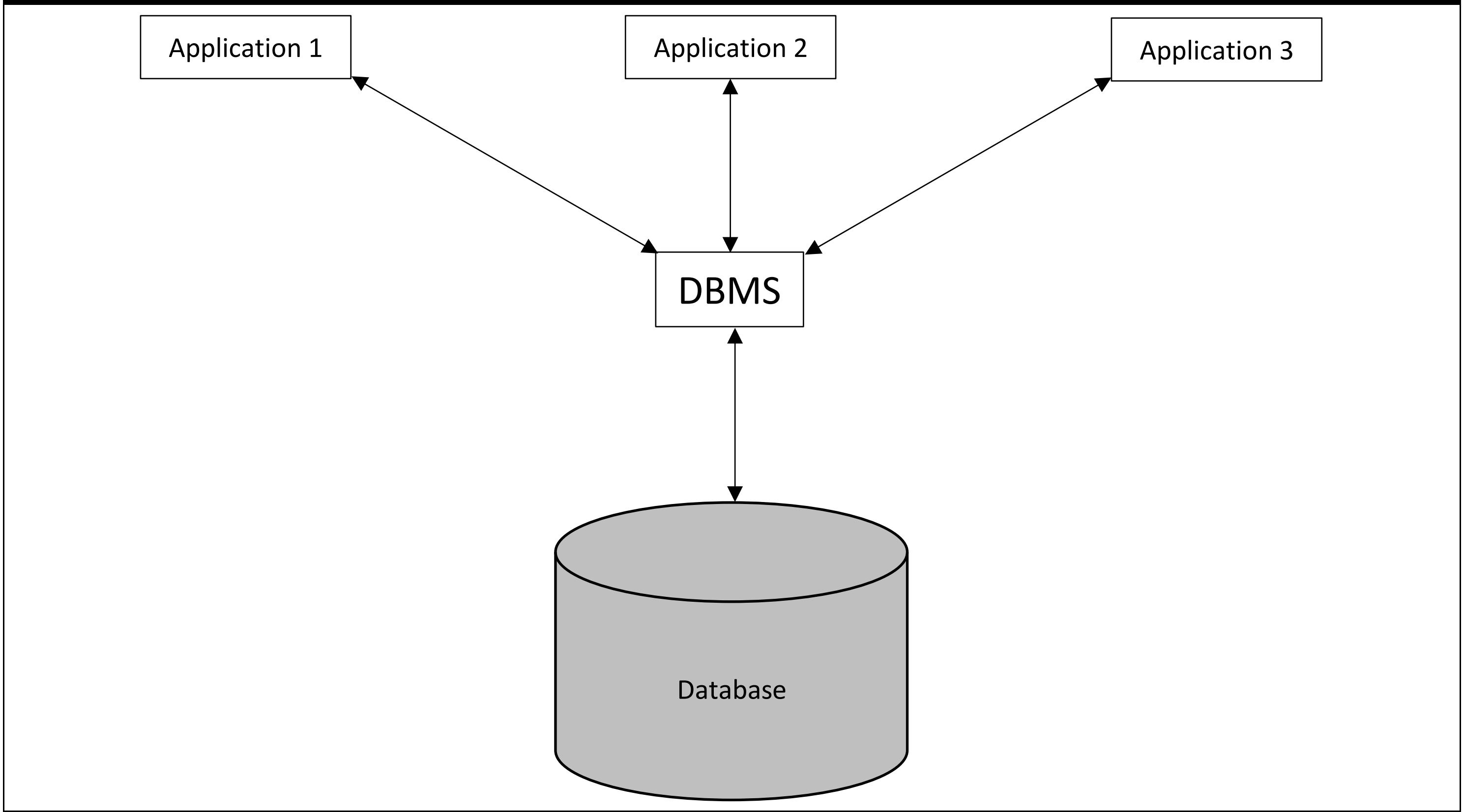
A **database management system (DBMS)** is a software system used to define, create, maintain and control access to one or more databases that may be across one or more database servers. It is a software application for the management of databases.

An **application program** is a computer program that interacts with the database by making requests to the DBMS.

DBMS and application programs

One or more application programs may make connection to a DBMS.

Diagram: Application program connection to DBMS



The DBMS governs interactions between application programs and the database.

These application programs may include:

- Oracle SQL Client;
- Oracle Application Express (APEX); and
- any other program that makes connections to the DBMS, such as through language extensions in C++ or PHP.

Theory: Introduction to DBMS

Role of a DBMS

Properties of a database

A database is a set of related data with the following properties:

- permanently available;
- potentially large;
- integrated;
- usable independently of the program that created them;
- multi-user operation;
- consistent, safe, secure;
- comfortably, flexibly and efficiently usable; and
- possibly distributed in computer networks (i.e. transparency).

Key requirements of a DBMS

A DBMS must deal with the management of **large** amounts of **persistent**, **reliable** and **shared** data. The DBMS must be **available** and offer **security**.

- **Large** – The data is of size that it cannot fit in to main memory; support for a file storage mechanism is required.
- **Persistent** – Data remains present between sessions (the database or application programs are turned on/off).
- **Reliable** – Data is recoverable despite memory or system failures. Recovery can be completed accurately as to recover the state of the database at a specific time (e.g. just before the failure occurred).
- **Shared** – Multiple users can have simultaneous access. A user may have multiple connections to a database.
- **Available** – The DBMS are online and operational at all times and makes the database available.
- **Security** – Authorisation and authentication of access is implemented.

These key requirements of a DBMS have a large influence on the design of a DBMS.

Theory: Introduction to DBMS

DBMS schemas

Definition

A **schema** is the set of metadata (data dictionary) used by the database, typically generated using DDL. A schema defines attributes of the database, such as tables, columns, and properties.

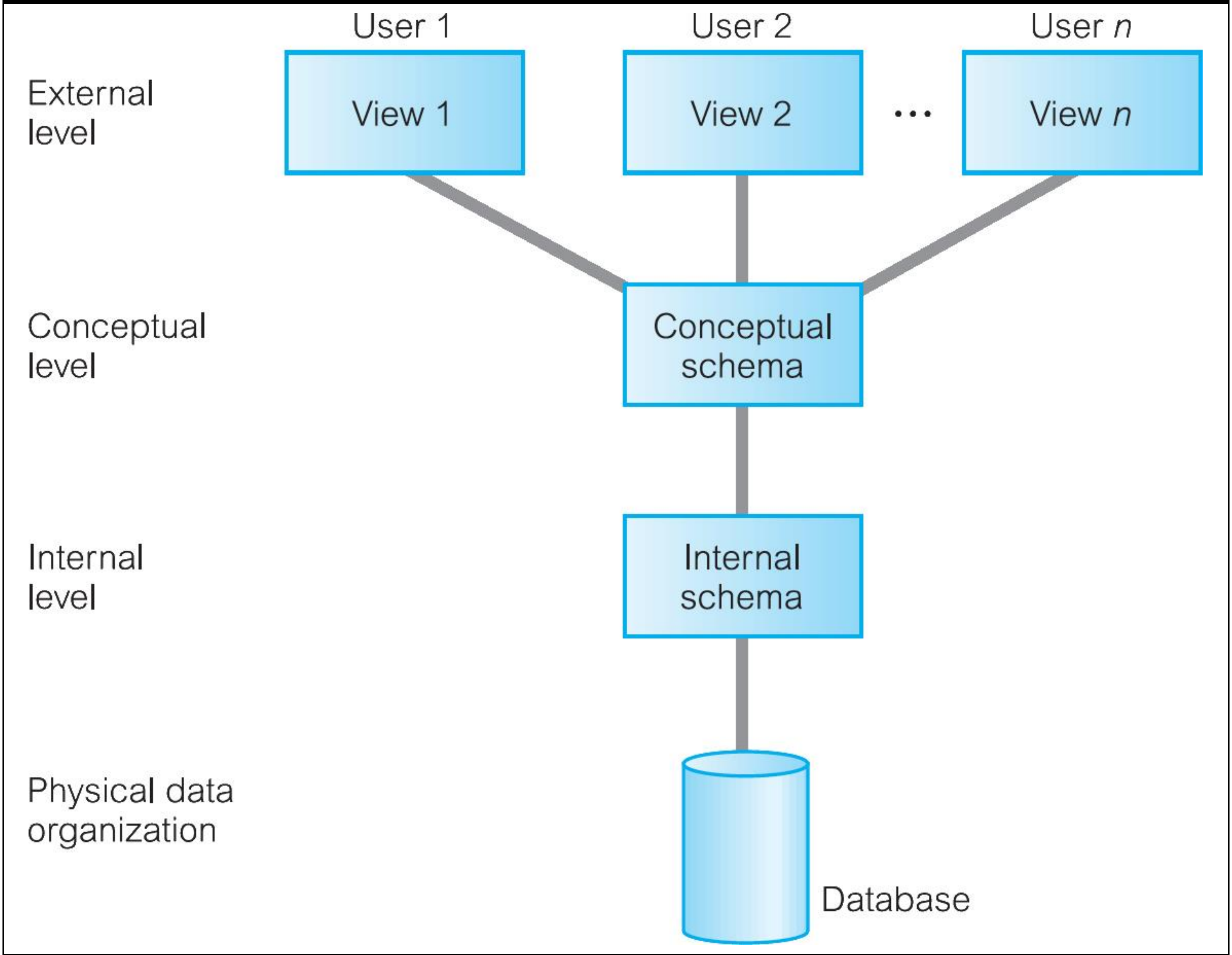
Views of the database schema

A **view** of the data in the database that is prepared for users with different needs and different user access levels.

The **ANSI-SPARC three-level architecture** is an abstract design standard for a DBMS.

Views of the database schema follow the ANSI-SPARC three-level architecture, comprised of three distinct levels.

Diagram: ANSI-SPARC three-level architecture



Theory: Introduction to DBMS

Level	Explanation	Contents
External Level (External Schema) *The way users perceive the database*	• A user's view of the database describes a part of the database that is relevant to a particular user. • Excludes irrelevant data as well as data which the user is not authorised to access.	• Views which limit users to specific tables or specific columns in tables. • Views may be created using `CREATE VIEW` or by using methods in different programming languages

Conceptual Level (Conceptual Schema)
 	
Metadata
	• Describes what data is stored within the whole database and how the data is inter-related.
	• Does not specify how the data is physically stored.
	• Describes the structure of all users.
	• Provides a global view of the database.
	• Uses the idea of data models.
	• Database administrators work at this level and only they can define this level.
	• Independent of hardware and software.
		• Describes all the entities, attributes, and relationships together with integrity constraints.
		• Metadata, which is everything in a database that is not strictly data, including:
		  • tables;
		  • column names;
		  • primary keys / foreign keys;
		  • indexes;
		  • triggers;
		  • stored procedures etc.

Internal Level (Internal Schema) *Data*	• How the database is physically represented on the computer system. • Describes how the data is actually stored in the database and on the computer hardware. • Describes how the data should be accessed – sequential access, direct access or hash files?	• Data structures. • File organisation. • Definitions of stored records. • Methods of representation. • Data fields. • Indexes and storage structures used.

There may be multiple external schemas per database, however there is only one conceptual schema and one internal schema per database.

Below the internal level is the physical data organisation by the operating system (OS). This is concerned with how the operating system (OS) supports the internal level in reading data from and writing data to secondary storage devices.

The objectives of the ANSI-SPARC three-level architecture is based on the idea of separating the views of the database for different users.

- It allows independent customised user views. Each user should be able to access the same data but have a different customised view of the data. These views should be independent, such that changes to one view should not affect other views.
- It hides the physical storage details from users. Users should not have to deal with the physical database storage details.
- The database administrator should be able to change the database storage structures without affecting the users' views.
- The internal structure of the database should be unaffected by changes to the physical aspects for the storage, such as migration to a new storage device.

This allows the DBMS to restrict access to the database for security purposes or in order to present data for a particular purpose.

Theory: Introduction to DBMS

Schema mappings

DBMS responsibility

The DBMS is responsible for mapping between the three types of schema in the ANSI-SPARC three-level architecture. The schemas must be checked for consistency:

- the DBMS must confirm that each external schema is derivable from the conceptual schema; and
- the DBMS must use the information in the conceptual schema to map between each external schema and the internal schema.

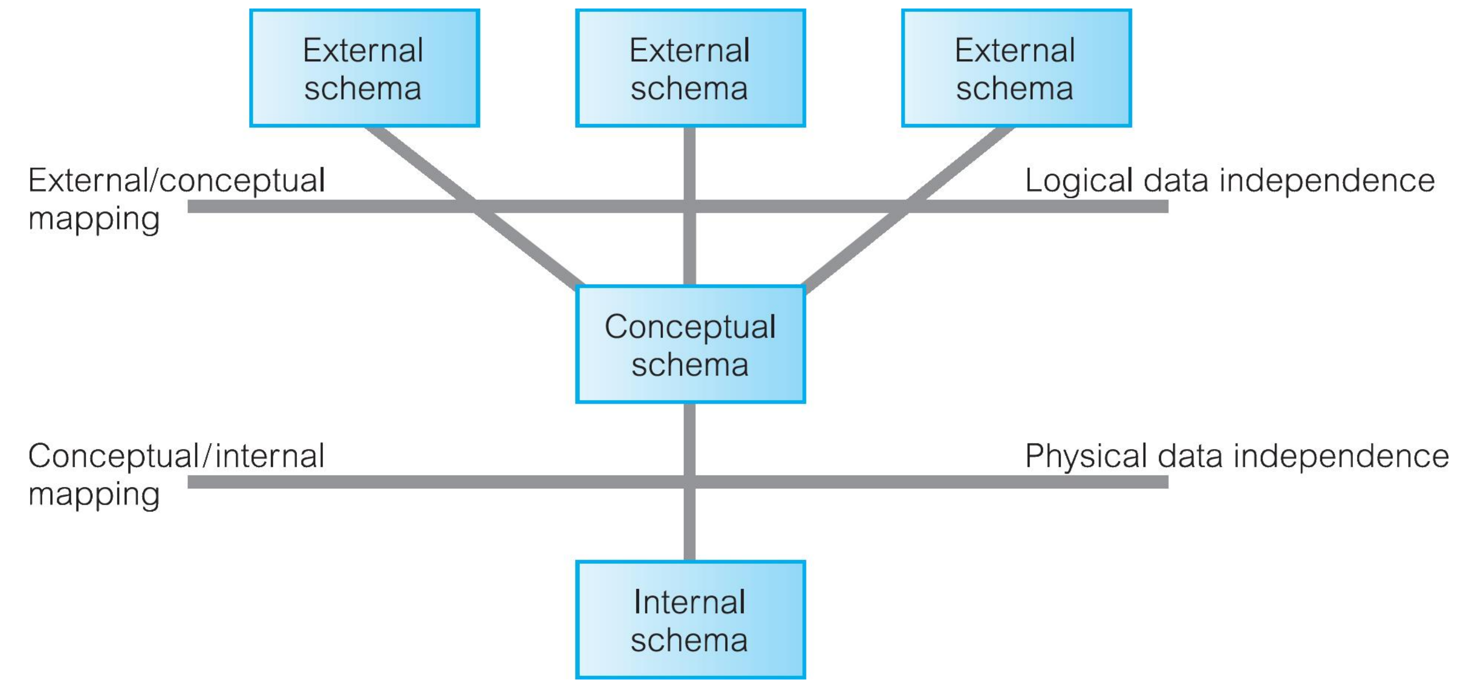
Types of mapping and data independence

A major objective for the ANSI-SPARC three-level architecture is to provide **data independence**, which means that upper levels are unaffected by changes to lower levels.

Type of Mapping	What does this enable?	Data Independence
Conceptual/Internal	Enables the DBMS to find the actual record or combination of records in physical storage that constitute a logical record in the conceptual schema, together with any constraints to be enforced on the operations for that logical record. Enables any differences in entity names, attribute names, attribute order, data types, and so on to be resolved.	Conceptual/Internal mapping provides Physical Data Independence. **Physical Data Independence** describes the immunity of the conceptual schema to changes in the internal schema. As a result, changes to the internal schema should be possible without having to change the conceptual or external schemas. Examples of changes to the internal schema include changing: • file organisation; • storage structures; • storage devices; • indexes; and • hashing algorithms. From the users' point of view, the only effect that may be noticed is a change in performance. Deterioration in performance is the most common reason for internal schema changes.
	What is this concerned with?	
	Which records in physical storage constitute a logical record in the conceptual schema. This means that physical storage (file systems) are mapped in to tables and other database objects.	
External/Conceptual	**What does this enable?**	External/Conceptual mapping provides Logical Data Independence. **Logical Data Independence** describes the immunity of the external schemas to changes in the conceptual schema. As a result, changes to the conceptual schema should be possible without having to change existing external schemas or having to rewrite application programs. Examples of changes to the conceptual schema include the addition or removal of: • entities; • attributes; and • relationships. From the users' point of view, only those for whom the changes have been made should be aware of the changes. Other users should remain unaware of the changes.
	Enables the DBMS to map names in the user's view to the relevant part of the conceptual schema.	
	What is this concerned with?	
	How the conceptual schema is presented in the external schema: • fields can have different data types; • fields and record names can be changed; and • several fields in the conceptual schema can be combined in to a single field in the external schema. There could be several mappings between external schemas and conceptual schemas as there may be multiple external schemas.	

Theory: Introduction to DBMS

Diagram: Mappings in the ANSI-SPARC three-level architecture



Theory: Introduction to DBMS

Conceptual level: tablespaces and datafiles

Conceptual schemas

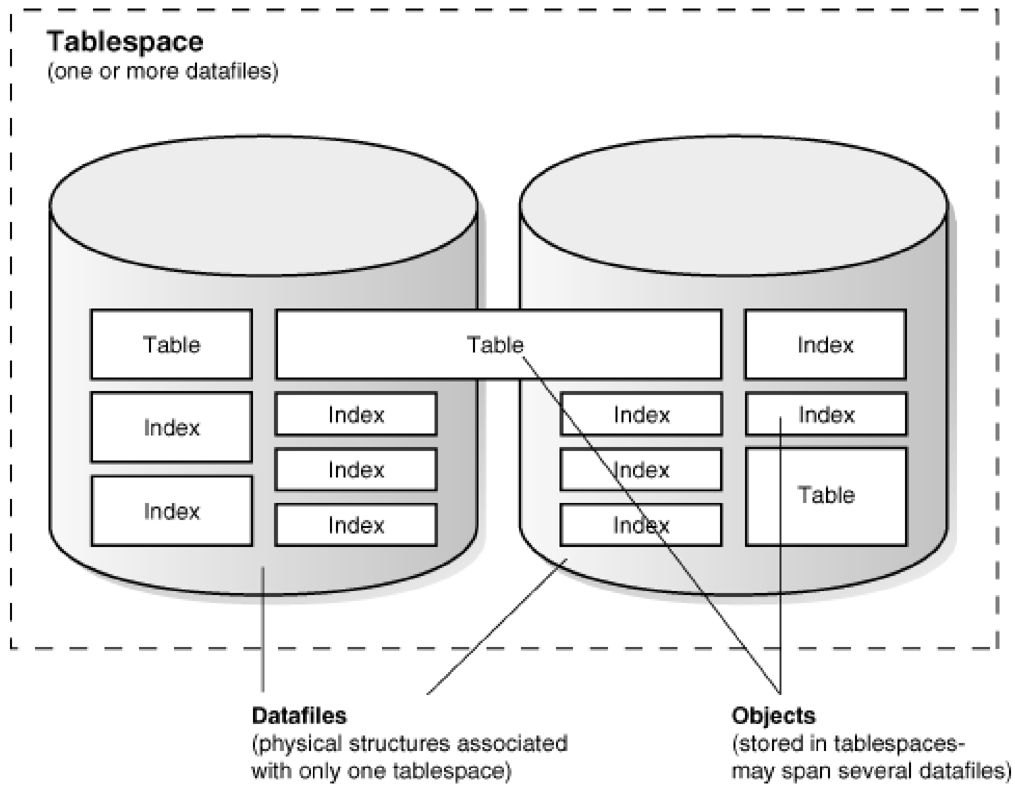
A **user schema** is a schema owned by each database user. It contains schema objects owned by a user such as tables, indexes, triggers, stored procedures etc.

Tablespaces store segments. Each **segment** stores data for schema objects.

Mapping

The internal datafiles are mapped in to the tablespaces; data is stored logically in tablespaces and physically in datafiles that are associated with the corresponding tablespace.

Diagram: Application program connection to DBMS



Typical tablespaces include:

- SYSTEM – holds all of the system tables that describe the contents of the database;
- SYSAUX – holds of non-system-related tables and indexes that traditionally were placed in the SYSTEM tablespace.
- USERS – database users;
- UNDOTBS – stands for undo tablespace and supports `ROLLBACK` functionality;
- TEMP – support applications that work with the database on a temporary storage basis; and
- EXAMPLE – example schemas provided by Oracle.

These tablespace names are usually the same as the names of their corresponding datafiles.

Tablespaces are stored in one or more datafiles as they may have a fixed size. Once this has been exhausted, another datafile may be added and mapped to the tablespace. However, on the conceptual side, this would be transparent.

Theory: Introduction to DBMS

Internal level: pages

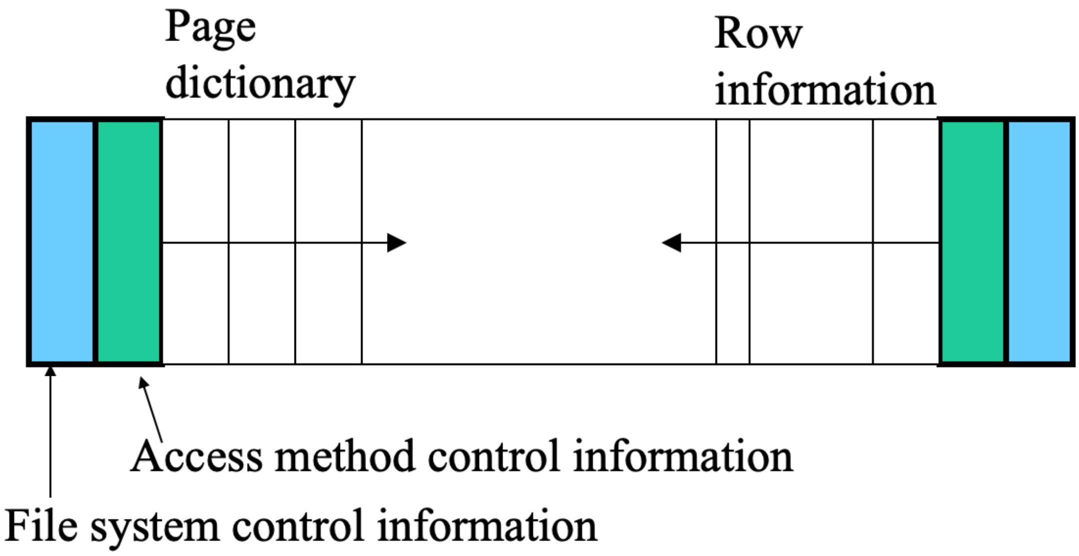
Definition

A **page** is a unit of the amount that can be read from a datafile and put in to local memory (RAM).

Page structure

A datafile in memory is split in to pages. Typically, more than one page is read at a time; instead, a chunk is taken.

Diagram: Page structure



As shown in the diagram above, a page contains the following:

- row information – includes columns, tables, data entries etc.; and
- page dictionary – contains the access method control information and file system control information.

The access method control information includes:

- a pointer to the previous page (left); and
- a pointer to the next page (right).

The file system control information contains:

- includes the type of operating system (OS) is running;
- whether the page should be accessed sequentially (indexed data) or directly (data in different data structure);
- to which table the data in the page belongs; and
- where the row starts/ends.

The page dictionary and row information are expanding sections of the page structure. They expand inwards until the datafile's size is exhausted.

Example

Example: Page file for the `Branch` table

```
Line 1 | INTERNAL (prefix stores flags, pointers):
Line 2 | Stored Branch BYTES=22
Line 3 | PREFIX       BYTES=6, OFFSET=0
Line 4 | branchName   BYTES=6, OFFSET=6, INDEX=BranchX
Line 5 | branchCity   BYTES=6, OFFSET=12
Line 6 | assets       BYTES=4, OFFSET=18
```

The example above shows a portion of a page file that is responsible for storing information regarding the `Branch` table.

- **`Line 2`** – Specifies the `Branch` table and the length of the information regarding the `Branch` table in the page file.
- **`Line 3`** – Specifies that six bytes are used for the `PREFIX`, which contains information that describes the contents of the page file, and that this information will start zero bytes after the start of this section of the page file.
- **`Line 4`** – Specifies that six bytes are used for storing the data in the `branchName` column, the `branchName` column has index `BranchX`, and that this information will start six bytes after the start of this section of the page file.
- **`Line 5`** – Specifies that six bytes are used for storing the data in the `branchCity` column, and that this information will start 12 bytes after the start of this section of the page file.
- **`Line 6`** – Specifies that four bytes are used for storing the data in the `assets` column, and that this information will start 18 bytes after the start of this section of the page file.

The bytes used is dependent on the data type used for a specific column. As the data types are of fixed size, the records are also of fixed size.

Theory: Introduction to DBMS

Structure of a DBMS

Components of a DBMS

A DBMS is comprised of:
- a Data Definition Language (DDL);
- a Data Manipulation Language (DML); and
- control mechanisms.

Data Definition Language (DDL)

Data Definition Language (DDL) is used for operations on the data model and contains a category of statements which are used to define the database structure of schema.

This is used to:
- write the schema;
- describe data items to be stored and their relationships;
- create tables;
- define primary and foreign keys; and
- define validation rules.

Statements	Operates on
• `CREATE` • `ALTER` • `DROP`	• Tables • Views • Stored procedures/functions • Triggers

Data Manipulation Language (DML)

Data Manipulation Language (DML) is used for data storage and retrieval and contains a category of statements which are used for managing data within schema objects.

This is used to:
- access, query, sort and search data;
- insert data into tables; and
- update and delete data.

Statements	Operates on
• `SELECT` • `INSERT` • `UPDATE` • `DELETE`	• Tables • Views

Theory: Introduction to DBMS

Control mechanisms

Control mechanisms in a DBMS include:

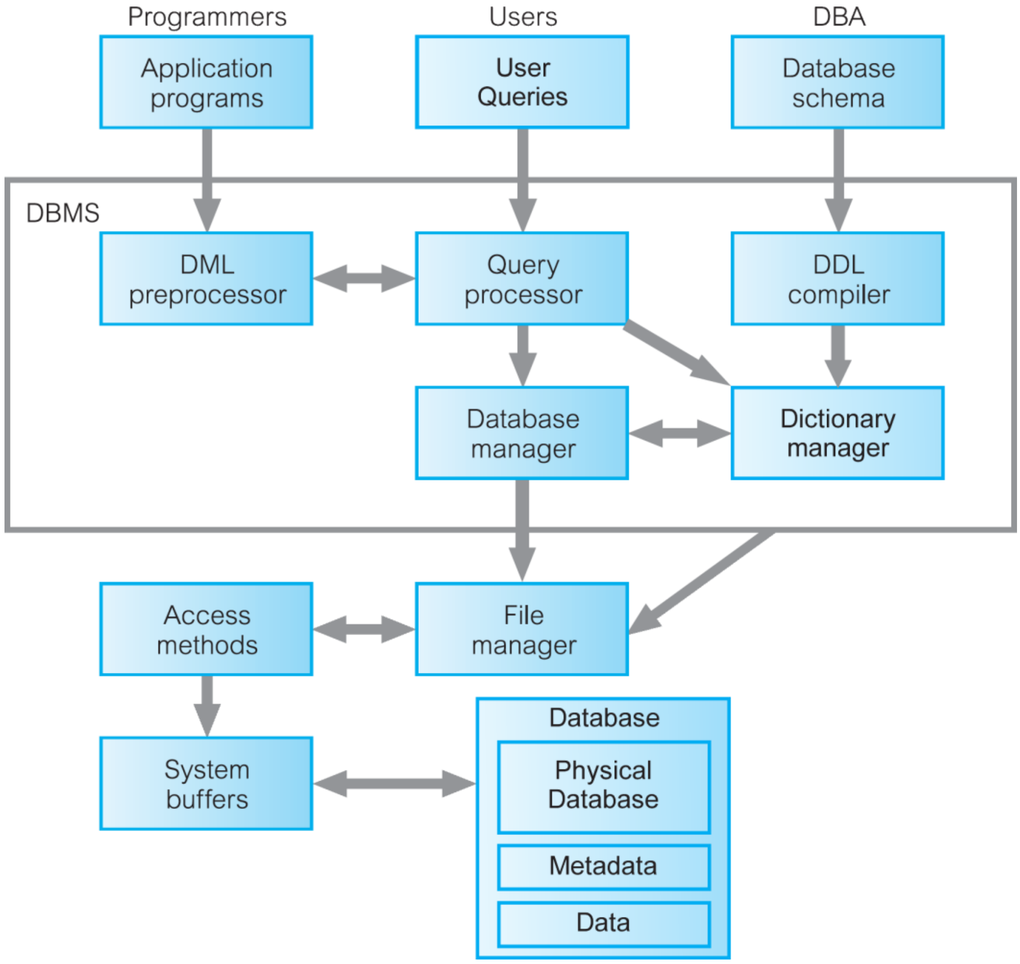
- a transaction model;
- data consistency maintenance;
- recovery;
- backup;
- archiving;
- secondary storage management;
- access control;
- security; and
- distribution management in a network.

DBMS architecture

It is not possible to generalize the component structure of a DBMS, as it varies greatly from system to system. However, it is useful when trying to understand database systems to try to view the components and the relationships between them.

Some of DBMS functions are supported by the underlying operating system. However, the operating system provides only basic services; the DBMS must be built on top of the operating system. Therefore, the design of a DBMS must take into account the interface between the DBMS and the operating system.

Diagram: DBMS architecture



Theory: Introduction to DBMS

The diagram above shows the three groups of people who interface with software components:

- programmers – application programs (such as APEX or other programs written in C++/PHP);
- users – queries (using DML);
- database administrators (DBA) – database schema (using DDL).

The DBMS interfaces with these software components using a range of components.

Component	Role
DML preprocessor	Converts DML statements embedded in an application program in to standard function calls in the host language. Ensures that the query is valid, and that table names and column names are correct.
Query processor	Transforms queries into a series of low-level instructions.
DDL compiler	Converts DDL statements into a set of tables containing metadata.
Database manager (DM)	Interfaces with user-submitted application programs and queries. It accepts queries and examines the external and conceptual schemas to determine what conceptual records are required to satisfy the request. This includes the: • transaction manager; • scheduler (for concurrent operations); • query optimiser; • buffer manager; and • recovery manager.
Dictionary manager	Coordinates all of the tasks in the DBMS by managing and querying the metadata in the data dictionary. Manages data dictionary views, such as: • `USER_TABLES`; and • `USER_INDEXES`. This enables access to names of: • tables; • indexes; • columns; • constraints; and • triggers.
File manager	Manipulates the underlying storage files and manages the allocation of storage space on disk. Establishes and maintains the list of structures and indexes defined in the internal schema and represents a mapping from the conceptual to the internal schemas.

Theory: Introduction to DBMS

Using these components, typical interactions between groups of people and the database can be inspected more closely.

Group of People	Programmers	Users	Database Administrators (DBA)
Software Component	Application programs	User queries	Database schema
Stage 1	**Application Programs <-> DML Preprocessor** The DML query is passed from the **application program** to the **DML preprocessor**.	**User query -> Query processor** The **user query** is passed to the **query processor**.	The DDL query is passed from the **database schema** to the **DDL processor**.
Stage 2	The **DML preprocessor**: • ensures that the query is valid; and • converts the DML statements in to standard function calls in the host language.	**Query Processor <-> DML preprocessor** The **DML processor**: • ensures that the query is valid; and • converts the DDL statements in to standard function calls in the host language.	The **DDL processor**: • ensures that the query is valid; and • converts the DDL statements in to standard function calls in the host language.
Stage 3	**DML Preprocessor <-> Query Processor** The **DML preprocessor** interacts with the **query processor** to generate the appropriate code.	-	-
Stage 4	**Query Processor -> Dictionary Manager** The **query processor** accesses the **dictionary manager** to check if the names of database objects (tables, indexes etc.) actually exist.	-	**DDL processor <-> Dictionary Manager** The **DDL processor** accesses the **dictionary manager** in order to add or remove database objects (tables, indexes, etc.).
Stage 5	**Query Processor -> Database Manager** The **query processor** passes the code to the **database manager**.	**Query Processor -> Database Manager** The **query processor** passes the code to the **database manager**.	**Query Processor -> Database Manager** The **query processor** passes the code to the **database manager**.
Stage 6	**Database Manager <-> Dictionary Manager** The **database manager** may make further validity checks to the **dictionary manager**.	**Database Manager <-> Dictionary Manager** The **database manager** may make further validity checks to the **dictionary manager**.	**Database Manager <-> Dictionary Manager** The **database manager** may make further validity checks to the **dictionary manager**.
Stage 7	**Database Manager -> File Manager** The **database manager** sends the code to the **file manager**.	**Database Manager -> File Manager** The **database manager** sends the code to the **file manager**.	**Database Manager -> File Manager** The **database manager** sends the code to the **file manager**.
Stage 8	The **file manager** determines which file should be written to or read from.	The **file manager** determines which file should be written to or read from.	The **file manager** determines which file should be written to or read from.
Stage 9	**File Manager <-> System Buffers** The **file manager** pushes page(s) from the file in to the **system buffer** using specific access methods (file management techniques for storing and retrieving data).	**File Manager <-> System Buffers** The **file manager** pushes page(s) from the file in to the **system buffer** using specific access methods (file management techniques for storing and retrieving data).	**File Manager <-> System Buffers** The **file manager** pushes page(s) from the file in to the **system buffer** using specific access methods (file management techniques for storing and retrieving data).
Stage 10	**System Buffers <-> Database** The contents of the **system buffer** eventually go in to the **database**.	**System Buffers <-> Database** The contents of the **system buffer** eventually go in to the **database**.	**System Buffers <-> Database** The contents of the **system buffer** eventually go in to the **database**.

Theory: Introduction to DBMS

Types of DBMS

Overview

Type	Description	Products
Relational DBMS (RDBMS)	Tables of data. Introduced with SQL-92 (1992).	• MySQL (Oracle) • SAP ASE (formerly Sybase) (SAP) • TerraData (TerraData)
Object-Relational DBMS (ORDBMS)	Tables of data that allow user-defined types and provides extensions to typical RDMS. User-defined object data types have functions as well as data Introduced with SQL-99 (1999).	• Oracle (Oracle) • SQL Server (Microsoft) • DB2 (IBM) • Informix (IBM) • PostgreSQL (open source)
Object-Oriented DBMS (OODBMS)	Allows user-defined objects to be stored. These more closely resemble real world entities. These can be found in use at airports and CERN.	• ObjectDB (ObjectDB) • Objectivity/DB (Obectivity) • FastObject (Versant) • Versant OD (Versant) • db40 (Versant) • Ozone (open source) • Object Store (Ignite Technologies) • Gemstone (GemTalk Systems)
Native-XML DBMS	Databases designed to use the structure and query language of XML.	

User-defined object data types that have functions as well as data. For example, a data type “circle” may have functions such as `calculateArea` and `calculateCircumference` etc.

Object types can be composed of other data types. For example, a data type “square” may be composed of many “point” data types.

Object types can also be in hierarchical, inheritance structures. For example, a “rectangle” data type may inherit some attributes or functions from a “square” data type as they are likely to share similarities.

Theory: Introduction to DBMS

Revenue of database companies

Statistics

The table below shows the Worldwide 2006-2009 vendor revenue estimates for RDBMS software based on total software revenue.

Company	2006 (millions, $)	2007 (millions, $)	2008 (millions, $)	2008 Market share (%)	2006-2008 Growth (%)
Oracle	7116.3	8161.4	8901.3	43.5	9.1
IBM	3500.9	3966.1	4442.0	21.7	12.0
Microsoft	3052.0	3478.0	4000.0	19.5	15.0
TerraData	558.0	630.0	653.6	3.2	3.7
Sybase	492.2	546.1	617.5	3.0	13.1
Progress Software Corp.	237.5	245.9	238.3	1.2	-3.1
Fujitsu	197.3	181.7	205.6	1.0	1
SAS	132.9	141.2	109.5	0.5	-22.5
Netezza Corp.	44.1	70.3	109.1	0.5	55.2
Hitachie	66.9	62.0	70.9	0.3	14.4
Other vendors	404.0	1712.5	715.9	8.6	
Total	16379.7	17502.0	20479.3	100.0	10.7

Source: IDC, June 2009

The revenue of companies producing software for databases is generally higher than the revenue of companies who produce other types of software due to the importance of database software.

Key points

- Each of the major three vendors continue to dominate their particular platform; Oracle on Unix and Linux, Microsoft on Windows, and IBM on the zSeries.
- Unix and Windows Server were still the leading RDBMS operating system (OS) in 2006 with 34.8% and 34.5% percent market share respectively.
- Linux was the third most popular RDBMS operating system (OS) with 15.5% market share, but it continued to dominate in terms of OS growth, with 67% growth over 2005.
- Gartner has moved to measure market share in terms of total software revenue which includes revenue generated from new license, updates, subscriptions and hosting, technical support and maintenance. Professional services and hardware revenue are not included in total software revenue.
- IDC, a market research firm, reported global 1999 sales revenue of $11.1 billion for relational and object-relational databases, but only $211 million for OO databases.
- TeraData specialises in data warehousing and analytic applications.
- Investment banking is one of Sybase's largest client bases.

Theory: Introduction to DBMS

Business intelligence and data analytics

OLTP

Online Transaction Processing (OLTP) systems are designed to maximise transaction processing. In these systems, it is typical for the data to be dynamic and only the current value be stored.

Examples of use cases for OLTP systems include e-commerce websites, such as Amazon.

OLAP

Online Analytical Processing (OLAP) systems are designed to analyse data using complex, multidimensional views to access and forecast data. These systems have SQL extensions for aggregating data and performing analytics.

Example of use cases for OLAP systems include applications in:
- scientific research; and
- weather forecasting.

Data mining

Data mining is the process of discovering new patterns and relationships in data by using statistical and artificial intelligence (AI) methods.

Example of use cases for data mining include:
- predictions on shopping habits for customers who have shared their information through programs such as Tescos' Clubcard; and
- providing personalised experiences for users on websites, such as Google and Amazon, based on patterns in their searches and visits to web pages.

Data warehouse

A **data warehouse** is a relational database that is designed for query and analysis rather than for transaction processing. It usually contains historical data derived from transaction data but can include data from other sources. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources.

Data warehousing is the idea of storing integrated data collections that can be used for support decision making. This practice typically involves storing largely static, historic data.

In addition to a relational database, a data warehouse environment can include an extraction, transportation, transformation, and loading (ETL) solution, online analytical processing (OLAP) and data mining capabilities, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users."

Data warehousing typically supports data mining.

Oracle supports data warehouses, OLAP and data mining.

Theory: Introduction to DBMS

RDBMS

Definition

A **Relational Database Management System (RDBMS)** is a DBMS designed specifically for relational databases; RDBMS is a subset of DBMS.

People involved

Group of People	Level	Role
End users	External	End users can be split in to two main categories: • naïve – such as a checkout assistant at a supermarket; and • sophisticated – such as a programmer using SQL.
Application Programmers	External	Creates programs that request the RDMS to perform some operation on the database.
Database Designers	Logical / Physical	Interact with users to define the database at all levels. They interact on both logical and physical levels with the RDBMS: • logical – relationships between data; and • physical – mapping between logical design and tables.
Database Administrators (DBA)	Physical	Physical design, implementation and management of databases. Physical design and implementation of the database includes: • schemas; • views; • authorisation; • indexes; and • tuning parameters (for performance). Management of the database includes ensuring that practices are in place for: • uptime; • recovery; • security; and • availability.
Data Administrators (DA)	Logical	Management and logical design of data resource. This is a high-level management role and involves defining: • standards; • policies; and • procedures.

Theory: Introduction to DBMS

Evaluation

Advantages	Disadvantages
Several RDBMS in use and therefore many options available for end users and organisations.	**No user-defined types** when compared to ORDBMS however this avoids the overhead of implementation. If a user should desire user-defined types, ORDBMS solutions are available.
Stable as they are based on mathematics and therefore users understand their structure and the queries can be taken and implemented in algorithms very easily.	**No complex data types or inheritance** when compared to ORDBMS however this avoids the overhead of implementation. If a user should desire complex data types and/or inheritance, ORDBMS solutions are available.
Flexible as new tables can be added easily.	
Well suited for data structured in tables and they are easy to handle.	
Well establish technology in the marketplace and so a large amount of support and expertise is available.	

Practical: Introduction to DBMS

Referential integrity in DBMS

COMMIT and ROLLBACK

The following examples are likely to alter the data in the tables. Oracle's DBMS supports functionality that will allow changes to the table data to be undone:

- the command `COMMIT;` will save the state of the table data for subsequent rollbacks; and
- the command `ROLLBACK;` will revert the state of the table data to that of the most recent commit.

It is important to note that the command `exit;` will perform a commit before exiting the client.

Examples

Example: Updating a row	
Update the row in table `Branch` that has `branchName = 'RoyalBank'` so that the `branchName` value is set to `'Royal'`.	
SQL Statement	**Output**

`UPDATE Branch`
`SET branchName = 'Royal'`
`WHERE branchName = 'RoyalBank';`
	`ERROR at line 1:`
	`ORA-02292: integrity constraint (SYSTEM.BORTOBR) violated - child record found`

Explanation	
The update is not allowed. The "parent" row in `Branch` has "child" rows in table `Deposit` for `branchName = 'RoyalBank'`; the branch `RoyalBank` is a primary key to many foreign key `RoyalBank` in the `Deposit` table.	

Example: Updating rows	
Update rows in table `Deposit` that have `branchName = 'RoyalBank'` so that the branchName values are set to `'Royal'`.	
SQL Statement	**Output**

`UPDATE Deposit`
`SET branchName = 'Royal'`
`WHERE branchName = 'RoyalBank';`
	`ORA-02291: integrity constraint (SYSTEM.DEPTOBR) violated - parent key not found`

Explanation	
The update is not allowed. The "child" rows in table `Deposit` cannot have their `branchName` altered to a "parent" row value that does not exist in table `Branch` as this would break the foreign key integrity. The branch `Royal` does not exist in the `Branch` table. The `branchName` in `Deposit` is a foreign key to the `branchName` in `Branch`, so for any foreign key value, a matching primary key must exist.	

Practical: Introduction to DBMS

Example: Updating rows

Update rows in table `Deposit` that have `branchName = 'RoyalBank'` so that the branchName values are set to `'HFE'`.

SQL Statement	Output

`UPDATE Deposit`
`SET branchName = 'HFE'`
`WHERE branchName = 'RoyalBank';`
	`2 rows updated.`

Explanation

The update is allowed. The child rows in the `Deposit` table with `branchName = 'RoyalBank'` have become child rows to `branchName = 'HFC'`.

The branch `HFE` exists in the `Branch` table.

Example: Deleting a row

Delete the row in table `Branch` that has `branchName = 'RoyalBank'`.

SQL Statement	Output
`DELETE FROM Branch` `WHERE branchName = 'RoyalBank';`	`1 row deleted.`

Explanation

The delete is allowed. The delete in the "parent" row is cascaded to the "child" rows in table `Deposit` with `branchName = 'RoyalBank'`.

The columns `customerName` and `branchName` in both the `Deposit` and `Loan` tables have constraints that are set to `ON DELETE CASCADE`. As a result, when a value is deleted from the primary key table, any child records / foreign keys also will be deleted. This maintains referential integrity and stops violation of foreign key constraints.

Example: Deleting rows

Delete rows in table `Deposit` that have `branchName = 'RoyalBank'`.

SQL Statement	Output
`DELETE FROM Deposit` `WHERE branchName = 'RoyalBank';`	`2 rows deleted.`

Explanation

The delete is allowed. The "parent" row with `branchName = 'RoyalBank'` in table `Branch` no longer has "child" rows in table `Deposit`. The `Deposit` table only contain foreign keys and are therefore "child" rows of the `branchName` in the `Branch` table. This means that there are no other rows that depend on these rows so they can be deleted without violating foreign key constraints.

This could cause problems if the effect is not understood. However, this is the best option as other solutions may prevent data from `Deposit` being deleted without first deleting the primary key in the "parent" table Branch.

ON DELETE actions

As discovered on page 18, it is possible to change the action taken by the DBMS when a primary key is deleted.

Response Mode	Syntax	Description
CASCADE	CONSTRAINT <constraint name> FOREIGN KEY (<column(s)>) REFERENCES <table name> (<column name>, …, <column name>) ON DELETE CASCADE	When referenced data in the parent table is deleted, all rows in the child table that depend on those values in the parent table have are also deleted.
SET NULL	CONSTRAINT <constraint name> FOREIGN KEY (<column(s)>) REFERENCES <table name> (<column name>, …, <column name>) ON DELETE SET NULL	When referenced data in the parent table is deleted, all rows in the child table that depend on those values in the parent table have their foreign keys set to null.
SET DEFAULT	CONSTRAINT <constraint name> FOREIGN KEY (<column(s)>) REFERENCES <table name>(<column name>, …, <column name>) ON DELETE SET DEFAULT	??
No action	CONSTRAINT <constraint name> FOREIGN KEY ((<column(s)>) REFERENCES <table name>(<column name>, …, <column name>)	No action will occur.

The impact of referential action taken by the DBMS when using different SQL statements is shown below.

SQL Statement	Response Mode	Impact on "Parent" Table	Impact on "Child" Table
INSERT	Any	Always OK if the parent key value is unique.	OK only if the foreign key value exists in the parent key or is partially or all null.
UPDATE	No action	Allowed if the statement does not leave any rows in the child table without a referenced parent key value.	Allowed if the new foreign key value still references a referenced key value.
DELETE	CASCADE	Always OK.	Always OK.
	SET NULL	Always OK.	Always OK.
	No action	Allowed if no rows in the child table reference the parent key value.	Always OK.

Theory: Query Optimisation

Physical design

Definition

Physical design is the process of deciding how to store relations and access data.

Objectives

The main objective of successful physical design is to provide an acceptable data access speed.

In order to optimise query processing, the designer must select the most appropriate storage structure supported by the RDBMS.

Physical design is a complex and technically demanding task.

Theory: Query Optimisation

Query processing

Definition

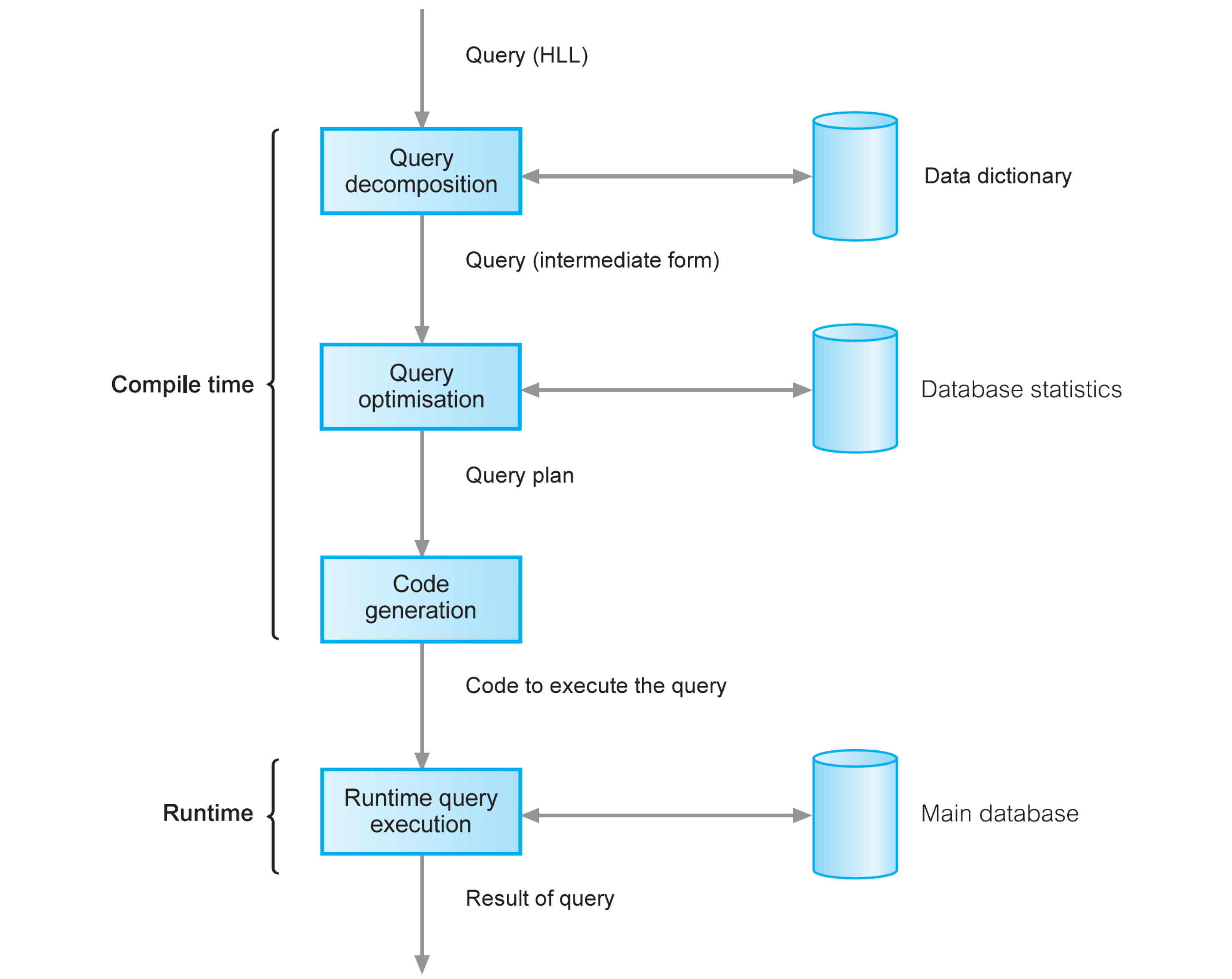
Query processing is the activities involved in parsing, validating, optimising, and executing a query.

Overview

Query processing can be divided into four main phases:

- query decomposition;
- query optimisation;
- code generation; and
- runtime query execution.

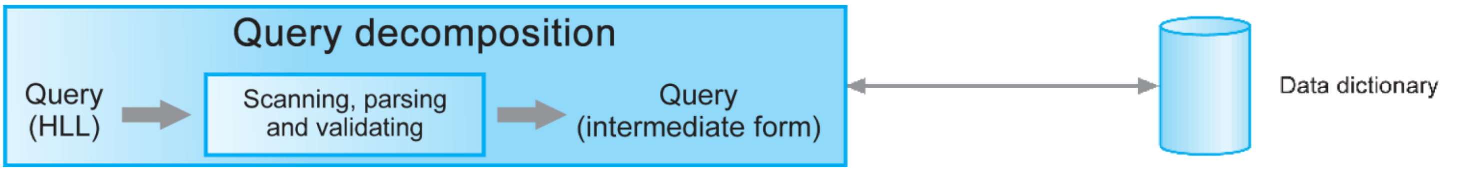
Diagram: Query processing



Theory: Query Optimisation

Query decomposition

Diagram: Query decomposition

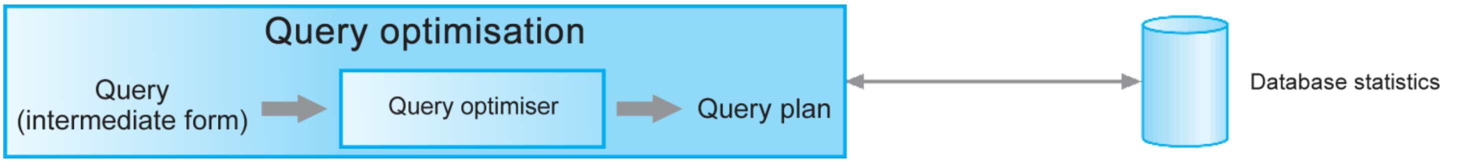


Query decomposition is a compile time stage of query processing where a query written in a high-level language, typically SQL, is scanned, parsed and validated. The result is a query in an intermediate form.

- Scanner – This identified the language components in the text of the query.
- Parser – This checks the query syntax to determine whether it is formulated according to the syntax rules (grammar) of the language.
- Validation – This process determines if all attribute and relation names are valid and semantically meaningful.

Query optimisation

Diagram: Query optimisation

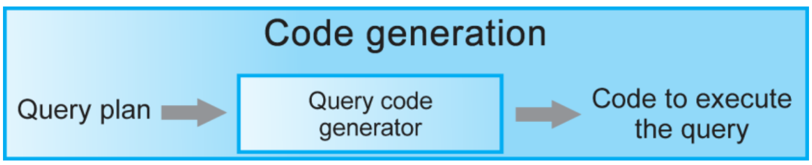


A **query tree** is the internal form of a query.

Query optimisation is a compile time stage of query processing where the query optimiser compares a number of query plans, which are strategies, for retrieving the result of the query from the internal database files. This is achieved by using current database statistics from the data dictionary.

Code generation

Diagram: Code generation

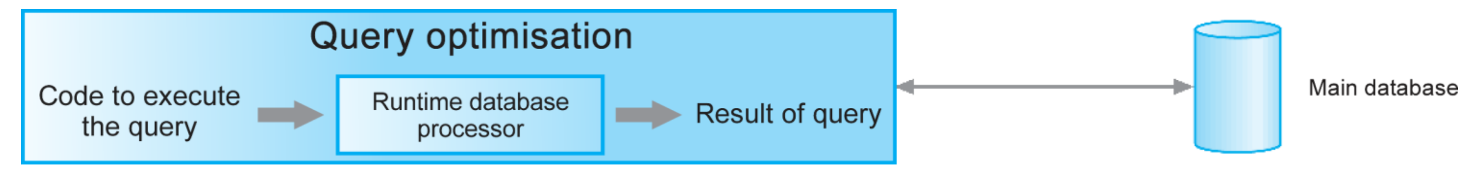


Code generation is a compile time stage of query processing where the query code generator generates the code to execute the query.

Theory: Query Optimisation

Runtime query execution

Diagram: Runtime query execution



Runtime query execution is a runtime stage of query processing where the runtime database processor runs the query code.

If a runtime error occurs, an error message is generated.

The chosen query plan is likely to be a near-optimal strategy; finding the optimal strategy may be time-consuming, especially for complex queries.

Theory: Query Optimisation

Query speed

Impact of file structure and access

The access speed of a query depends on the number of read and write operations to secondary memory. This is because read and write operations to secondary memory are much slower than operations taking place on the CPU.

Example

The query below assumes 400 customers, 1000 deposit accounts and ten balances greater than 10000.

```
SELECT c.customerName, street, customerCity
FROM Customer c, Deposit d
WHERE c.customerName = d.customerName
AND balance > 10000;
```

The table below shows the operations that must take place on the database server and the respective costs of reading data from secondary memory.

Operation	Read Cost
Join `Customer` and `Deposit` tables	Read 400x1000 rows, creating a temporary table of 1000 rows.
Select balances greater than 10000	Read 1000 rows, creating a temporary table of 10 rows.
Project customer details	Read 10 rows and output to screen.

When completing the operations in this order, a total cost of 401010 reads is derived.

The table below shows the same operations that must take place on the database server and the respective costs of reading data from secondary memory, however the order of operations has been changed.

Operation	Read Cost
Select balances greater than 10000	Read 1000 rows, creating a temporary table of 10 rows.
Join `Customer` and `Deposit` tables	Read 400x10 rows, creating a temporary table of 10 rows.
Project customer details	Read 10 rows and output to screen.

With this alternative order of operations, a total cost of 5010 reads is derived.
This shows that the order of operations is important for improving the speed of query, as observed with a $\sim 80\%$ improvement shown above ($\frac{401010}{5010} = 80.04\ (2\ d.p.)$).

As a result, when writing a set of queries, it is important to be aware of the costs of each operation. This can be achieved by using query optimisation strategies; the example above demonstrates operation ordering using heuristics.

Theory: Query Optimisation

Query optimisation strategies

Definition

Query optimisation is the activity of choosing an efficient execution strategy for processing a query. This is necessary to achieve good performance as algorithms are used that implement relational operators, such as `SELECT` and `JOIN` statements, but each has a cost in terms of read and write operations to secondary memory.

Cost estimation

Cost estimation is a query optimisation strategy that involves choosing the query plan (strategy) with the lowest execution cost.

In order to compare query plans, the query optimiser estimates a cost for each query plan (strategy). The query plan (strategy) with the lowest cost is chosen.

The cost is usually a measure of how long each query plan (strategy) will take to execute in terms of input/output (I/O) operations. An approximate cost is sufficient as the query optimiser only has to eliminate poor query plans (strategies) and choose a near-optimal plan.

Heuristics

Heuristics is about using a “rule of thumb” approach in order to find a “good enough” solution.

Using heuristics, query optimisation strategies have been devised that use transformation rules to lower cost by better ordering of algorithms. These include condition selection and operator ordering.

As these query optimisation strategies are based on using heuristics, they mostly provide a near-optimal solution and may not give the best solution. However, the objective is to find a good solution in a reasonable time frame and therefore optimality, completeness, accuracy and/or precision may be traded for speed.

Condition selection

Condition selection is a heuristic rule where the size of the selection for each condition is considered in choosing between simple conditions in a complex query.

As shown in the example on page 115, it is desirable to use the condition the reduces the size of the selection to a minimum first as this will result in less read operations in subsequent operations.

Operator ordering

Operator ordering is a heuristic rule where `SELECT` and project operations are applied before applying `JOIN` operations.

`JOIN` operations generate larger tables, while `SELECT` and project operations reduce the size of a single table. As a result, performing `SELECT` and project operations first will mean that the `JOIN` operations are joining together smaller tables.

Theory: Query Optimisation

File searching algorithms

Definition

A **search algorithm** is any algorithm which solves the search problem to retrieve information stored within some data structure or calculated in the search space of a problem domain, either with discrete or continuous values.

File searching algorithms apply typical searching algorithms to find files.

Linear search

Linear search is a brute force and iterative searching algorithm which sequentially checks each element of the list to see if it matches the search criteria until a match is found or until all the elements have been searched.

A DBMS would use linear search to retrieve every record in the file starting at the beginning.

Binary search

Binary search is a divide and conquer iterative searching algorithm which works by repeatedly dividing in half the portion of a list which contains the required data item until there is only one item in the list. This can also be implemented recursively.

A DBMS would use binary search to access the middle record and subsequently go to the top of bottom section accordingly. This process would be repeated until the selection is found.

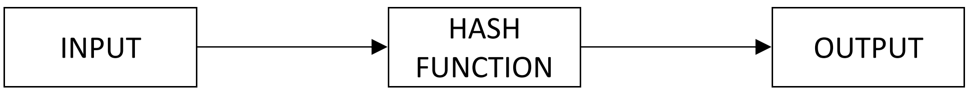
A binary search would be used if the selection condition involves an equality comparison on the key attribute on which the file is ordered, for example a `studentID` field in a `Student` relation.

Hash search

Hash search is an algorithm which provides direct access to a record by using a hash function to calculate its location in memory.

A **hash function** is code that provides a mapping between an arbitrary length input and a fixed length output. This process is one-way and therefore the original input cannot be obtained using the output.

Diagram: Hash function



The input is arbitrary sized, and the output is fixed sized.

Theory: Query Optimisation

Example of file searching algorithms on data

This example considers a sorted file diagram populated with the data shown below.

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

Considering a search for the record where `LotNumber = 18`, below is the stages taken when using a linear search and a binary search.

Linear Search

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

Binary Search

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

LotNumber	Area	Use	UrbanZone
14	900	resid	Re-1
15	1400	resid	Re-1
16	850	resid	Re-1
17	1000	comm	Com-1
18	950	resid	Re-2
19	1500	comm	Com-2
20	1600	resid	Re-2

Comparisons using linear search – 5
Comparisons using binary search – 3

This example shows that, when the data is sorted, a binary search is generally more efficient than a linear search as less comparisons are to be made before the result is found.

Theory: Query Optimisation

Use of file searching algorithms

The table below shows the respective possible access methods (file searching algorithms) for different file structures.

File Structure	Type of Access	Access Method
Heap file	Unordered	Linear search.
Sequential file	Ordered	Binary search on ordering field. Ordered files have to be maintained during `INSERT`, `UPDATE` and `DELETE` operations.
Hash file	Direct	Hash function of hash field used to calculate the memory address.

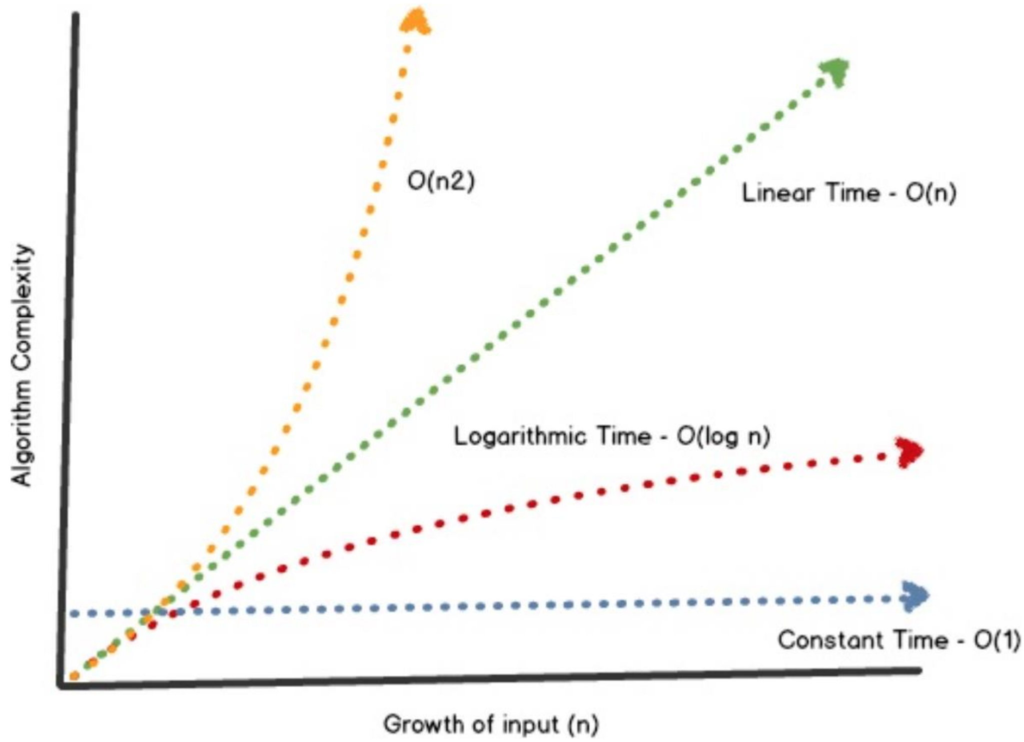
Comparison and evaluation

		Linear Search	Binary Search	Hash Search
Time Complexity	**Average**	$O(n)$	$O(\log(2N))$	$O(1)$
	Worst case	$O(\frac{n}{2})$	$O(\log(2N))$	$O(1)$
Advantages		**Works on all data sets** as no ordering or special treatment of data is required as each data item is checked until the search criteria is found.	**Logarithmic time complexity** means that the growth of the time complexity will decrease as the number of data items increases.	**Constant time complexity** allowing direct access to data and no increase in time complexity following an increase in the number of data items.
Disadvantages		**Linear time complexity** means that the growth of time complexity will be directly proportional to the increase in the number of data items.	**Only works when the selection condition involves an equality comparison on the key attribute on which the file is ordered** as the data must be somehow ordered to allow the binary search algorithm to accurately calculate the midpoint of the data.	**May be inefficient** for ranges of values or pattern matching.
				Collision management is required if the same memory address location is generated for two or more records.

Theory: Query Optimisation

It is possible to evaluate the time complexity of each file searching algorithm when taking in to account the size of the data and how the time complexity of each algorithm grows with the number of items in the data.

Diagram: Time complexity graph



n represents the number of data items to be searched.

Using the graph above, it is possible to deduce that as the size of data increases:

- the time complexity of a linear search will increase linearly;
- the time complexity of a binary search will increase logarithmically; and
- the time complexity of a hash search will remain constant.

Theory: Query Optimisation

Indexing

Definitions

Indexes are access structures that are used to speed up the retrieval of rows in a database table.

An **index** is a table of index values and associated address pointers to the records containing the relevant rows.

Types of indexes

A **primary key index** is built on a unique, ordered field.

A **secondary key index** is built on a non-unique, un-ordered field. Secondary indexes provide a mechanism for specifying an additional key for a base relation that can be used to retrieve data more efficiently.

Multilevel indexes involve splitting an index in to a number of shorter indexes. This provides an index to the indexes, known as a B-tree. These may be used when an index is large.

Index information needs to be stored in the database as well as the tables.

Example of an index

Example: Index

Age index

Age	Address
21	*
23	*
23	*
25	*
25	*
25	*
30	*

Student Relation

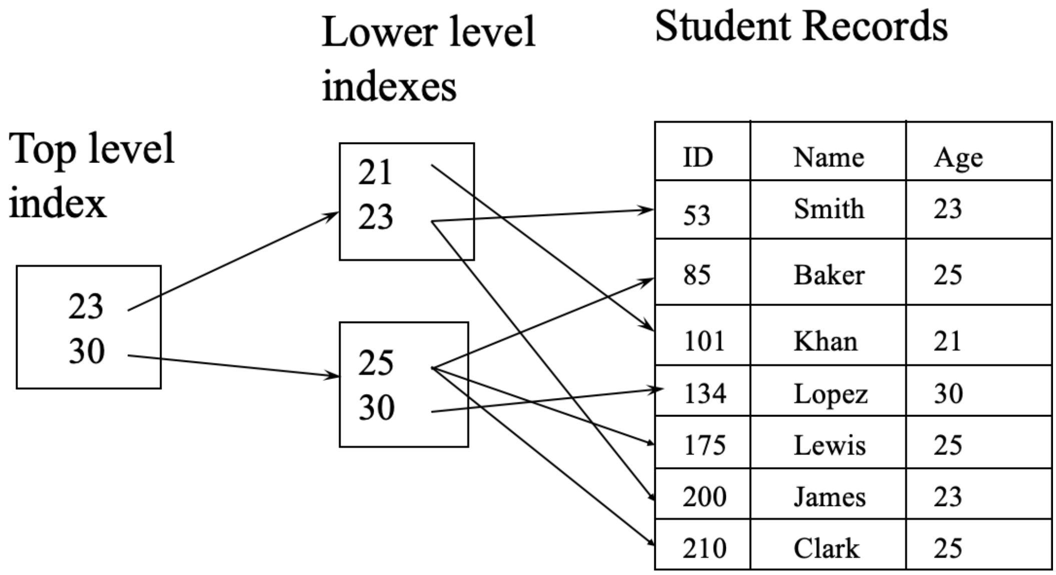
ID	Name	Age
53	Smith	23
85	Baker	25
101	Khan	21
134	Lopez	30
175	Lewis	25
200	James	23
210	Clark	25

In this example, the `Student` relation with an index on `Age` will provide direct access to rows for `Student` who have a required age.

Theory: Query Optimisation

Example of a multilevel index

Example: Multilevel index



In this example, the top level index is searched for an age equal to or greater than the required age. The address pointer is then followed to the lower level index.

Creating an index

Format: Creating an index

```
CREATE [UNIQUE] INDEX <index name>
ON <table name> (<column name> [ASC / DESC]);
```

An index may use more than one column. Different indexes may be created for:

- different combinations of columns in the table; and
- different orders of the same columns.

Example: Creating an index

Create an index named `MyIndex` on the `balance` column in the `Deposit` table, ordered by `DESC`.

SQL statement

```
CREATE INDEX MyIndex ON Deposit (balance DESC);
```

Format: Deleting an index

```
DROP INDEX <index name>;
```

Theory: Query Optimisation

Using an equality

It is possible to use an index to retrieve multiple records. If the comparison condition is any of the following:

- greater than (>);
- greater than or equal to (>=);
- less than (<); or
- less than or equal to (=<)

on a key field with an index, such as `Student.ID < 150`, then use the index to find the record satisfying the condition then retrieve all the preceding records.

For more complex queries, such as `Student.ID < 150 and Student.age > 25`, this will involve two steps to producing a query plan (strategy) in which either of the operations could be carried out first.

Theory: Query Optimisation

Evaluation

Query optimisation strategies

The impact of query optimisation strategies depend on:
- the amount of data being processing;
- the complexity of the query; and
- the DBMS being used.

Indexes

Indexes improve the efficiency of retrieving rows from a database file.

There are overheads when implementing indexes:
- they take up disk space; and
- incur an update overhead as every time a record is updated, the indexes will also have to be updated to remain consistent.

Indexes are usually created to satisfy particular search criteria after the table has been in use for some time and has grown in size. Therefore, it is necessary to consider which columns (attributes) to index based on query usage and characteristics of the DBMS.

Practical: Query Optimisation

Indexes

Example: Creating an index	
Add an index to the `balance` column in table `Deposit`.	
SQL Statement	**Output**
`CREATE INDEX index_balance` `ON Deposit(balance);`	`Index created.`

The example above has added an index to the `balance` column in the `Deposit` table, however there would need to be several hundred rows in the table for the benefit of this index to be realised.

Practical: Query Optimisation

The DATE data type

Example: Adding columns with the DATE data type

Add two more columns to tables Deposit and Loan that will store:

- **the date (and time) an account or loan was started with a default value of the current date (and time); and**
- **the date (and time) an account or loan was last updated.**

SQL Statement	Output
`ALTER TABLE Deposit` `ADD creationDate DATE DEFAULT SYSDATE;`	Table altered.
`ALTER TABLE Loan` `ADD creationDate DATE DEFAULT SYSDATE;`	Table altered.
`ALTER TABLE Deposit` `ADD modificationDate DATE;`	Table altered.
`ALTER TABLE Loan` `ADD modificationDate DATE;`	Table altered.

In the example above, the default value for the `creationDate` column is automatically set to the value of the current date (and time). This is achieved by using the `SYSDATE` function for the `DEFAULT` value.

Example: Trigger to update columns with the DATE data type

Write a trigger on the Deposit and Loan tables that will automatically update the modificationDate to SYSDATE when an update is made on the Deposit and Loan tables.

SQL Statement	Output

`CREATE OR REPLACE TRIGGER setModificationDateDeposit`
`BEFORE UPDATE ON Deposit`
`FOR EACH ROW`
`BEGIN`
`:new.modificationDate := SYSDATE;`
`END;`
`/`
	Trigger created.

`CREATE OR REPLACE TRIGGER setModificationDateLoan`
`BEFORE UPDATE ON Loan`
`FRO EACH ROW`
`BEGIN`
`:new.modificationDate := SYSDATE;`
`END;`
`/`
	Trigger created.

The date syntax for `INSERT` operations is `'DD-MON-YY'` but the `DATE` data type can store time as well using the `TO_CHAR` function to enter the time. For example,

```
SELECT customerName, branchName, accountNumber, balance,
       TO_CHAR(creationDate, 'DD-MON-YYYY HH24:MI:SS')
FROM Deposit;
```

would return the data shown below.

```
                                                         TO_CHAR(creationDate, 'DD-MON-
CUSTOMERNAME  BRANCHNAME  ACCOUNTNUMBER  BALANCE  YYYY HH24:MI:SS')
------------  ----------  -------------  -------  ------------------------------
Jones         Yorkshire   1              121.55   07-JAN-2019 21:12:50
Braun         Midlands    20             150      07-JAN-2019 21:12:50
...           ...         ...            ...
```

Practical: Query Optimisation

Functions in `INSERT` and `UPDATE` operations

Example: Adding columns with the `DATE` data type

Write `INSERT` commands that adds a `Loan` and `Deposit` row for `customerName` and `branchName`.

SQL Statement	Output
`INSERT INTO Deposit(branchName, accountNumber, customerName, Balance) VALUES ('Midlands', 200, 'Jones', 340);`	`1 row created.`
`INSERT INTO Loan(branchName, loanNumber, customerName, amount) VALUES ('Midlands' 234, 'Jones', 440);`	`1 row created.`

For the example above, it is possible to check the date and time of the creation of the new rows added to the `Loan` and `Deposit` tables using the `TO_CHAR` function in a `SELECT` statement:

```
SELECT TO_CHAR(creationDate,'HH24:MI DD MONTH YY')
FROM Deposit;
```

The following data would be returned:

```
TO_CHAR(CREATIONDATE,'HH24:MIDDMONTHYY')
-----------------------------------------------
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
21:53 07 JANUARY   19
21:12 07 JANUARY   19
21:12 07 JANUARY   19
```

In addition, the `TO_DATE` function may be used in an `INSERT` statement to enter dates and times in to a `DATE` data type. For example:

- `TO_DATE('27-OCT-98 11:21:34', 'DD-MON-RR HH:MI:SS');` and
- `TO_DATE('January 15, 1989, 11:00 A.M.', 'Month DD, YYYY, HH:MI A.M.').`

Example: Updating columns with the `DATE` data type

Write an `UPDATE` command that modifies the `balance` and `amount` columns of the new rows created in the previous example.

SQL Statement	Output
`UPDATE Deposit SET Balance = 200 WHERE AccountNumber = 200;`	`1 row updated.`
`UPDATE Loan SET Amount = 234 WHERE LoanNumber = 234;`	`1 row updated.`

For the example above, it is possible to check the date and time of the modification of the rows in the `Loan` and `Deposit` tables using the `TO_CHAR` function in a `SELECT` statement:

```
SELECT TO_CHAR(modificationDate,'HH24:MI DD MONTH YY')
FROM Deposit;
```

The following data would be returned:

```
TO_CHAR(CREATIONDATE,'HH24:MIDDMONTHYY')
-----------------------------------------------
21:56 07 JANUARY   19
21:56 07 JANUARY   19
```

Practical: Query Optimisation

Functions and formatting in output from SELECT operations

Example: Adding columns with the DATE data type

Write a database query on the Deposit table that outputs customer deposit details for each branchName (order by branchName and then customerName).

The output from the query should include an extra output column with alias 'Query DateTime' for the date and time this query was made, as well as all the six deposit column details.

- **Format the output of the date-time columns so that they look like '15:30 12 March 2003'.**
- **Find a function that will replace the branchName 'RoyalBank' by 'RoyalNorthernBank' in the query output.**
- **Format the Balance and Amount columns so that the numbers have two decimal places and commas after every three digits.**
- **Add aliases to the columns containing format functions in order to get suitable output column header names.**

SQL Statement

```
SELECT
      TO_CHAR(SYSDATE,'HH24:MI DD MONTH YY') AS "Query DateTime",
      SUBSTRB(REPLACE(BranchName,'RoyalBank','RoyalNorthernBank'),1,13)
      AS BranchName, CustomerName, AccountNumber,
      TO_CHAR(Balance, '9,999.99') AS Balance,
      TO_CHAR(CreationDate,'HH24:MI DD MONTH YY') AS CreationDate,
      TO_CHAR(ModificationDate,'HH24:MI DD MONTH YY') AS ModDate
FROM Deposit
ORDER BY BranchName, CustomerName;
```

Output

QUERY DATETIME	BRANCHNAME	CUSTOMERNAME	ACCOUNTNUMBER	BALANCE	CREATIONDATE	MODDATE
22:13 07	HFE	Jones	42	4,100.00	21:53 07	
JANUARY	Midlands	Braun	20	150.00	JANUARY 19	
19	Midlands	Jones	61	200.00	"	
"	Midlands	Jones	72	2,000.00	"	
"	Midlands	Jones	200	340.00	"	21:56
"	Midlands	Patel	22	70.00	"	07
"	Midlands	Patel	51	200.00	"	JANUARY
"	Midlands	Smith	50	200.00	"	19
"	Midlands	Smith	21	600.00	"	
"	RoyalNorthern	Ahmed	30	480.00	"	
"	RoyalNorthern	Patel	31	450.00	"	
"	Southern	Braun	41	2,000.00	"	
"	Yorkshire	Jones	1	121.55	"	

In the example above, the TO_CHAR function converts:

- the DATE type to a string using a date format string; and
- a number type to a string using a number format string.

Theory: Data Administration and Security

Data dictionary

Definition

The **data dictionary**, or **data directory** or **system catalog**, contains metadata about the database; data about the data in the database. For example, when a table is created in the database, the data dictionary may store:
- table name;
- column names;
- column data types; and
- constraints (primary keys and foreign keys) etc.

The data dictionary also stores similar metadata for stored procedures/functions, triggers, views, users, transactions and sessions etc.

Views of the data dictionary

The data dictionary provides views of tables of the metadata, rather than providing access to the underlying system tables. These views may be compiled from one or more system tables.

Static data dictionary views

Static data dictionary views only changes when a change is made to the data dictionary, such as a table being added or removed.

The information provided in static data dictionary views is often accessed by management scripts that check, access and make changes to the data. For example, a stored procedure may access information from static data dictionary views in order to find out which users have not accessed a table in a specified amount and subsequently remove their user account for inactivity.

Each set of static data dictionary views are prefixed with either:
- `ALL_` – Only lists the objects that the currently logged in user has permissions to access, this contains information accessible to the current user.
- `USER_` – Only lists the objects owned by the currently logged in user, this contains information from the schema of the current user.
- `DBA_` – Lists all objects unless restricted by the `WHERE` clause, this contains information accessible to the admin user (DBA).

Examples of static data dictionary views include: `USER_TABLES`, `USER_TAB_COLUMNS`, `USER_INDEXES`, `USER_CONSTRAINTS` and `USER_TRIGGERS`.

Dynamic performance views

Dynamic performance views are continuously updated while a database is open and in use, and their contents relate primarily to performance including:
- current memory usage; and
- time taken to perform transactions on different datafiles in different data blocks.

Tools, such as Oracle Enterprise Manager, provide charts to view data populated from dynamic performance views.

Theory: Data Administration and Security

Core administration tasks

Tasks

Core administration tasks are performed by a database administrator (DBA).

These tasks include:

- creating and configuring an Oracle database, using SQL statements or GUI applications that interact with the DBMS, and considering including the following:
  - control files;
  - redo logs;
  - tablespaces;
  - datafiles;
  - temp files; and
  - undo files;
- managing memory requirements, considering whether statically or dynamically allocated memory be used as more datafiles are created;
- managing users and securing the database; and
- managing backups and recovery.

Theory: Data Administration and Security

Database security

Definition

Database security describes the mechanisms that protect the database against intentional or accidental threats and involves limiting the access, or modification of data, to authorised users.

Threats and situations

Threats to database security can be classified by situations, these include:

- theft and fraud – may include theft of the organisation's data or fraud committed using the organisation's data;
- loss of confidentiality (secrecy) – organisation data may no longer remain private, this may allow competitor organisation access to the data or result in loss of data;
- loss of privacy – may lead to non-compliance with the Data Protection Act (1998/2018) as customer data may become publicly accessible;
- loss of integrity;
- loss of availability – could result in loss of customers or productivity, for example an online ordering system may be affected, or a production line is rendered non-operational as it has lost access to the database.

These situations could arise through the actions of malicious users or by other events, such as natural disasters.

Only certain people may be legally authorised to access private or secure information. Companies policies and security provisions:

- may restrict the types of data available to the public and employees; and
- need to prevent malicious attempts to steal or modify data.

Some countermeasures for these threats and situations include:

- controlling users – controls user access to the database through authorisation and authentication;
- access controls –
  - discretionary access control restricts access to objects based on the identity of users and/or groups to which they belong; and
  - mandatory access control restricts access to objects based on specifications provided by the system, rather than the users, clearance and classification data are stored in the security labels, which are bound to the users and objects

  a DBMS will typically use discretionary access control, for example to provide users with access to create a session (connect to the database) or create a table etc.;
- backup and recovery – it is important to consider offsite and duplicated backups in the event that the building is destroyed, such as in a natural disaster;
- integrity – a higher number of good constraints on data in database reduces the chance of user errors and unauthorized access and maintains valid relationships between the data; and
- encryption.

Theory: Data Administration and Security

The table below shows an overview of the threats and situations to a database.

Situation	Threat				
	Theft and Fraud	Loss of Confidentiality	Loss of Privacy	Loss of Integrity	Loss of Availability
Using another person's means of access	✓	✓	✓	X	X
Unauthorised amendment or coping of data	✓	X	X	✓	X
Program alteration	✓	X	X	✓	✓
Inadequate policies and procedures that allow a mix of confidential and normal output	✓	✓	✓	X	X
Wire tapping	✓	✓	✓	X	X
Illegal entry by hacker	✓	✓	✓	X	X
Blackmail	✓	✓	✓	X	X
Creating "trapdoor" into system	✓	✓	✓	X	X
Theft of data, programs and equipment	✓	✓	✓	X	✓
Failure of security mechanisms, giving greater access than normal	X	✓	✓	✓	X
Staff shortages or strikes	X	X	X	✓	✓
Inadequate staff training	X	✓	✓	✓	✓
Viewing and disclosing unauthorised data	✓	✓	✓	X	X
Electronic interference and radiation	X	X	X	✓	✓
Data corruption owing to power loss or surge	X	X	X	✓	✓
Fire (electrical fault, lightning strike, arson), flood, bomb	X	X	X	✓	✓
Physical damage to equipment	X	X	X	✓	✓
Breaking cables or disconnection of cables	X	X	X	✓	✓
Introduction of viruses	X	X	X	✓	✓

Physical level security

Physical level security provides:

- protection of equipment from natural disasters such as floods and power failure etc.;
- protection of disks from theft, erasure, physical damage etc.;
- protection of network and cables from wiretaps, non-invasive electronic eavesdropping, physical damage etc.

The importance of the data in the database is likely to dictate the level of attention and effort given to physical level security. For example, if the loss of data is likely to damage an organisation or a sector of their operations, efforts to increase physical level security will help to prevent business failure in the future.

Solutions for physical level security are described in the table below.

Theory: Data Administration and Security

Solution	What is involved?	Impact on Physical Level Security
Replicated hardware	• Mirrored disks. • Dual busses. • Multiple access paths between every pair of devices.	• Having multiple disks on multiple sites enables backups to be recovered in events, such as natural disasters. • If one access path is down, another is available.

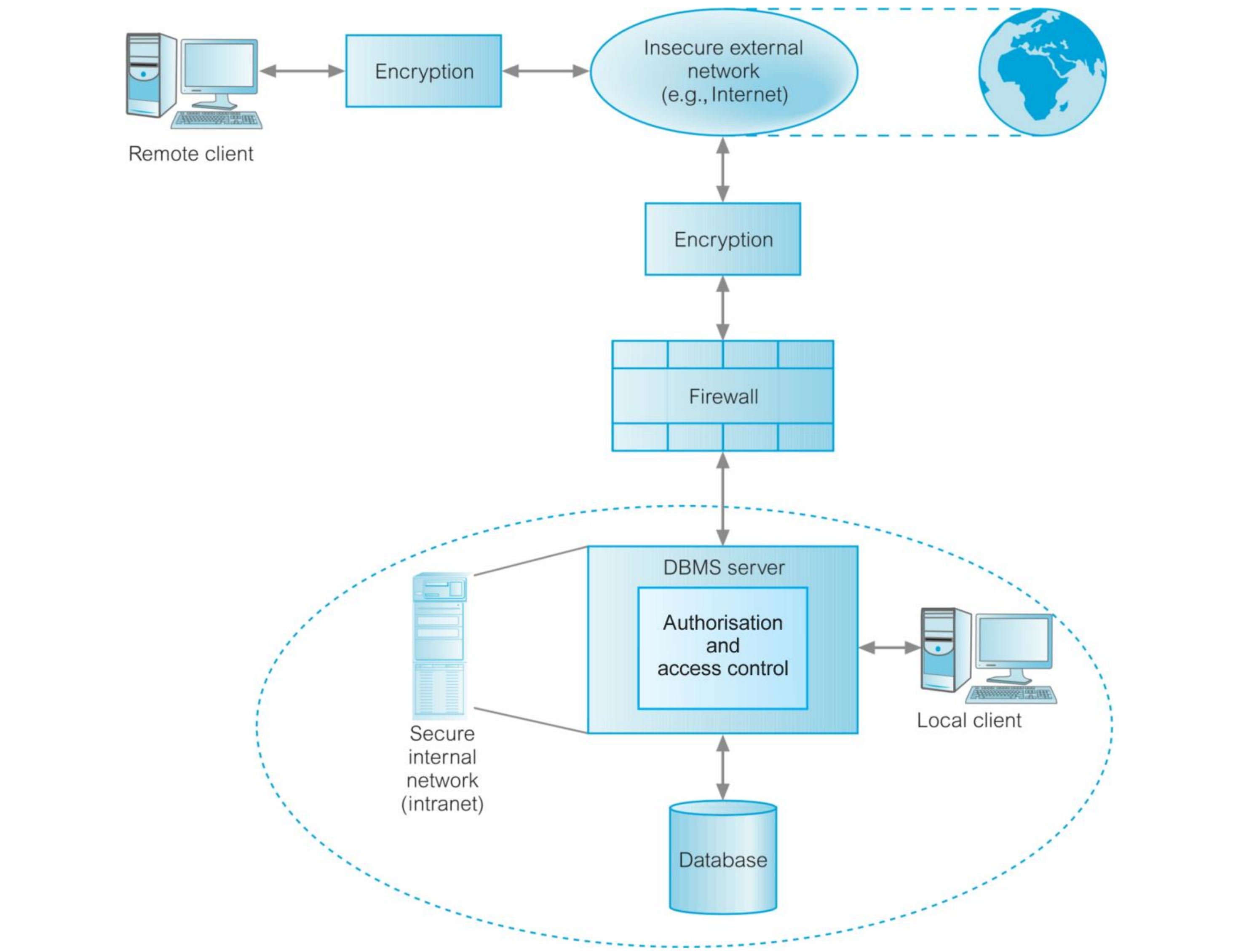
Backup and Recovery	• Backups on separate hard disks.
	• Backups including:
	  ○ database files;
	  ○ control files; and
	  ○ log files (track changes to the database).
		• Offsite backups enable backups to be recovered if the building containing the database servers is damaged.

Physical Security	• Locks on rooms/buildings.	• Prevents theft. • Prevents unauthorized access.

Software Techniques	• Network security including:
	  ○ firewalls; and
	  ○ encryption
		• Helps to detect physical security breaches.

The solutions for physical level security are setup in a multi-user architecture (or multi-user computer environment).

Diagram: Multi-user architecture



Theory: Data Administration and Security

Human level security

Human level security provides protection from stolen passwords, sabotage etc.

Sabotage could include alterations to make programs insecure, such as trapdoors. This is where a user may leave code in scripts, such as stored procedures/functions, to later gain unauthorised access to the database.

Solutions for human level security are to be implemented by management level in an organisation, they include:
- ensuring passwords are frequently changed;
- using "non-guessable" password;
- providing training to prevent careless security breaches, such as writing down passwords on paper;
- logging all invalid access attempts; and
- data audits – setting up the database to audit different users, databases and access to tables.

Operating system (OS) level security

Operating system (OS) level security provides:
- protection from invalid logins, some DBMSes allow the database to be accessed through the operating system (OS) by using OS access controls;
- file-level access protection, however this will provide little protection if an unauthorised user gains admin rights;
- protection from improper use of "superuser" authority; and
- protection from improper use of privileged machine instructions, some machine instructions are not available to normal applications.

Database level security

Database level security assumes that security is implemented at other levels.

Database level security focuses on database specific issues; database access controls based on security policy using:
- authorisation of users – setting up user accounts with usernames
- authentication of users – setting up password control so that every user defined in the database has a user schema protected by a password; and
- user privileges – assigning access permissions to operations in the database, such as the ability to create tables, add data or select data.

- Only some system users need access to the database.
- Each database user may have authority to read only part of the data and to write only part of the data, i.e. entire relations and/or only parts of relations (specific attributes and/or specific rows).
- If database system is distributed, need to decide where control should be; either at the local systems or at a central location.
- Integrity constraints can prevent data from becoming invalid.

More information about database level security can be found in subsequent sections of this book.

Theory: Data Administration and Security

Database level security: user authorisation/authentication, audit trails and encryption

User authorisation/authentication

Each authorised database user must have an account on the database, usually on a user schema, to gain access to the database.

A user logs in to an account with:
- an authorisation identifier (username); and
- a password.

The DBMS provides authorisation and authenticates user logins and maintains logs of:
- all valid access attempts;
- all invalid access attempts; and
- all operations performed by each user.

The DBMS also provides access control (privileges).

Audit trails

An **audit trail** (or **log**) is a record of the history of actions executed by a DBMS showing:
- which user has accessed the database;
- what operations the user has performed;
- the period of time the operations were performed; and
- where (terminal) the operations were performed.

Audit trails are useful for:
- maintaining security if tampering is suspected;
- inspecting performance as it is possible to determine which operations are used most frequently; and
- recovering lost transactions.

Encryption

For	Against
Helps to protect sensitive data, such as passwords, as if an unauthorised user gains access to the encrypted data it may be very difficult to decrypt.	Encryption takes **processing power and time**.
	Is encryption necessary for the context?

Either way, the main objective is to configure database level security in such a way that the data is never accessed anyway.

Theory: Data Administration and Security

Database level security: users and profiles

Creating users

Format: Creating a user

```
CREATE USER <username> IDENTIFIED BY <password>
      DEFAULT TABLESPACE <tablespace name>
      QUOTA <size> on <tablespace name>
      TEMPORARY TABLESPACE <temporary tablespace name>
      PROFILE <profile name>;
```

When creating a user, it is possible to assign the following parameters:

- username – used to authorise the user;
- password – used to authenticate the user;
- `DEFAULT TABLESPACE` – the schema where the user's objects will be stored, usually `users` however a large database may have more than one user schema;
- `QUOTA` – the space allowed on secondary memory to the user in a specified tablespace;
- `TEMPORARY TABLESPACE` – specifies the tablespace for the user's temporary segments, usually `temp`; and
- `PROFILE` – defines other user information, including other parameters, to prevent redundancy in defining user's parameters.

Example: Creating a user

```
CREATE USER jsmith IDENTIFIED BY PvLYSJLGLJKS6H8yBCZxBwpy
      DEFAULT TABLESPACE users
      QUOTA 5M on users
      TEMPORARY TABLESPACE temp
      PROFILE student_small_profile;
```

In the example above, the user has the following parameters:

- username – `jsmith`;
- password – `PvLYSJLGLJKS6H8yBCZxBwpy`;
- `DEFAULT TABLESPACE` – `users`;
- `QUOTA` – `5M` allowed space on `users` tablespace;
- `TEMPORARY TABLESPACE` – `temp`.
- `PROFILE` – `student_small_profile`.

Altering users

Format: Altering a user's password

```
ALTER USER <username> IDENTIFIED BY <new password>;
```

Example: Altering a user's password

```
ALTER USER jsmith IDENTIFIED BY tNXNwL53eAY5nFwJG9TcjBRy;
```

In the example above, the password for the user `jsmith` has been changed to `tNXNwL53eAY5nFwJG9TcjBRy`.

Creating profiles

A **profile** defines other user information, including other parameters, to prevent redundancy in defining user's parameters. Default resource limits are set globally for the database and then specifically for users by assigning profiles to users.

Format: Creating a profile

```
CREATE PROFILE <profile name>
      LIMIT SESSIONS_PER_USER <number (integer)>
      CPU_PER_SESSION <number (1/100th seconds)>
      CPU_PER_CALL <number (1/100th seconds)>
      IDLE_TIME < number (minutes)>
      CONNECT_TIME <number (minutes)>
      LIMIT PASSWORD_REUSE_MAX <number (integer)>
      PASSWORD_REUSE_TIME <number (days)>;
```

When creating a profile, it is possible to assign the following parameters:

- `LIMIT SESSIONS_PER_USER` – limits the number of concurrent sessions a user may have connected to the database;
- `CPU_PER_SESSION` – the CPU time ($1/100^{th}$ seconds) per session;
- `CPU_PER_CALL` – the runtime ($1/100^{th}$ seconds) for a process call;
- `IDLE_TIME` – the amount of time (minutes) before a user is logged out for inactivity;
- `CONNECT_TIME` – the amount of time (minutes) a user can be connected to the database in one session;
- `LIMIT PASSWORD_REUSE_MAX` – limits the number of times the password must be changed before the same password can be used again; and
- `PASSWORD_REUSE_TIME` – the amount of time (days) before a password can be reused.

Example: Creating a profile

```
CREATE PROFILE student_small_profile
      LIMIT SESSIONS_PER_USER 2
      CPU_PER_SESSION unlimited
      CPU_PER_CALL 6000
      IDLE_TIME 60
      CONNECT_TIME 120
      LIMIT PASSWORD_REUSE_MAX 10
      PASSWORD_REUSE_TIME 30;
```

In the example above, the profile has the following parameters:

- `LIMIT SESSIONS_PER_USER` – `2`;
- `CPU_PER_SESSION` – `unlimited`;
- `CPU_PER_CALL` – `6000` $1/100^{th}$ seconds (60 seconds);
- `IDLE_TIME` – `60` minutes;
- `CONNECT_TIME` – `120` minutes;
- `LIMIT PASSWORD_REUSE_MAX` – `10`; and
- `PASSWORD_REUSE_TIME` – `30` days.

Theory: Data Administration and Security

Database level security: controlling user access

Access control

Access control in a database is based on granting or revoking privileges to users. There are two general levels of user privileges, system-level and object-level. Oracle also has administration level privileges

System-level privileges

System-level privileges include authorisation to modify any object of a particular type in a database schema. This could include the creation, alteration and deletion of users and any:

- table;
- view;
- procedure;
- trigger;
- index;
- database;
- session etc.

These privileges generally operate on creational operations at the top level.

Only users with admin rights can grant and revoke system level privileges to new users. On DBMS installation, there are normally some superusers. Oracle has the following superusers by default:

- `SYSTEM` – used to perform admin tasks; and
- `SYS` – operates a higher level than `SYSTEM` and controls the automatic maintenance of the database.

The superusers can:

- create more users;
- grant system-level privileges to users; and
- grant object-level privileges to users.

This shows that RDBMSes use discretionary access control. As discussed before on page 130:

- discretionary access control restricts access to objects based on the identity of users and/or groups to which they belong; and
- mandatory access control restricts access to objects based on specifications provided by the system, rather than the users, clearance and classification data are stored in the security labels, which are bound to the users and objects.

Object-level privileges

Object-level privileges include authorisation on specific objects in a database schema. This could include the selection, insertion, updating and deletion of data from specific:

- tables or specific columns of tables; and
- views or specific columns of views.

These privileges generally operate on tables at the bottom level.

When a user creates an object, they become the owner of the object. The owner automatically has all permission on that object. The owner of a table or view can grant and revoke privileges on that object to other users.

Granting privileges

Syntax

Format: Granting privilege(s)

```
GRANT <privilege list> ON <object>
      TO <user list> [WITH GRANT OPTION];
```

When granting privilege(s), the following parameters must be specified:

- `<privilege list>` – a comma-delimited list of the following possible privileges:
  - `select;`
  - `delete;`
  - `insert [(<column list>)]`; and
  - `update [(<column list>)],`

  or `ALL PRIVILEGES` which grants privileges for all of the operations.
- `<object>` – the name of a table or view;
- `<user list>` – a comma-delimited list of user-ids to which the privilege(s) should be granted, or `PUBLIC` which grants the privilege(s) to all valid users.

The `WITH GRANT OPTION` clause means that users identified after the `TO` clause can pass on any of the privileges to other users.

A user who was granted privileges on an object with the "`WITH GRANT OPTION`" may grant and revoke the same privileges on the object to other users. The privileges can propagate to other users without knowledge of the owner. In the same fashion, revoking the privilege will also revoke any privileges granted on the chain of propagation to other users.

Users can be granted privileges on a view without having permission on the underlying tables that construct the view.

Example

`user1` owns two tables, `Branch` and `Loan`. It is then possible for `user1` to grant privileges to other users, as seen below.

Example: Granting privileges

```
GRANT insert, delete ON Loan
      TO user2;

GRANT select ON Branch
      TO user3 WITH GRANT OPTION;
```

In the example above, `user1` has granted:

- `insert` and `delete` privileges on the `Loan` table to `user2`, but cannot grant any privileges on the table; and
- `select` privileges on the `Branch` table to `user3` and allowed `user3` to grant `select` privileges to other users.

Example: Granting privileges through propagation

```
GRANT select ON Branch
      TO user4;
```

In the example above, `user3` has granted `select` privileges on the `Branch` table to `user4`. This demonstrates propagation of privileges between users as the owner of the `Branch` table, `user1`, may not be aware of `user3` granting `select` privileges to `user4`.

Example: Granting privileges for specific columns of a table

```
GRANT update (branchCity, assets) ON Branch
      TO user5
```

In the example above, `user5` has been granted `update` privileges on the `Branch` table but only for the `branchCity` and `assets` columns.

Revoking privileges

Syntax

Format: Granting privilege(s)

```
GRANT <privilege list> ON <object>
      FROM <user list>;
```

When revoking privilege(s), the following parameters must be specified:

- `<privilege list>` – a comma-delimited list of the following possible privileges:
  - `select`;
  - `delete`;
  - `insert [<column list>]`; and
  - `update [<column list>]`,

  or `ALL PRIVILEGES` which revokes privileges for all of the operations.
- `<object>` – the name of a table or view;
- `<user list>` – a comma-delimited list of user-ids to which the privilege(s) should be revoked, or `PUBLIC` which revokes the privilege(s) from all valid users.

If the same privilege was granted twice to the same user by different grantees, the user may retain privilege after the revocation.

All privileges that depend on the privilege being revoked are also revoked. Revocation of a privilege from a user may cause other users also to lose that privilege.

Example

Following on from the example on page 138, if `user1` revokes the select privilege from `user3`, then `user4` will also lose this privilege.

Example: Revoking privileges

```
REVOKE select ON Branch
      FROM user3;
```

Theory: Data Administration and Security

Limiting access with views

Views can be used to limit which columns and/or which rows a user can retrieve or delete.

This may be necessary as rows cannot be specified in the `GRANT select` or `GRANT delete` statements.

Example: Creating a view

Create a view that omits the customer loan details if they belong to another branch.

```
CREATE VIEW CustomerLoanDetails (customerName, loanNumber, amount) AS
      SELECT customerName, loanNumber, amount
      FROM Loan
      WHERE UPPER(branchName) = USER
      UNION
      SELECT customerName, NULL, NULL
      FROM Loan
      WHERE UPPER(branchName) <> USER;  // <> means is not equal to
```

The example above shows the creation of a view that could be used to limit access for specific users to specific rows in a table. Subsequently, privileges could be assigned to a user for a view, rather than the underlying table(s), as the user will only see the data in the view and not all of the data in the underlying table(s).

For example, the following permissions may be granted:

```
GRANT select ON CustomerLoanDetails
TO Midlands WITH GRANT OPTION;
```

The `USER` function returns the currently logged in user; this is achieved by silently performing the SQL statement `SELECT USER FROM DUAL;` which returns the current user from the one-column, one-row table `DUAL` automatically created by Oracle containing the username of the current user. Therefore:

- if `branchName` is equal to `USER` i.e. the currently logged in user has username `Midlands`, then the following data will be returned:

```
customerName  loanNumber  amount
------------  ----------  ------
Smith                 12     300
```

- if `branchName` is not equal to `USER` i.e. the currently logged in user does not have username `Midlands`, then the following data will be returned:

```
customerName  loanNumber  amount
------------  ----------  ------
Smith
```

This view allows the data to be limited to different users such that:

- users can see the `customerName`, `loanNumber` and `amount` for loans from their own branch; and
- users may only see the `customerName` for loans from other branches.

User and role access

A **role** defines a set of privileges that can be granted to multiple users.

Privileges can be granted to roles, as well as users, such that a role contains a set of privileges. A role can then be granted to one or more users. Changes to the role privileges automatically affect the users with that role.

Theory: Data Administration and Security

Format: Creating a role

```
CREATE ROLE <role name>;
```

Example: Creating a role

```
CREATE ROLE demo_small_role;
```

Format: Granting privileges to a role

```
GRANT <privilege list>
      TO <role name>;
```

When granting privilege(s) to a role, the following parameters must be specified:

- `<privilege list>` – a comma-delimited list of the following possible privileges:
    - `select`;
    - `delete`;
    - `insert [(<column list>)]`; and
    - `update [(<column list>)]`,

  or `ALL PRIVILEGES` which grants privileges for all of the operations; and
- `<role name>` – the name of a role.

Example: Granting privileges to a role

```
GRANT create session, create table, create sequence, create procedure, create
operator, create view, create indextype, create trigger
      TO demo_small_role;
```

In the example above, the role `demo_small_role` is granted the following privileges:

- `create session`;
- `create table`;
- `create sequence`;
- `create procedure`;
- `create operator`;
- `create view`;
- `create indextype`; and
- `create trigger`.

Format: Granting a role to a user

```
GRANT ROLE <role name>
      TO <username>;
```

Example: Granting a role to a user

```
GRANT ROLE demo_small_role
      TO jsmith;
```

In the example above, the role `demo_small_role` is granted to the user `jsmith`. This means that the user `jsmith` will inherit all of the privileges in the role `demo_small_role` and will remain subject to changes in privileges in the role as they will inherit changes to this role.

As seen in this example, roles prevent continued maintenance of user's permissions as they can be changed by making changes to roles that are assigned to multiple users.

Practical: Data Administration and Security

Controlling user access with object-level privileges

Example: Granting privileges to a table

From `user1`, grant the `SELECT` permission on the `Deposit` table to `user2`.

From `user2`, test this `GRANT` statement by selecting data from the table `Deposit`.

SQL Statement (user1)	
`GRANT SELECT ON Deposit TO user2;`	
SQL Statement (user2)	**Result**
`SELECT * FROM user1.Deposit;`	Can select from the table successfully.

Example: Revoking privileges to a table

From `user1`, revoke the `SELECT` permission on the `Deposit` table from `user2`.

From `user2`, test this `REVOKE` statement by selecting data from the table `Deposit`.

SQL Statement (user1)	
`REVOKE SELECT ON Deposit FROM user2;`	
SQL Statement (user2)	**Result**
`SELECT * FROM user1.Deposit;`	Can no longer select from the table: `ORA-00942: table or view does not exist`

Practical: Data Administration and Security

Controlling user access with views

Example: Creating a view

From user1, create a view of the `customerName` and `branchName` columns from the table `Deposit` for the customers with balances greater than `400`.

SQL Statement

```
CREATE VIEW DepositRecords AS
      SELECT customerName, branchName
      FROM Deposit
      WHERE balance > 400;
```

Example: Granting privileges to a view

From `user1`, grant the `SELECT` permission on the `DepositRecords` view to `user2`.

From `user2`, test this `GRANT` statement by selecting data from the view `DepositRecords`.

SQL Statement (`user1`)

```
GRANT SELECT ON DepositRecords TO user2;
```

SQL Statement (user2)	Result
`SELECT * FROM user1.DepositRecords;`	Can select from the view successfully.

Example: Revoking privileges to a view

From `user1`, revoke the `SELECT` permission on the `DepositRecords` view from `user2`.

From `user2`, test this `REVOKE` statement by selecting data from the view `DepositRecords`.

SQL Statement (`user1`)

```
REVOKE SELECT ON DepositRecords FROM user2;
```

SQL Statement (user2)	Result
`SELECT * FROM user1.DepositRecords;`	Can no longer select from the view: `ORA-00942: table or view does not exist`

Theory: Transactions and Recovery

Introduction to transactions and recovery

Definitions

A **transaction** is a “logical unit of work” performed within a DBMS against a database. It can be described as the unit of recovery and concurrency.

Recovery techniques are those that are required to ensure that transactions complete successfully despite system failures, such as disk problems or main memory (RAM).

Logs are used to aid recovery, and **checkpoints** are taken to identify what transactions need recovery.

Theory: Transactions and Recovery

Properties of transactions

Transaction manager

Overview

The **transaction manager**, which is a subsystem of the DBMS, ensures that all transactions either:
- `COMMIT` – the transaction completes successfully by completing its read/write operations; or
- `ROLLBACK` – the transaction fails and all updates made so far must be "rolled back" (undone).

All transactions begin with a `BEGIN TRANSACTION` and end with a `COMMIT` or `ROLLBACK` statement.

A `ROLLBACK` statement rolls the database back to the state it was in before the transaction started.

How transaction managers work

	Oracle	MySQL
Autocommit	Defaults to operation with autocommit mode disabled; each SQL statement that modifies the database is only committed when the transaction ends.	Defaults to operation with autocommit mode enabled; each SQL statement that modifies the database is executed immediately.
Transaction Begin	A transaction begins with the first executable SQL statement.	Using the `START TRANSACTION` (or `BEGIN`, `BEGIN WORK`) statement disables autocommit until the transaction is ended.
Transaction End	A transaction ends when it is committed or rolled back either: • explicitly with `COMMIT` (`exit` also performs `COMMIT` on the Oracle client) or `ROLLBACK` statements; or • implicitly with a DDL statement, such as `CREATE TABLE` or `DROP TABLE`.	A transaction that has been started with the `START TRANSACTION` statement can be ended explicitly with `COMMIT` or `ROLLBACK` statements. The autocommit mode then reverts to its previous state.

Note that if you are not using transaction-safe tables, any changes are stored at once, regardless of the status of autocommit mode.

Oracle autocommit

It is possible to manually enable autocommit in SQLPlus.

Format: Enabling autocommit

```
Set autocommit on
```

Running the above command in SQLPlus will enable autocommit. Autocommit will remain enabled until manually disabled.

Format: Disabling autocommit

```
Set autocommit off
```

Running the above command in SQLPlus will disable autocommit.

Theory: Transactions and Recovery

It is also possible to enable autocommit for a specified number of statements.

Format: Enabling autocommit for a number of statements

```
Set autocommit <number of statements>
```

Running the command above in SQLPlus will enable autocommit for the for the number of statements specified.

Example: Enabling autocommit for a number of statements

```
Set autocommit 10
```

In the example above, autocommit will be enabled until ten statements have been executed.

The state of autocommit can be determined by performing `Show autocommit` in SQLPlus which will return whether autocommit is enabled or disabled.

Oracle savepoints

Savepoints enable transactions to be "partitioned". It is possible to create savepoints to which the current transaction can be rolled back. If this is omitted, the `ROLLBACK` statement will undo the entire transaction.

Format: Creating a savepoint

```
SAVEPOINT <savepoint name>;
```

Format: Rolling back to a savepoint

```
ROLLBACK TO SAVEPOINT <savepoint name>;
```

Example: Using savepoints

```
COMMIT;

UPDATE Loan
SET amount = 4000
WHERE loanNumber = 11;

SAVEPOINT update11;

UPDATE Loan
SET amount = 350
WHERE loanNumber = 12;

SAVEPOINT update12;

ROLLBACK TO SAVEPOINT update11;

UPDATE Loan
SET amount = 3500
WHERE loanNumber = 12;
```

In the example above two savepoints are created, `update11` and `update12`. A statement is then issued to rollback to the savepoint `update11`, and therefore the effects of the `UPDATE` statement issued between the two savepoints is undone.

Theory: Transactions and Recovery

ACID

ACID is a standard set of properties which guarantee that transactions are processed reliably.

Atomicity	This requires that a transaction is “all or nothing”; if a transaction fails, then the whole transaction must fail.
Consistency	A transaction must take the database from one consistent valid state to another consistent valid state according to the rules of the database, such as primary key and foreign key constraints.
Isolation	A transaction should be isolated, such that they are not seen by other transactions until the initial transaction is committed. This ensures that a user of process cannot access data that is being transacted until the new consistent state has been committed to the database.
Durability	Once a transaction commits, its updates must be durable. This ensures that changes made to the database cannot be lost afterwards as a result of any system failure problems; the transaction should be written to secondary memory as quickly as possible.

Theory: Transactions and Recovery

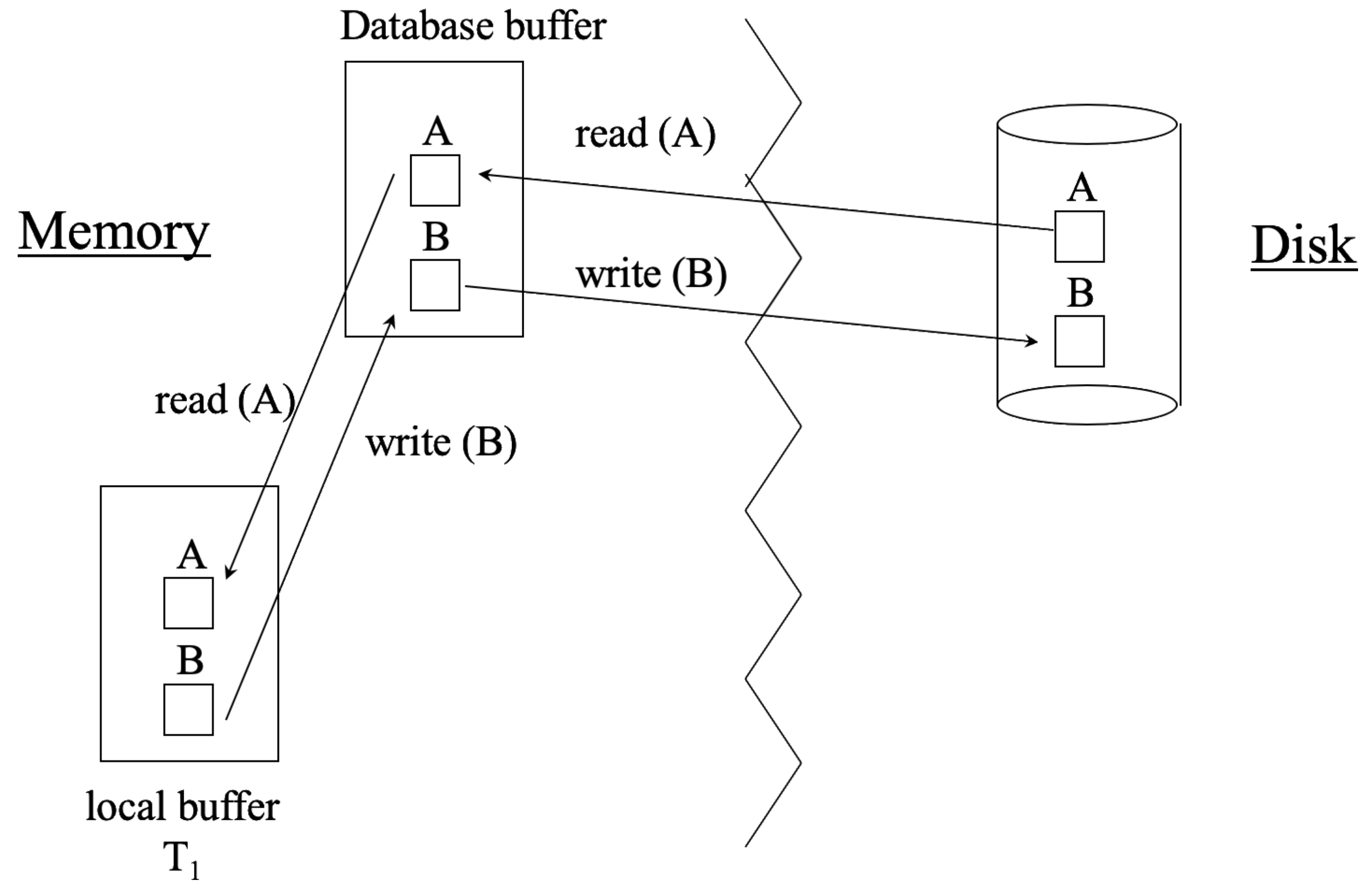
Operation of the database and recovery

System model

The **system model** of a database when considering the buffer between main memory and secondary memory.

A database must be optimised for speed and these optimisations may mean that the system is more vulnerable to crashes.

Diagram: System model



The table below shows the characteristics of the components of the system model.

Component	Disk	Database Buffer	Local Buffer
Location	Disk	Memory	Memory
Usage	Secondary memory for long-term storage.	Used for transferring data between disk and memory.	Used to store data that is to be processed.
Characteristics	• Large capacity. • Slow. • Persistent, used for long-term storage. • Non-volatile, its contents are preserved between power cycles.	• Small capacity. • Very fast. • Transient, used for short-term storage. • Volatile, its contents are lost between power cycles.	• Small capacity. • Fast. • Transient, used for short-term storage. Volatile, its contents are lost between power cycles.

Theory: Transactions and Recovery

Operation	Reading from the disk	Writing to the disk
Stage 1	The page containing the record is found and retrieved from the disk, a relatively slow storage medium.	A page is not instantly transferred from the database buffer to disk because write operations to the disk are relatively slow. Leaving the page in the database buffer allows that page to be accessed and used again quickly in subsequent operations.
Stage 2	**Disk -> Database Buffer** The page is transferred to the database buffer.	A "buffer flush" will occur when conditions are met by a policy for emptying the database buffer.
Stage 3	Searching for the particular record in the page occurs on the database buffer as this is a faster medium than the disk, therefore saving time.	If a "buffer flush" is to be performed, the contents of the database buffer will then be written to the disk according to a set policy, such as: • a FIFO policy in which the first page read from the disk and put in to the database buffer will be the first page to be written back to the disk; or • a policy in which the least recently used pages are written back to the disk first.
Stage 4	**Database Buffer -> Local Buffer** Individual records may then be transferred to the local buffer.	
Stage 5	A single column or row may then be processed by the CPU.	

Policies for performing a "buffer flush" are described in the table below.

Policy	Steal	No-steal	Force	No-force
Description	Allows the buffer manager to write a buffer to disk before a transaction commits, as such the buffer manager "steals" a page from the transaction.	Does not allow the buffer manager to write a buffer to disk before a transaction commits.	Ensures that all pages updated by a transaction are immediately written to disk when the transaction commits.	Pages may remain in the buffer until the buffer becomes full and only then will they be written to disk.
Advantage	**Avoids the needs for a very large buffer space** to store all updated pages by a set of concurrent transactions.	**The changes of an aborted transaction do not have to be undone** as they have not been written to disk.	**The changes of committed transaction do not have to be redone** if there is a subsequent crash as they have already been written to disk at commit.	**Pages do not have to be re-written to disk** for a later transaction that has been updated by an earlier committed transaction as they may still be in the database buffer.

It is possible to combine policies to create the following standard policies:

- steal, force;
- steal, no-force;
- no-steal, force; and
- no-steal, no-force.

However, when inspecting the advantages of the policies above, most DBMSs employ a no-steal, no-force policy.

Theory: Transactions and Recovery

Transactions may be interleaved as while input/output (I/O) is in process for one transaction, such as relatively slow reads/writes to the disk, the CPU can be used to perform another transaction in the queue using data in the local buffer.

Whilst this model aids to optimise speed, it poses an issue as pages left in the database buffer waiting to be written back to the disk may be lost in the event of a system crash. This issue is explored in the example below.

Example of a transaction

A transaction is to transfer `£50` from `accountA` to `accountB`.

The stages in this transaction will be as follows:

1) `read(accountA)` from the database buffer or disk;
2) subtract `50` from `accountA`;
3) `write(accountA)` to the database buffer or disk;
4) `read(accountB)` from the database buffer or disk;
5) add `50` to `accountB`; and
6) `write(accountB)` to the database buffer or disk.

If the system crashes between stages 3 and 6 inclusive, then the database would be left in an inconsistent state. Subsequently, the transaction must be undone or rolled back – this can be achieved by using log files.

Theory: Transactions and Recovery

Log files

Definition

A **log file** (or **audit trail**) is a record of the history of actions executed by a DBMS showing:

- which user has accessed the database;
- what operations the user has performed;
- the period of time the operations were performed; and
- where (terminal) the operations were performed.

A log file may be used to guarantee ACID properties over crashes or hardware failures.

Entries to a log file

A record containing the old and new values of the changed object, such as the contents of a cell in a table, are written to the log:

- every time a change is made to the database; and
- when transactions `BEGIN`, `COMMIT` or `ROLLBACK`.

The log is normally stored on disc.

Write-ahead log

Definition

A **write-ahead log** is one where a record is added to the log file before the respective operation is executed.

How is data written?

A transaction in a log is represented as T_i, where i is an integer index value.

The table below shows the records that may be written to the log file when a transaction T_i performs operations.

When transaction T_i...	Record written to log	Purpose
...starts	BEGIN T_i	The transaction is registering itself on the log.
...executes `write(X)`, where `X` is an object	$< T_i, X, \text{old value}, \text{new value} >$	The old and new value are written to the log before the operation `write(X)` is executed to enable restore in the future.
...reaches its last statement	$< T_i\ \text{COMMITS} >$	The transaction is signaling the end of the transaction on the log.

If `X` is modified with the operation `write(X)`, then its corresponding log record is always written on the log before the changes are committed to the disk.

Before transaction T_i is commited, all of its corresponding log records must be in “stable storage”, as such they must be written to the disk. The system can use the corresponding log entry to restore the object to its original value.

Theory: Transactions and Recovery

Example transactions and log records

The table below contains some example transactions and their respective operations.

Transaction T_1	Transaction T_2
read(A)	read(A)
Add 50 to A	Add 10 to A
read(B)	write(A)
Add 100 to B	read(D)
write(B)	Subtract 10 from D
read(C)	read(E)
Multiply C by 2	read(B)
write(C)	Add B to E
Add B and C to A	write(E)
write(A)	Add E to D
	write(D)

This example assumes the following initial values:

A = 100
B = 300
C = 5
D = 60
E = 80

As a result, the table below shows the log records written by transaction T_1 and transaction T_2 can be seen in the table below complemented by the operations in the transaction that triggered the record to be written.

Transaction T_1		Transaction T_2	
Records written to log	**Operations**	**Records written to log**	**Operations**
$< \text{BEGIN } T_1 >$	read(A)	$< \text{BEGIN } T_2 >$	read(A)
	Add 50 to A		Add 10 to A
	read(B)	$< T_2, A, 510, 520 >$	write(A)
	Add 100 to B		read(D)
$< T_1, B, 300, 400 >$	write(B)		Subtract 10 from D
	read(C)		read(E)
	Multiply C by 2		read(B)
			Add B to E
$< T_1, C, 5, 10 >$	write(C)	$< T_2, E, 80, 480 >$	write(E)
	Add B and C to A		Add E to D
$< T_1, A, 100, 510 >$	write(A)	$< T_2, D, 60, 540 >$	write(D)
$< T_1, \text{COMMITS} >$		$< T_2, \text{COMMITS} >$	

This example demonstrates how a log file could be used to restore objects to their initial values in the event of a system crash during transaction T_1 or transaction T_2.

The records in the log file can be reversed and followed through to obtain the initial value for each object, given the object's current value. This is useful because it does not require knowledge of:

- precise time of the crash, rather it is known that the crash occurred after BEGIN and before COMMITS; or
- the specific operations performed on the objects.

Theory: Transactions and Recovery

Failure of transactions

Types of failure

Failure Type	Impact	Possible Cause(s)	Resolution
Local Failure	Describes an error within a single transaction.	Deadlock.	This is dealt with by `ROLLBACK`.
System Failure	Affects all transactions in progress.	Power failure affecting the local memory (RAM) and therefore affecting the database buffers.	Use logs to restore the database to a valid and consistent state.
Media Failure	Disastrous failure.	Head crash on disk.	It is necessary to reload the database from the most recent backup copy and then use the log file to redo transactions that committed since the copy was taken.

The resolution to these failures assumes that:

- log files are backed-up on a different disc(s);
- the most recent backup is the one to be restored; and
- all transactions that were committed are to be reapplied using redo logs.

System failures

Resolution

In system failures, the contents of main memory, including database buffers, are lost as this is a volatile storage medium.

The table below shows the actions that must be taken in order for the database to be restored to a valid and consistent state that adheres to the ACID properties.

The transaction was...	Consequence	Action
...in progress as no COMMIT was issued.	The operations in the transaction are not complete.	These transactions must be undone.
...completed as a COMMIT was issued.	The buffers may have not been written to the database.	These transactions must be redone.

Checkpoint records are used to enable the system to determine which transactions are to be undone and which transactions are to be redone.

Checkpoint records and restart procedures

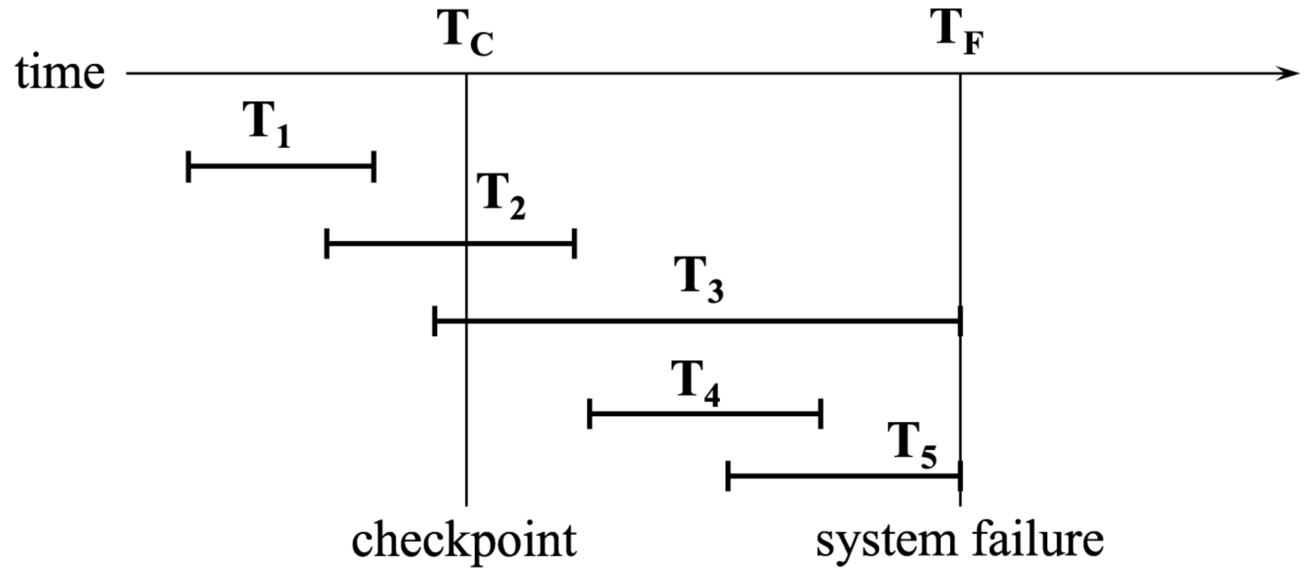
At certain intervals, a checkpoint record is taken. This involves:

- forcing log buffers on to the physical log, such that logs in the buffer are written to the disk;
- forcing database buffers on to the physical database, such that the pages in the buffer are written to the disk; and
- writing a checkpoint record to the log which lists all transactions in progress.

It is important to note that the log buffers are forced on to the physical log before the database buffers are forced on to the physical database because, if there is a problem with the data, the logs can be used to amend the data by undoing or redoing transactions.

Theory: Transactions and Recovery

Diagram: Example of checkpoints



In the diagram above:

- $\mathbf{T_C}$ represents the point in time when a checkpoint was made; and
- $\mathbf{T_F}$ represents the point in time when a system failure occurred.

The **restart procedure** contains the actions that need to be taken on each transaction, on the event of system failure at $\mathbf{T_F}$, in order to restore the database to a valid and consistent state.

The table below shows the restart procedure for the database with transactions shown in the diagram above.

Transaction	Action	Explanation
$\mathbf{T_1}$	No action	The transaction was completed and committed before the checkpoint record was made at $\mathbf{T_C}$. Therefore, the updates were forced to the database and written to the disc at $\mathbf{T_C}$.
$\mathbf{T_2}$	Redone	The transaction commits after the checkpoint is made at $\mathbf{T_C}$ The transaction commits before the system failure occurred at $\mathbf{T_F}$. Therefore, the buffers may not have been written to the database.
$\mathbf{T_3}$	Undone	The transaction did not commit before the system failure occurred at $\mathbf{T_F}$. Therefore, the operations in the transaction were not complete.
$\mathbf{T_4}$	Redone	The transaction commits after the checkpoint is made at $\mathbf{T_C}$ The transaction commits before the system failure occurred at $\mathbf{T_F}$. Therefore, the buffers may not have been written to the database.
$\mathbf{T_5}$	Undone	The transaction did not commit before the system failure occurred at $\mathbf{T_F}$. Therefore, the operations in the transaction were not complete.

The checkpoint record will have recorded that there were two transactions in progress, transaction $\mathbf{T_2}$ and transaction $\mathbf{T_3}$.

Any transactions which were rolled back before the system failure occurred at $\mathbf{T_F}$ do not enter the restart procedure because the purpose of a `ROLLBACK` is to undo the actions of a set of operations such that it appears that they were never executed in the first place. As a result, there is no need for these transactions to enter the restart procedure as they are not intended to be committed to the database.

Theory: Transactions and Recovery

The restart procedure uses two distinct lists, `UNDO` and `REDO`. Below are the steps taken in the restart procedure.

1) The `UNDO` list is set equal to the list of transactions in progress in the last checkpoint record.
2) The `REDO` list is set to empty.
3) The system searches forward through the log from the checkpoint record:
    - for each `BEGIN` record, add the corresponding transaction to the `UNDO` list; and
    - for each `COMMIT` entry, move the transaction from the `UNDO` list to the `REDO` list.
4) The system works backwards through the log, undoing transactions on the `UNDO` list.
5) The system works forwards through the log, redoing transactions on the `REDO` list.

Using the diagram on page 154, the `UNDO` and `REDO` lists will be as shown below.

Time	`UNDO` List	`REDO` list	Explanation
Checkpoint (T_C)	T_2, T_3	empty	T_2 and T_3 are in progress, but not committed and are added to the `UNDO` list.
T_2 COMMITs	T_3	T_2	T_2 is completed and is moved from the `UNDO` list to the `REDO` list.
T_4 BEGINs	T_3, T_4	T_2	T_4 is now in progress, but not committed and is added to the `UNDO` list.
T_5 BEGINs	T_3, T_4, T_5	T_2	T_4 is now in progress, but not committed and is added to the `UNDO` list.
T_4 COMMITs	T_3, T_5	T_2, T_4	T_4 is completed and is moved from the `UNDO` list to the `REDO` list.

When the system uses these logs to restore transactions, the order of operations will be:
- T_5 undone;
- T_3 undone;
- T_2 redone; and then
- T_4 redone.

Practical: Transactions and Recovery

Updating data

Example: Updating by multipliers

Add 10% interest to each customer balance in the deposit table for customers with branch name `'RoyalBank'`.

SQL Statement

```
UPDATE Deposit
SET balance = balance * 1.1
WHERE branchName = 'RoyalBank';
```

Example: Updating with `WHERE EXISTS` clause

Interest charges are going to be made on the amounts of some customer loans.

If a customer has one or more deposits with the branch it has a loan with, that branch wants to add interest at 2% of the loan amount. Update the loan amounts using one SQL command.

SQL Statement

```
UPDATE Loan
SET amount = amount * 1.02
WHERE EXISTS
      ( SELECT * FROM Deposit
        WHERE Deposit.customerName = Loan.customerName
        AND Deposit.branchName = Loan.branchName );
```

OR

```
UPDATE Loan
SET amount = amount * 1.02
WHERE (customerName, branchName) IN
      ( SELECT Loan.customerName, Loan.branchName
        FROM Deposit, Loan
        WHERE Deposit.customerName = Loan.customerName
        AND Deposit.branchName = Loan.branchName );
```

In the example above:

- the first solution uses the `Loan` table from the `UPDATE` clause in the subquery as a second `Loan` table is not defined in the subquery; and
- the second solution defined a second `Loan` table in the subquery.

Theory: Concurrency Control

Introduction to concurrency control

Definitions

Concurrency control is needed to prevent concurrent transactions from interfering with each other. It defines strategies that allow multiple transactions (by multiple users) to run simultaneously and not interfere with each other's progress. This is important for transactions that access the same data, rather than independent transactions.

Locking and **timestamping** are the major techniques for implementing concurrency control.

Throughput is the amount of transactions in a given time interval.

Problems with concurrent transactions

Collisions can occur between concurrent tractions that want to access the same data by performing read or write operations on the same data.

Problem	Definition	Explanation
Lost Update	An update to an object by some transaction is overwritten by another interleaved transaction without knowledge of the initial update	The read operation by the second transaction occurs before the write operation of the first transaction.
Uncommitted Dependency	A transaction reads an object updated by another transaction that later fails.	The second transaction reads the values written by the first transaction, subsequently the first transaction is later rolled back or fails and therefore the second transaction commits values based on values that should have been rolled back due to user intervention or a deadlock.
Inconsistent Analysis	A transaction calculating an aggregate function uses some but not all updated objects or another transaction.	The write operations from another transaction are modifying the values used by the aggregate function and therefore the result will be composed of values from two different points in time.

The problems in the table above can be resolved using concurrency control.

Why is concurrency control required?

Concurrency control is required to handle problems that can occur when transactions execute concurrently (at the same time).

In order to execute transactions in an interleaved manner, it is necessary to have some form of concurrency control. This enables a more efficient use of computer resources.

It is necessary to isolate the transactions to avoid the problems while maintaining concurrency. It is important to maintain concurrency as to improve throughput by allowing the CPU to execute multiple transactions concurrently (at the same time) instead of waiting for input/output (I/O) operations, i.e. read/write operations, to complete.

Theory: Concurrency Control

Locks

How do locks work?

When a transaction requires a database object, such as a row or set of rows, it must obtain a **lock**.

A lock is obtained from a system component called the **lock manager**.

The table below shows the two typical types of locks.

Lock Type	Description	When is this lock obtained?
Exclusive Lock (X) ***(or Write Lock)***	Only one transaction may have this lock at any given time.	An exclusive lock (X) is obtained when a transaction is to perform a write operation. This is because only one transaction should update an object at any given time. As a result, when a transaction intends to update an object it must obtain an exclusive lock (X) on the object. If one is not available, the transaction must wait.
Shared Lock (S) ***(or Read Lock)***	One or more transactions may have this lock at any given time.	A shared lock (S) is obtained when a transaction is to perform a read operation. This is because it should be possible for multiple transactions to read an object at the same time, as long as the transactions are not to update the object.

Using locks in the lost update problem

Problem example

The occurrence of the lost update problem can be seen by considering the two transactions, $\boldsymbol{T_A}$ and $\boldsymbol{T_B}$, shown below. This example assumes that starting value of `R` is `12`.

Transaction $\boldsymbol{T_A}$	Transaction $\boldsymbol{T_B}$	Value of R
`read(R)`		12
Add `50` to `R`		12
	`read(R)`	12
	Add `100` to `R`	12
`write(R)`		62
	`write(R)`	112

Transaction $\boldsymbol{T_A}$ adds `50` to the value `R` but does not write the updated value back to the database until after transaction $\boldsymbol{T_B}$ has read the value `R`.

This means that transaction $\boldsymbol{T_B}$ will add `100` to an outdated value of `R`. As a result, the operations taken place in Transaction $\boldsymbol{T_A}$ are lost.

The desired effect of the two transactions is `R+50+100` or `R+100+50`, however the actual operation that takes place is `R+100`.

This problem occurs because the operation `read(R)` in transaction $\boldsymbol{T_B}$ occurs before the operation `write(R)` in transaction $\boldsymbol{T_A}$.

Solution

The lost update problem can be handled by using exclusive locks (X).

Using the example above, transaction $\boldsymbol{T_A}$ can obtain an exclusive lock (X) on value `R` such that transaction $\boldsymbol{T_B}$ will have to wait until transaction $\boldsymbol{T_A}$ releases the lock so that it can have an exclusive lock (X) on value `R`.

Theory: Concurrency Control

Transaction T_A	Transaction T_B	Value of R
Begin		12
`x_lock(R)`		12
`read(R)`		12
Add 50 to R	Begin	12
	`x_lock(R)`	12
	wait	12
`write(R)`	*wait*	62
`COMMIT/unx_lock(R)`	*wait*	62
	`read(R)`	62
	Add 100 to R	62
	`write(R)`	62
	`COMMIT/unx_lock(R)`	112

At the beginning of transaction T_A, it gains an exclusive lock (X) on the value R and is able to perform its read operation and subsequent operations.

When transaction T_B begins and attempts to obtain gain an exclusive lock (X) on the value R, it must wait until the exclusive lock is released by transaction T_A before performing its read operation and subsequent operations.

As shown above, when a `commit` operation is performed, the exclusive locks (X) are released.

The use of the exclusive lock (X) prevents the operation `read(R)` in transaction T_B occurring before the operation `write(R)` in transaction T_A, and therefore the desired result of R+50+100 or R+100+50 is achieved.

Using locks in the uncommitted dependency problem

Problem example

The occurrence of the uncommitted dependency problem can be seen by considering the two transactions, T_A and T_B, shown below. This example assumes that starting value of R is 100.

Transaction T_A	Transaction T_B	Value of R
`read(R)`		100
Subtract 50 from R		100
`write(R)`		50
	`read(R)`	50
	Add 75 to R	125
`ROLLBACK`		
	`write(R)`	125
	`COMMIT`	125

Transaction T_A subtracts 50 from the value R but does not commit the write operation back to the database until after transaction T_B has read the value R, instead transaction T_A is rolled back.

This means that transaction T_B will add 75 to an uncommitted value of R. As a result, the value R will be incorrect. The impact of transaction T_A should not be present in the database as it has been rolled back, however its changes have been written and committed to the database by transaction T_B.

The desired effect of the two transactions is R+75, however the actual operation that takes place is R−50+75.

This problem occurs because the operation `read(R)` in transaction T_B occurs before the operation `ROLLBACK` in transaction T_A.

Solution

The uncommitted dependency problem can be solved by an extension to the locking protocol. The order of read and write operations in the example above are in the correct order, however because transaction T_B was rolled back, there is still an issue.

It is necessary to ensure that exclusive locks (X) are retained until the end of a transaction in case the transaction is rolled back.

Using the example above, transaction T_B would not be allowed to obtain an exclusive lock (X) on value R until the transaction T_A has completed, either by a COMMIT or ROLLBACK.

Transaction T_A	Transaction T_B	Value of R
Begin		100
x_lock(R)		100
read(R)		100
Subtract 50 from R	Begin	100
	x_lock(R)	100
	wait	100
write(R)	*wait*	50
ROLLBACK/unx_lock(R)	*wait*	50
	read(R)	50
	Add 75 to R	50
	write(R)	125
	COMMIT/unx_lock(R)	125

At the beginning of transaction T_A, it gains an exclusive lock (X) on the value R and is able to perform its read operation and subsequent operations.

When transaction T_B begins and attempts to obtain gain an exclusive lock (X) on the value R, it must wait until the exclusive lock (X) is released by transaction T_A before performing its read operation and subsequent write operations.

As shown above, when a COMMIT or ROLLBACK operation is performed, the exclusive locks (X) are released.

The use of the exclusive lock (X) prevents the operation read(R) in transaction T_B occurring before the operation ROLLBACK in transaction T_A, and therefore the desired result of R+75 is achieved.

Using locks in the inconsistent analysis problem

Problem example

The occurrence of the inconsistent analysis problem can be seen by considering the two transactions, T_A and T_B, shown below. This example assumes that the values for ACC1, ACC2 and ACC3 are 30, 40 and 50 respectively.

Transaction T_A	Transaction T_B	Values			
		SUM	ACC1	ACC2	ACC3
read(ACC1)			30	40	50
SUM = ACC1		30	30	40	50
read(ACC2)		30	30	40	50
SUM = SUM + ACC2		70	30	40	50
	read(ACC1)	70	30	40	50
	Add 10 to ACC1	70	30	40	50
	read(ACC3)	70	30	40	50
	Subtract 10 from ACC3	70	30	40	50
	write(ACC3,ACC1)	70	40	40	40
	COMMIT	70	40	40	40
read(ACC3)		70	40	40	40
SUM = SUM + ACC3		110	40	40	40

Transaction T_A attempts to calculate the sum of the three values ACC1, ACC2 and ACC3. Meanwhile, transaction T_B updates the values for ACC1 and ACC3 and commits the changes.

This means that transaction T_A will read the updated value of ACC3 and therefore the value in SUM will be inconsistent.

The desired effect of the two transactions is 30+40+50, however the actual operation that takes place is 30+40+40.

This problem occurs because the operation read(ACC3) in transaction T_A occurs after the operation COMMIT in transaction T_B.

Solution

The inconsistent analysis problem can be solved by a further extension to the locking protocol. Transaction T_A in the example above only performs read operations, however because transaction T_B updates values that transaction T_A is reading, there is still an issue.

A transaction may need to keep an object locked even if it is not updating the object.

Using the example above, it is necessary for transaction T_A to obtain a shared lock (S) on the values `ACC1`, `ACC2` and `ACC3` until the transaction T_A has completed, either by a `COMMIT` or `ROLLBACK`.

Transaction T_A	Transaction T_B	Values			
		SUM	**ACC1**	**ACC2**	**ACC3**
Begin			30	40	50

`s_lock(ACC1),`
`s_lock(ACC2),`
`s_lock(ACC3)`
		30	30	40	50

`read(ACC1)`		30	30	40	50
`SUM = ACC1`		70	30	40	50
`read(ACC2)`		70	30	40	50
`SUM = SUM + ACC2`	Begin	70	30	40	50
	`x_lock(ACC1), x_lock(ACC3)`	70	30	40	50
	wait	70	30	40	50
`read(ACC3)`	*wait*	70	30	40	50
`SUM = SUM + ACC3`	*wait*	120	30	40	50

`uns_lock(ACC1),`
`uns_lock(ACC2),`
`uns_lock(ACC3)`
	wait	120	30	40	50

	`read(ACC1)`	120	30	40	50
	Add `10` to `ACC1`	120	30	40	50
	`read(ACC3)`	120	30	40	50
	Subtract `10` from `ACC3`	120	30	40	50
	`write(ACC3,ACC1)`	120	40	40	40
	`COMMIT/unx_lock(ACC1),` `unx_lock(ACC3)`	120	40	40	40

At the beginning of transaction T_A, it obtains a shared lock (S) on the values `ACC1`, `ACC2` and `ACC3` and is able to perform its read operation and addition operations.

When transaction T_B begins and attempts to obtain an exclusive lock (X) on the values `ACC1` and `ACC3`, it must wait until the shared lock (S) is released by transaction T_A before performing its read operation and subsequent write operations.

As shown above, when a `COMMIT` operation is performed, the exclusive locks (X) are released by transaction T_B. However, the shared locks (S) are released by transaction T_A once the read operations and arithmetic are complete.

The use of the shared lock (S) prevents the operation `write(ACC3,ACC1)` in transaction T_B occurring before the operation `SUM = SUM + ACC3` in transaction T_A, and therefore the desired result of `30+40+50` is achieved.

Theory: Concurrency Control

Lock compatibility matrix

The **lock compatibility matrix** enforces that a transaction may only set a lock on an object if the lock is compatible with the locks already held on the item by other transactions.

Locks can only be obtained from the lock manager according to the compatibility matrix shown below.

Diagram: Lock compatibility matrix

		Transaction T_A has...	
		A Shared Lock (S)	An Exclusive Lock (X)
Transaction T_B requests...	A Shared Lock (S)	✓	X
	An Exclusive Lock (X)	X	X

The lock compatibility matrix shows that if transaction T_A has a shared lock (S) on an object then transaction T_B:

- can request a shared lock (S) on the same object; and
- cannot request an exclusive lock (X) on the same object.

The lock compatibility matrix also shows that if transaction T_A has an exclusive lock (X) on an object then transaction T_B:

- cannot request a shared lock (S) on the same object; and
- cannot request an exclusive lock (X) on the same object.

In short, it is only possible for a transaction to request a lock on an object:

- a transaction may only request a shared lock (S) if no other transaction has an exclusive lock (X) on the same object; and
- a transaction may never request an exclusive lock (X) on an object if any other transaction has any lock on the same object.

In addition, if the lock compatibility matrix allows, it is possible for a transaction to “upgrade” its lock from a shared lock (S) to an exclusive lock (X).

Locking rules

- Any number of transactions can hold shared locks (S) on an object.
- If any transaction holds an exclusive lock (X) on an object, no other transaction may hold any lock on the object.
- A transaction holding an exclusive lock (X) may issue a write or read request on the object.
- A transaction holding a shared lock (S) may only issue a read request on the object.

Theory: Concurrency Control

Serialisability and two-phase locking (2PL)

Serialisability

What is serialisability?

A given interleaved execution of a set of transactions (concurrent transactions) is considered correct if it is **serialisable**.

A given interleaved execution of a set of transactions (concurrent transactions) is said to be **serialisable** if and only if it produces the same results as some serial execution of the same transactions.

Justification of serialisability

If it is assumed that individual transactions are correct, it can be deduced that:

- running the transactions one at a time in any serial order is correct; and
- an interleaved execution is also correct if it serialisable, i.e. it is equivalent to some serial execution.

Example

In the examples shown in the previous section, the original interleaved executions were not equivalent to running:

- transaction $\boldsymbol{T_A}$ then transaction $\boldsymbol{T_B}$; or
- transaction $\boldsymbol{T_B}$ then transaction $\boldsymbol{T_A}$.

These problems were solved using locks. The effect of locking was to force serialisability.

Concurrency control problems occur when transactions are not serialisable. In order to guarantee serialisability, two-phase locking (2PL) must be used.

Two-phase locking (2PL)

Definition

Two-phase locking (2PL) is a protocol which guarantees serialisability of transactions.

How it works

As with normal locks, before operating on an object, through read or write operations, a transaction must obtain a lock on that object.

However, two-phase locking (2PL) enforces that, after a transaction releases a lock, the transaction must never obtain any new locks.

A transaction obeying this protocol will have two phases:

- a growing phase – during which locks are obtained; and
- a shrinking phase – during which locks are released.

The shrinking phase is often compressed into the single operation `COMMIT` or `ROLLBACK`.

Using two-phase locking (2PL)

Problem example

A problem that must be solved using two-phase locking (2PL) can be seen by considering the two transactions, T_A and T_B, shown below. This example assumes that the values for A and B are both 20.

Transaction T_A	Transaction T_B
read(A)	read(A)
Add 10 to A	A = A * 1.2
write(A)	write(A)
read(B)	read(B)
Subtract 10 from B	B = B * 1.2
write(B)	write(B)

If transaction T_A was to be executed first, followed by transaction T_B:

- transaction T_A – A = 20 + 10 = 30
  B = 20 - 10 = 10
- transaction T_B – A = 30 * 1.2 = 36
  B = 10 * 1.2 = 12

If transaction T_B was to be executed first, followed by transaction T_A:

- transaction T_B – A = 20 * 1.2 = 24
  B = 20 * 1.2 = 24
- transaction T_A – A = 24 + 10 = 34
  B = 24 - 10 = 14

This shows that the order in which transaction T_A and transaction T_B are executed has an impact on the resulting values of A and B.

Possible solution

Transaction T_A	Transaction T_B	Values A	Values B
x_lock(A)		20	20
read(A)		20	20
Add 10 to A		20	20
write(A)		30	20
unx_lock(A)		30	20
	x_lock(A)	30	20
	read(A)	30	20
	A = A * 1.2	30	20
	write(A)	36	20
	x_lock(B)	36	20
	read(B)	36	20
	B = B * 1.2	36	20
	write(B)	36	24
	unx_lock(A), unx_lock(B)	36	24
x_lock(B)		36	24
read(B)		36	24
Subtract 10 from B		36	14
write(B)		36	14
unx_lock(B)		36	14

At the beginning of transaction T_A, it obtains an exclusive lock (X) on the value A and is able to perform its read operation and addition operations before releasing the lock.

Subsequently, transaction T_B obtains an exclusive lock (X) on the values A and B and is able to perform its read operation and multiplication operations before releasing the locks.

Lastly, transaction T_A obtains an exclusive lock (X) on the value A and is able to perform its read operation and subtraction operation before releasing the lock.

In this solution, the locks are working however there is still an issue. This is because transaction T_B performs its operations during the lifetime of transaction T_A.

This is a problem because the two serial orderings of transaction T_A and transaction T_B as shown in the problem example do not result in the same resulting values of A and B as this possible solution.

Theory: Concurrency Control

Example using two-phase locking (2PL)

Transaction T_A		Transaction T_B	
Growing Phase	`s_lock(Y)`		
	`read(Y)`		
	`x_lock(Y)`		
Shrinking Phase	`unlock(Y)`		
		`x_lock(Y)`	***Growing Phase***
		`read(Y)`	
		`write(Y)`	
	`read(X)`		
	`write(X)`		
	`unlock(X)`		
		`s_lock(X)`	
		`read(X)`	
		`unlock(X)`	***Shrinking Phase***
		`unlock(Y)`	

Both transaction T_A and transaction T_B have a growing phase, where they obtain locks, and a shrinking phase, where they release locks.

Using two-phase locking (2PL) removes the issues present in the possible solution on the previous page as serialisability of the concurrent transactions, T_A and T_B, is guaranteed.

During the growing phase, a transaction can:

- obtain an exclusive lock (X) on an object;
- obtain a shared lock (S) on an object; and
- convert a shared lock (S) to an exclusive lock (X).

During the shrinking phase, a transaction can:

- release an exclusive lock (X);
- release a shared lock (S); and
- convert an exclusive lock (X) to a shared lock (S).

This protocol may be adapted to optimise timings and improve performance. For example, Oracle implements table-level locks.

Deadlock

Definition

Deadlock occurs when two transactions are each waiting for locks held be the other to be released.

Example

Transaction $\mathbf{T_A}$	Transaction $\mathbf{T_B}$
`x_lock(R)`	
`read(R)`	
	`x_lock(S)`
	`read(S)`
`request x_lock(S)`	
wait	`request x_lock(R)`
wait	*wait*
wait	*wait*
…	…

Transaction $\mathbf{T_A}$ obtains an exclusive lock (X) on the value `R` and transaction $\mathbf{T_B}$ obtains a shared lock (S) on the value `S`.

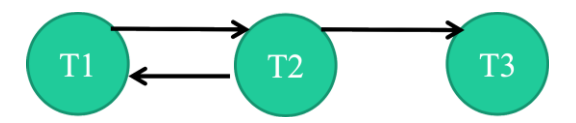
Subsequently:
- transaction $\mathbf{T_A}$ requests an exclusive lock on the value `S` but has to wait as transaction $\mathbf{T_B}$ has a shared lock on the value `S`; and
- transaction $\mathbf{T_B}$ requests an exclusive lock on the value `R` but has to wait as transaction $\mathbf{T_A}$ has an exclusive lock on the value `R`.

In this example, both transaction $\mathbf{T_A}$ and transaction $\mathbf{T_B}$ are waiting for a lock to be released by one another, and therefore deadlock has occurred.

Prevention

If a deadlock occurs, it must be detected by the system. The **wait-for graph** shows which transactions are waiting and the transactions for which they are waiting.

Diagram: Wait-for graph



The diagram shows an example of a wait-for graph in which:
- transaction T1 is waiting for transaction T2 to release a lock;
- transaction T2 is waiting for transaction T3 to release a lock; and
- transaction T2 is waiting for transaction T1 to release a lock.

The system can detect a deadlock by detecting a cycle in the wait-for graph. In this example, there is a cycle in the wait-for graph as transaction T1 and transaction T2 are both waiting for each other to release a lock.

Breaking the deadlock then involves choosing one of the transactions in the deadlock and rolling it back. This releases the transaction's locks and allows other transactions to proceed. Subsequently, the transaction that has been rolled back can later be restarted.

The system may choose which transaction to be rolled back by using a timeout mechanism. This assumes deadlock if a transaction has performed no operations for a prescribed period of time. Once this time is up, the transaction will be rolled back. However, in order for the rollback to have the least impact on the database in terms of timing and performance, the transaction to be rolled back may be selected based on it:
- being the youngest transaction, assuming that this transaction has the least operations to undo in the rollback;
- having the least updates, assuming that this transaction has the least impact on the database; or
- having the most updates to be made, assuming that the transaction will have a bigger impact on the database in the future but has made little impact on the database so far, so let the bigger impact occur later and be uninterrupted.

Theory: Concurrency Control

Timestamp protocol

Definition

A **timestamp** is a unique identifier that is assigned to a transaction or transaction operations.

How it works

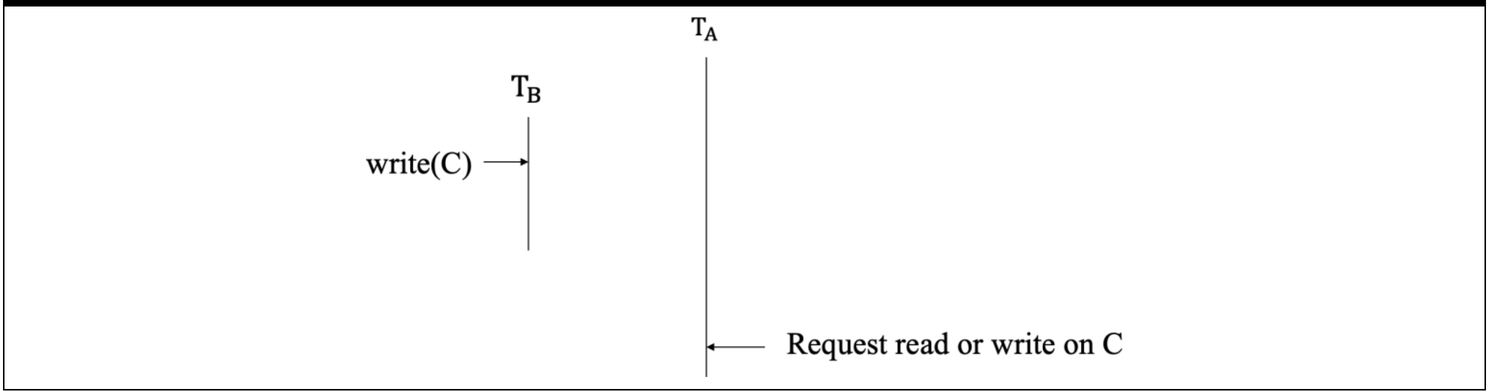
Timestamps can be assigned to:

- transactions when they enter the system;
- read/write operations performed by transactions; and
- when an object was last read from or written to.

For any given request, the system compares the timestamp of the requesting transaction with the timestamp of the transaction that last retrieved or updated that object.

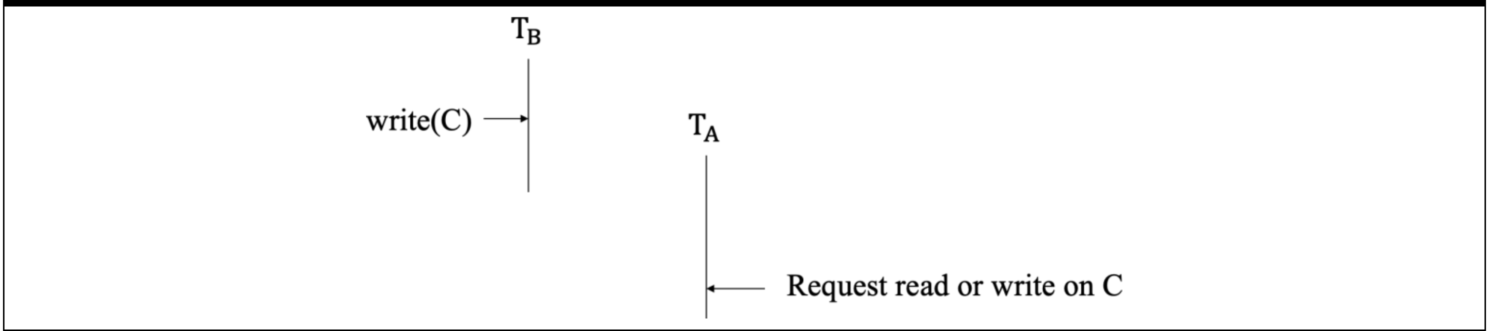
Conflicts occur if a transaction requests to read an object last read or updated by a younger transaction. They can be resolved by restarting the requesting transaction and assigning it a new timestamp.

Diagram: Example of timestamps



In the example above, transaction $\mathbf{T_A}$ is older than transaction $\mathbf{T_B}$ and the last write operation is performed by transaction $\mathbf{T_B}$. This means that transaction $\mathbf{T_A}$ started before being able to read the last write operation on the object `C`. As a result, it is necessary to restart transaction $\mathbf{T_A}$ by rolling it back when the request to read or write on the object `C` is performed.

Diagram: Example of timestamps



In the example above, transaction $\mathbf{T_A}$ is younger than transaction $\mathbf{T_B}$ and the last write operation is performed by transaction $\mathbf{T_B}$. This means that transaction $\mathbf{T_A}$ started is able to read the last write operation on the object `C`. As a result, no rollbacks are required.

Theory: Concurrency Control

Evaluation of protocols

	Locking	Timestamping
Advantages	**Two-phase locking (2PL) ensure serialisability** as after a transaction releases a lock, the transaction must never obtain any new locks.	**Ensures serialisability** as transactions are ordered by precedence based on their timestamp.
		Ensures freedom from deadlock as no transaction ever waits.
Disadvantages	**Can cause waits**, although a good deadlock prevention protocol can minimise the waiting time.	**Can cause rollbacks** if a transaction starts and is unable to proceed as an older transaction accesses the same object.

Practical: Concurrency Control

Complex queries

Example: Using the `NVL` function and joins

Write a single select statement to output all the customer names, their customer city, the total of the customer's balances from their deposits.

SQL Statement

```
SELECT Customer.customerName,
       customerCity,
       NVL2(SUM(balance),SUM(balance),0) "Total Balance"
FROM Customer LEFT JOIN Deposit
ON Deposit.customerName = Customer.customerName
GROUP BY Customer.customerName, customerCity;
```

Output

```
CUSTOMERNAME  CUSTOMERCITY   Total Balance
------------  ------------   -------------
Jones         Nottingham              4200
Ahmed         Derby                    480
Patel         Nottingham               520
Smith         Leicester                600
Chan          Nottingham                 0
Braun         Derby                   2150
```

In the example above, the function `NVL2` replaces all NULL values with another output, such as `0`.

Practical: Concurrency Control

Example: Outputting text from `SELECT` statement

Write a single `SELECT` statement to output the following:

- **if a customer has no deposit and no loan, list the customer name and in a second column, headed status, print 'Has no account and no loan';**
- **if a customer has a deposit but no loan, list the customer name and in a second column, headed status, print "Has an account but no loan';**
- **if a customer has a loan but no deposit, list the customer name and in a second column, headed status, print "Has a loan but no account'; and**
- **if a customer has a deposit and a loan, list the customer name and in a second column, headed status, print "Has both an account and a loan'.**

SQL Statement

```
SELECT customerName, 'Has no account and no loan' Status
FROM Customer
WHERE customerName NOT IN
      ( SELECT customerName
        FROM Deposit )
AND customerName NOT IN
      ( SELECT customerName
        FROM Loan )

UNION

SELECT customerName, 'Has an account but no loan' Status
FROM Customer
WHERE customerName IN
      ( SELECT customerName
        FROM Deposit )
AND customerName NOT IN
      ( SELECT customerName
        FROM Loan )

UNION

SELECT customerName, 'Has a loan but no account' Status
FROM Customer
WHERE customerName NOT IN
      ( SELECT customerName
        FROM Deposit )
AND customerName IN
      ( SELECT customerName
        FROM Loan )

UNION

SELECT customerName, 'Has both an account and a loan' Status
FROM Customer
WHERE customerName IN
      ( SELECT customerName
        FROM Deposit )
AND customerName IN
      ( SELECT customerName
        FROM Loan );
```

Output

```
CUSTOMERNAME  STATUS
------------  ------------------------------
Ahmed         Has both an account and a loan
Braun         Has an account but no loan
Chan          Has a loan but no account
Jones         Has both an account and a loan
Patel         Has both an account and a loan
Smith         Has both an account and a loan
```

Practical: Concurrency Control

Example:

Write a single `SELECT` statement to output all the customer names, their customer city, the total of the customer's balances from their deposits, and the total of the customer's amount from their loans, where the total balances is less than the total amounts.

SQL Statement

```
// to do
```

Output

```
CUSTOMERNAME  CUSTOMERCITY  Total Balance  Total
------------  ------------  -------------  -----
Jones         Nottingham             4200   5000
Ahmed         Derby                   480   1800
Patel         Nottingham              520   1000
Smith         Leicester               600   5500
Chan          Nottingham                0   2500
```

Theory: OODBMS & ORDBMS 1

Why use an object-oriented approach?

Failure of conventional databases

Conventional databases are unable to support:

- A large number of complex types, each with a few instances;
- propagation of dynamic design and changes throughout many design objects, instead lengthy and complex operations written in stored procedures/functions must be created;
- Requires version history to be maintained as there may be a large number of developers working in parallel and therefore problems can occur when attempting to merge changes together.

Conventional databases also:

- require version history to be maintained as there may be a large number of developers working in parallel and therefore problems can occur when attempting to merge changes together;
- have long duration transactions – a checkout of the database may be made by a developer in order for them to work on the design of the database, this transaction could take hours/days before it is committed back to the database.

Whereas in an object-oriented approach, the data, attributes and behaviour are together with the object.

Limitations of relational databases

In relational databases, the process of normalisation creates entities that do not correspond to entities in the real world. This leads to many joins during query processing, and complex construction and deletion of data structures.

Relational databases only support passive data, in which the behaviour of the data is not related to the data. Whereas, an object-oriented approach allows both the structure of complex objects and their behaviour (operations) to be specified, including query structure.

Relational Databases (RDBs)	Object-oriented Programming Languages (OOPLs)
Support storage of data in tables (rows and columns) with constraints to enforce data integrity and use DML and DDL based on relational calculus.	Support building applications out of objects that have both data and behaviour based on SE principals.
Join rows from different tables using `SELECT` queries.	Transverse objects via associations; objects collaborate by using each other’s operations.

Relational databases are also limited as:

- all column types are same in each row;
- all rows have the same number of columns;
- they have column types which are single-valued or atomic;
- they are poor on recursive queries; and
- they use more code than OODBs.

Theory: OODBMS & ORDBMS 1

Industry demand

An object-oriented approach was born out of demand by the industry for a solution where:

- the data, attributes and behaviour are together with the object;
- there are hierarchical designs created by inheritance;
- the design is not static, but evolves with time and allows design changes to be propagated through the database, as some actions cannot be foreseen at the start; and
- supports type-specific behaviour.

Applications of this approach can be found in databases used for computer aided design, image and geographic processing, geometry and data mining. This may include large databases of digital images, video and other complex data, such as satellite images and medical images.

Theory: OODBMS & ORDBMS 1

RDB spatial databases

Definitions

A **spatial database** is a database that is optimised for storing and querying data that represents objects defined in a geometric space.

An **RDB spatial database** is a relational database that has been enhanced with object types.

Example

A possible use of an RDB spatial database is to store data about polygons (straight sided shapes).

An RDB spatial database could store data about these polygons using the following tables:

```
Polygon(polyId, polyName)
Line(lineId, polyLeft, polyRight, fromNode, toNode)
Point(pointId, x, y)
```

The `Polygon` table has the following columns:

- `polyId` – stores the ID number for the polygon; and
- `polyName` – stores the name for the polygon.

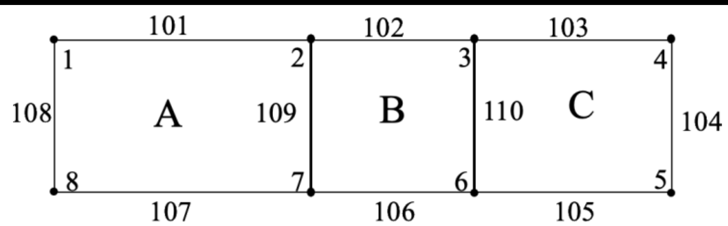
The `Line` table has the following columns:

- `lineId` – stores the ID number for the line;
- `polyLeft` – stores the name of the polygon directly to the left of the line (`0` if no polygon exists);
- `polyRight` – stores the name of the polygon directly to the right of the line (`0` if no polygon exists);
- `fromNode` – stores the ID of the point from which the line started; and
- `toNode` – stores the ID of the point to which the line terminates.

The `Point` table has the following columns:

- `pointId` – stores the ID number for the point;
- `x` – stores a `x`-coordinate of the point's physical location;
- `y` – stores a `y`-coordinate of the point's physical location; and

Diagram: Polygons in an RDB spatial database



The diagram above shows three rectangles `A`, `B` and `C`.

The table below shows the data that may be stored about the rectangles.

Polygon Table	Line Table	Point Table
(1, A), (2, B), (3, C)	(101, 0, A, 1, 2), (109, B, A, 2, 7), (102, ...	(1, 150, 350), (2, 170, 350), (3, 180, 350), ...

Theory: OODBMS & ORDBMS 1

In order to delete a polygon, multiple operations must take place, as seen in the example below.

Example: Deleting a polygon

Write the SQL statements that removes Polygon A entirely.

```
DELETE FROM Poly       // delete the reference to the polygon
WHERE polyName = 'A';


DELETE FROM Line                                // delete the lines surrounding
WHERE (polyLeft = 'A' AND polyRight = '0') OR  // the polygon but keep the lines
      (polyRight = 'A' AND polyLeft = '0');    // that belong to other polygons

                                                // achieved by deleting lines that
                                                // surround polygon A and no other
                                                // polygons


DELETE FROM Point                     // delete the points on the polygon but keep
WHERE pointID NOT IN                  // the points that belong to other polygons
      ( ( SELECT DISTINCT fromNode
          FROM Line                   // achieved by deleting points that are no
          UNION                       // longer a "fromNode" or a "toNode" of any
          ( SELECT DISTINCT toNode    // line
            FROM Line ) );
```

In this example, it would not be possible to use foreign keys as the complexity of the deletes. It would be possible to use `BEFORE` triggers to achieve the same impact as the `DELETE` statements, however the logic of triggers can become complex.

This shows that performing operations on these polygons requires multiple operations as the data and behaviour of the data are not linked. There is logic involve in the `DELETE` statements in the example above and the complexity of this logic is likely to increase as objects become more sophisticated.

In order to avoid this complex logic, it is more sensible to use an OODBMS that supports these types natively. In which case, the deletion of a polygon would automatically propagate to all objects of that polygon.

Theory: OODBMS & ORDBMS 1

OODBMS

Definition

An **object-oriented database management system (OODBMS)** is a database management system that supports the creation and modeling of data as objects. OODBMS also includes support for classes of objects and the inheritance of class properties, and incorporates methods, subclasses and their objects.

Objects and object types

An **object** is a uniquely identifiable entity that contain:

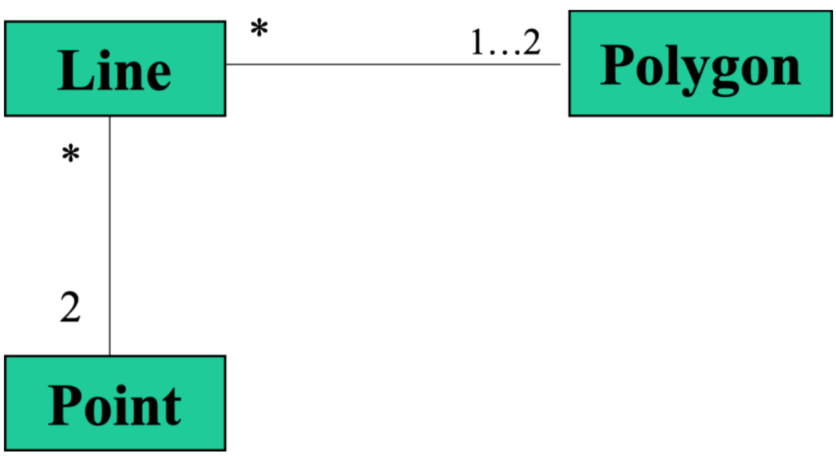
- **attributes** – values that describe the object's state;
- **operations** – define the objects' behaviour; and
- **associations** – links to other objects.

Every object has an **object type** (or **class**) that defines the attributes, operations and association.

OODB spatial database

A **class diagram** is a type of static structure diagram that describes the structure of an OODB by showing the object types and their attributes, operations, and associations.

Class diagram: Links in OODB spatial database



The class diagram above shows three object types: `Polygon`; `Line`; and `Point`. The links between the object types are shown and it is enforced that:

- a `Polygon` can have one or many `Line`;
- a `Line` can have only one or two `Polygon`;
- a `Point` can have one or many `Line`; and
- a `Line` can have only exactly two `Polygon`.

It is possible for an object type to inherit properties and operations from another object type. For example, an object type `Square` may inherit from an object type `Polygon`.

Theory: OODBMS & ORDBMS 1

The class diagram can be extended to show the class declarations.

Class diagram: Class declarations in OODB spatial database	Explanation
Class `Polygon`	

Attributes:
`polyId: integer`
`polyName: string`
`itsLines: list of line objects`
Operations:
`new( )`
`store( )`
`retrieve (polygonId: integer) : polygon`
`delete( )`
`resize(scaleFactor: real)`
`getArea() : real`
	The type `Polygon` has the attributes `polyId` and `polyName` which are effectively equivalent to using column names in a relational database.
	
	The attribute `itsLines` acts as an association and is an addition to how this object type would be represented in a relational database. This is required as to allow the objects of object type `Polygon` to be associated with lines that surround the shape.

Class `Line`	

Attributes:
`lineId: integer`
`fromNode: Point`
`toNode: Point`
`polyLeft: Polygon`
`polyRight: Polygon`
Operations:
`new( )`
`store( )`
`retrieve (lineId: integer) : line`
`delete( )`
`resize(scaleFactor: real)`
`getLength() : real`
	The type `Line` has the attributes `lineId`, `fromNode`, `toNode`, `polyLeft` and `polyRight` which are effectively equivalent to using column names in a relational database.

Class `Point`	

Attributes:
`pointId: integer`
`xLocation: real`
`yLocation: real`
`itsLines: list of line objects`
Operations:
`new( )`
`store( )`
`retrieve (pointId: integer) : point`
`delete( )`
`move(x: real, y:real)`
	The type `Point` has the attributes `pointId`, `xLocation` and `yLocation` which are effectively equivalent to using column names in a relational database.
	
	The attribute `itsLines` acts as an association and is an addition to how this object type would be represented in a relational database. This is required as to allow the objects of object type `Point` to be associated with the lines on which it sits.

It is possible for attributes to have normal data types, such as `integer`, but also to have their data type be an object type, such as `Point`.

The operations shown in the class diagram require definitions. Definitions include code that performs the operations and are sometimes written in object-oriented programming languages.

An explanation of some of the operations in the object types can be seen in the table below.

Theory: OODBMS & ORDBMS 1

Operation Name	Purpose
`new`	Instantiates an object instance.
`delete`	Destroys an object instance.
`store`	Takes an existing polygon and stores it.
`retrieve`	Returns any stored polygon object with the specified ID.
`resize`	Allows the `Polygon` object to be resized by moving its lines and points.
`getArea`	Returns the area of the `Polygon` object.

These operations allow propagation of dynamic design and changes.

For example, the deletion of a polygon would automatically propagate to all objects of that polygon. The operation `delete` in the `Polygon` class will call operations in the `Line` class, which in turn will call operations in the `Point` class. As a result, this avoids the increasingly complex logic of forming valid `DELETE` statements, as shown on page 174.

These operations form an intrinsic, natural property of the class model. This allows the analyst to think about how a `Line` will delete its points separately from what the `Polygon` is doing.

Evaluation of OODBMS

Advantages	Disadvantages
Encapsulation of both state and behaviour is a more natural as it is a realistic representation of real-world entities and attributes and behaviour are considered together.	**No theoretical background** to the data model; there is no relational algebra equivalent and therefore less possibility to manipulate queries for efficiency.
Inheritance allows complex data types to be built where a data type has different relationships with other data types that may form a hierarchy.	**Lack of universal approach and portability** as there are many different types of OODBMS; the Object Data Management Group (ODMG) set the standard for how object-oriented databases should behave, however this is not always followed in the same manner. However, the standards have since been revised by the Object Management Group (OMG).
More expressive query language; collections allow multiple objects to be considered together, such as a collection of polygons, and navigation allows information about objects to be easily retrieved using operations, such as "find all polygons where the lineId is 5".	**Complexity** in database management.
Improved performance.	**Query optimisation compromises encapsulation**; object-oriented programming languages often have private attributes that can only be accessed by getter and setter operations, however this is not implemented in OODBMS as to improve query performance.
Supports long-duration transactions.	**Impedance mismatch**. SQL is a declarative language that handles collections of rows of data, while low-level object-oriented programming languages (OOPLs) such as C++, Java and C# that require code to only handle one row of data at a time. Therefore, there must be a translation between objects and tables. SQL and OOPLs have different built-in data types, and therefore there must be a translation between the database and programming language representations.

Theory: OODBMS & ORDBMS 1

Strategies for OODB development

Strategies

An object-oriented database (OODB) can be produced by:

- extending the syntax of an existing object-oriented (OO) programming language;
- provide additional libraries for an OODB;
- extend SQL with object-oriented (OO) features, an example of this is ORDBMS; and
- develop a new object-oriented (OO) programming language with OODB capabilities, however this is the least-taken approach as industry tends to stick with existing programming languages quickly.

It is also necessary to consider the **lifetime** of objects, they can be:

- persistent – stored in a database for use in subsequent application sessions; or
- transient – exist for the lifetime of the application.

Theory: OODBMS & ORDBMS 1

ORDBMS

Definition

An **object-relational database management system (ORDBMS)** is an evolution of relational database management systems (RDBMSs) that allow user-defined types (UDTs), support inheritance and complex data types to be used in tables. This contrasts with OODBMS that stores the objects directly.

What is a user-defined type (UDT)?

A **user-defined type (UDT)** is a data type that derived from an existing data type. They can be used to extend the built-in types already available and create customised data types.

The power of using UDTs is in the more general case when the UDT consists of several attribute ("row") definitions and several procedure definitions.

Creating a UDT

Syntax: Creating an object type

```
CREATE TYPE <object type name> AS OBJECT (
      <attribute name> <data type>,
      <attribute name> <data type>,
      …
      <attribute name> <data type>
      MEMBER [PROCEDURE/FUNCTION] <operation name>[(argument(s))] RETURN <data type>
      MEMBER [PROCEDURE/FUNCTION] <operation name>[(argument(s))] RETURN <data type>
      …
      MEMBER [PROCEDURE/FUNCTION] <operation name>[(argument(s))] RETURN <data type> )
      [FINAL / NOT FINAL];
/
```

When creating an object type it is necessary to specify the attributes (or members) and their respective data type.

It is possible to declare the member methods (or operations) by specifying their name and respective data type. These are similar to stored procedures/functions; however, they belong to the object type.

If the `FINAL` clause is included, subclasses are not allowed to inherit attributes and member methods from the object type.

If the `NOT FINAL` clause is included, subclasses are allowed to inherit attributes and member methods from the object type.

Theory: OODBMS & ORDBMS 1

Example: Creating an object type

Write an SQL statement to create the object type `Polygon` with the attributes: `polyID` as an `INTEGER` and `polyName` as a `VARCHAR` with length 20.

The object type should also have the member function `getArea` which returns a value of data type `REAL`.

Allow other object types to inherit from `Polygon`.

```
CREATE TYPE Polygon AS OBJECT (
      polyID INTEGER,
      polyName VARCHAR(20),
      MEMBER FUNCTION getArea RETURN REAL )
      NOT FINAL;
/
```

The object type created in the example above can be deleted by performing the statement

```
      DROP TYPE Polygon;
```

Creating an object type body

The object type may contain declarations of member methods. It is necessary to define the behaviour of these member methods in the body of the object type.

Syntax: Creating an object type body with a member method

```
CREATE TYPE BODY <object type name> AS
      MEMBER [PROCEDURE/FUNCTION] <member method name> RETURN <data type> IS
            /* code to declare variables */
            BEGIN
                  /* member method code (PL/SQL) */
            END <member method name>;
END;
/
```

It is possible to define multiple member methods using the `CREATE TYPE BODY` statement. The object type body created in the example above can be deleted by performing the statement

```
      DROP TYPE BODY Polygon;
```

Example: Creating an object type

Write an SQL statement to create the object type body for the object type `Polygon`.

The object type should contain the definition for the member function `getArea`.

```
CREATE TYPE BODY Polygon AS
      MEMBER FUNCTION getArea RETURN REAL IS
            BEGIN
                  /* code to calculate area of regular polygon */
                  RETURN area;
            END getArea;
END;
/
```

UDT composition

A UDT can be composed of other UDTs.

Example: Creating an object type composed of other object types

Write an SQL statement to create the object type `Line` with the attributes: `lineID` as an `INTEGER`, `polyLeft` as a `Polygon`, `polyRight` as a `Polygon`, `fromNode` as a `Point` and toNode as a `Point`.

Do not allow other object types to inherit from `Line`.

```
CREATE TYPE Line AS OBJECT (
      lineID INTEGER,
      polyLeft Polygon,
      polyRight Polygon,
      fromNode Point,
      toNode Point )
      FINAL;
/
```

In the example above:

- the attributes `polyLeft` and `polyRight` uses the object type `Polygon` as their data type; and
- the attributes `fromNode` and `toNode` uses the object type `Point` as their data type.

Example: Creating an object type composed of other object types

Write an SQL statement to create the object type `Line` with the attributes: `lineID` as an `INTEGER`, `polyLeft` as a Polygon, `polyRight` as a `Polygon`, `fromNode` as a `Point` and toNode as a `Point`.

Do not allow other object types to inherit from `Line`.

```
CREATE TYPE Point AS OBJECT (
      pointID INTEGER,
      xLocation INTEGER,
      yLocation INTEGER )
      FINAL;
/
```

In the example above the attribute `itsLine` uses the object type `Line` as its data type.

Creating an array type and UDT associations

It is possible to create an object type that is an array of another object types.

Syntax: Creating an array type

```
CREATE TYPE <object type name> IS VARRAY(<length>) OF <object type in array>;
/
```

Example: Creating an array type

Write an SQL statement to create the object type `lines_array` that is an array of the object type `Line` with length `4`.

```
CREATE TYPE lines_array IS VARRAY(4) OF Line;
/
```

Now that the `lines_array` object type has been created it can be added to the previous `CREATE TYPE` statement for the `Polygon` object type, as seen below.

```
CREATE TYPE Polygon AS OBJECT (
      polyID INTEGER,
      polyName VARCHAR(20),
      itsLines lines_array,
      MEMBER FUNCTION getArea RETURN REAL )
      NOT FINAL;
/
```

As a result, there is a two-way association between the `Polygon` object type and the `Line` object type in which they are aware of each other.

This association differs to primary key and foreign key relationships, in which:

- each primary key row knows about its “child” foreign key rows; and
- each foreign key row knows about its “parent” primary key row.

Whilst the two-way association has no “parent” and “child” relationship and therefore avoids possible problems when creating tables with primary keys and foreign key loops in which “parent” rows can become a “child” row of itself.

Furthermore, the `lines_array` object type can also be added to the previous `CREATE TYPE` statement for the `Point` object type, as seen below.

```
CREATE TYPE Point AS OBJECT (
      pointID INTEGER,
      xLocation INTEGER,
      yLocation INTEGER,
      itsLines lines_array )
      FINAL;
/
```

Using UDTs in tables

Tables can use object types in column definitions. This is necessary as object types cannot be used by themselves, they must be stored persistently.

Example: Creating a table with object type columns

Create a table named `StoredShapes` with a `shape` column with data type `Polygon`, a `creationDate` with data type `DATE` and `author` column to take strings up to length `30`.

```
CREATE TABLE StoredShapes (
      shape Polygon,
      creationDate DATE,
      author VARCHAR(30)
);
```

Theory: OODBMS & ORDBMS 1

It is then possible to `SELECT` the attributes of an object type from a table by using an alias.

Syntax: Selecting object type attributes from a table

```
SELECT <alias name>.<column name>.<attribute>
FROM <table name> <alias name>;
```

The alias name may be arbitrary.

Example: Selecting object type attributes from a table

Write an SQL statement to select the `polyName` of `shape` stored in the table `StoredShapes`.

```
SELECT n.shape.polyName
FROM StoredShapes n;
```

In the example above, the alias name used is `n`.

SQL DML for UDTs in tables

When inserting values in to a table that uses object types in its column definitions, it is necessary to use the object type's constructor which takes the attributes of the object type as parameters and creates the object.

Syntax: Inserting data (object types)

```
INSERT INTO <table name> VALUES (
      <constructor name>(<values>),
      ...
      ...
);
```

The name of the constructor is the same as the name of the object type.

Example: Inserting data (object types)

Write an SQL statement to insert a record in to the `StoredShapes` table which contains a `Polygon` object with `polyID` 9 and `polyName` A as the `shape`, `2004-11-24` as the `creationDate` and `Smith` as the `author`.

```
INSERT INTO StoredShapes VALUES (
      Polygon(9, 'A'),
      '24-NOV-04',
      'Smith'
);
```

In the example above, the value for the column `shape` is created by the constructor function for the `Polygon` object type.

The constructor function for the `Polygon` object type takes the following parameters:

- `polyID` – `9`; and
- `polyName` – `A`.

Theory: OODBMS & ORDBMS 1

Calling member methods

It is possible to call member methods in a `SELECT` statement by using an alias.

Syntax: Calling member methods in a `SELECT` statement

```
SELECT <alias name>.<column name>.<member function name>([<parameter(s)>])
FROM <table name> <alias name>;
```

The alias name may be arbitrary.

Example: Calling member methods in a `SELECT` statement

Write a `SELECT` statement that returns the area for all shapes in the table `StoredShapes`.

```
SELECT n.shape.getArea()
FROM StoredShapes n;
```

In the example above, the alias name used is `n`.

Practical: OODBMS & ORDBMS 1

Object types

Example: Creating an object type

Create an object type called `personType` with attributes `dateOfBirth` (`DATE`), firstName (`VARCHAR(5)`), `lastName` (`VARCHAR(15)`), and a member function called `getAge` that returns an `INTEGER`. The object type should be set `FINAL` as it is not going to have a sub-type.

PL/SQL Code

```
CREATE TYPE personType AS OBJECT (
      dateOfBirth DATE,
      firstName VARCHAR(5),
      lastName VARCHAR(15),
      MEMBER FUNCTION getAge RETURN INTEGER )
      FINAL;
/
```

Example: Creating an object type body

Create a type body for `personType` to contain the implementation of the member function `getAge`.

The algorithm for `getAge` is:

- **Select or assign today's date into/to a variable called `todaysDate`.**
- **Evaluate the age by taking the year from `todaysDate` and subtracting the year from the attribute `dateOfBirth`.**
- **Return the age.**

PL/SQL Code

```
CREATE TYPE BODY personType AS
      MEMBER FUNCTION getAge RETURN INTEGER IS
            todaysDate DATE;
            age INTEGER;
            BEGIN
                  todaysDate:= SYSDATE();
                  age := EXTRACT( YEAR FROM todaysDate ) - EXTRACT( YEAR FROM
                  dateOfBirth );
                  RETURN age;
            END getAge;
END;
/
```

Practical: OODBMS & ORDBMS 1

Object types in tables

Example: Creating a table with an object type column

Create a table `newCustomer` (that stores products bought by a customer) with the following columns:

```
person PersonType,
product VARCHAR(15),
cost DECIMAL(6,2)
```

SQL Statement

```
CREATE TABLE newCustomer (
      person PersonType,
      product VARCHAR(15),
      cost DECIMAL(6,2)
);
```

Example: Inserting data in to a table with object type columns

Insert into the table `newCustomer` a row containing a `personType` with date of birth `‘26-OCT-60’`, first name `‘Chris’`, last name `‘Smith’`, and product `‘Computer’` and cost `534.99`.

SQL Statement

```
INSERT INTO newCustomer VALUES (
      personType('26-OCT-60', 'Chris', 'Smith'),
      'Computer',
      534.99
);
```

Example: Selecting data from a table with object type columns

Select the `getAge` member function to get the age of `Chris Smith` and also select the product `Chris Smith` is buying.

SQL Statement

```
SELECT n.person.getAge(), n.product
FROM newCustomer n
WHERE n.person.firstName = ‘Chris’
AND n.person.lastName = ‘Smith’;
```

Theory: OODBMS & ORDBMS 2

Inheritance and polymorphism

Inheritance

A **superclass** is the parent object type of a subtype.

A **subtype** is the child object type of a type.

Inheritance is where a subtype related in some way to a type is able to inherit the attributes and operations of its type and may have its own attributes and operations.

Inheritance should be used when the “is a” rule is satisfied; this states that the type and subtype should have a relationship, for example a dog is a type of animal.

Polymorphism

Polymorphism allows an object to be processed differently depending on its class, this uses overriding.

Overriding occurs when a method is created in a subtype with the same name as the method in the type so as to override the functionality declared in the superclass.

Uses of inheritance

Use	Description	Example
Specialisation	One object type can be defined as a special case of another by: • adding more attributes and operations; or • overriding operations.	A type `Car` may have subtypes `Racing Car`, `MPV` and `Estate`.
Generalisation	Many object types can be designed as a general case by collecting common attributes and operations in to a more general object type.	The types `Racing Car`, `MPV` and `Estate` are likely to share some common attributes and operations. Therefore they can become subtypes of a type `Car`.

Theory: OODBMS & ORDBMS 2

Inheritance in an RDB spatial database

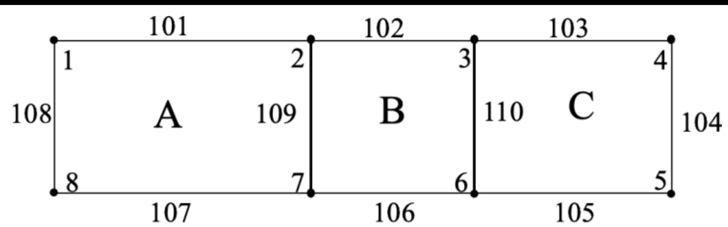
Recap

The idea of an RDB spatial database was introduced on page 173.

The RDB spatial database stored data about polygons using the following tables:

```
Polygon(polyId, polyName)
Line(lineId, polyLeft, polyRight, fromNode, toNode)
Point(pointId, x, y)
```

Diagram: Polygons in an RDB spatial database



The diagram above shows three rectangles `A`, `B` and `C`.

The table below shows the data that may be stored about the rectangles.

`Polygon` Table	`Line` Table	`Point` Table
(1, A), (2, B), (3, C)	(101, 0, A, 1, 2), (109, B, A, 2, 7), (102, ...	(1, 150, 350), (2, 170, 350), (3, 180, 350), ...

Using subtypes in an RDB spatial database

The previous implementation of storing data about the polygons did not allow overlapping.

In order to allow the polygons to overlap, a subtype `UnsharedLine` can be created based on the `Line` type. As such, the RDB spatial database would now store data about polygons using the following tables:

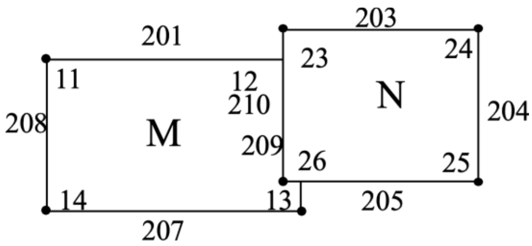
```
Polygon(polyId, polyName, displayLevel)
UnsharedLine(lineId, polyOwner, fromNode, toNode)
Point(pointId, x, y)
```

The subtype `UnsharedLine` shares many of the same attributes as the `Line` type as these are inherited. However, the attributes `polyLeft` and `polyRight` have been replaced with `polyOwner` as to the lines may not have a polygon directly to the left or right if the polygons are overlapping.

`polyOwner` is the `polyName` of the polygon to which the line belongs.

Theory: OODBMS & ORDBMS 2

Diagram: Overlapping polygons in an RDB spatial database



The diagram above shows two rectangles `M`, and `N`.

The table below shows the data that may be stored about the rectangles.

`Polygon` Table	`Line` Table	`Point` Table

`(1, M, 7)`
`...`
	`(201, M, 11, 12),`
	`(203, N, 23, 24),`
	`(209, ...`
		`(11, 160, 360),`
		`(12, 180, 360),`
		`(23, 170, 380),`
		`...`

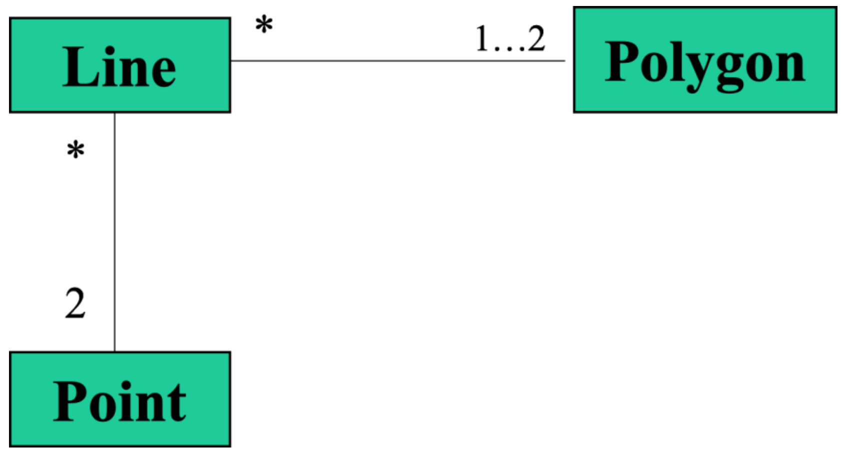
Theory: OODBMS & ORDBMS 2

Inheritance in an OODBMS

Recap

The original class diagram for the OODB spatial database was introduced on page 176.

Class diagram: Links in OODB spatial database



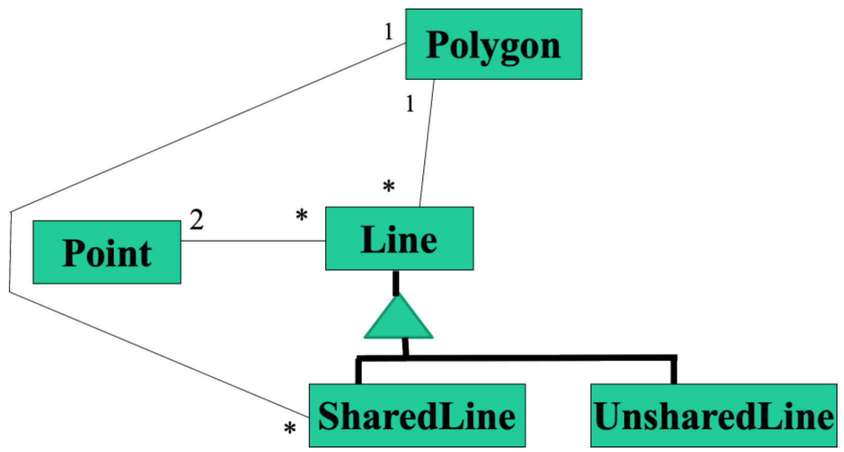
The class diagram above shows three object types: `Polygon`; `Line`; and `Point`. The links between the object types are shown and it is enforced that:

- a `Polygon` can have one or many `Line`;
- a `Line` can have only one or two `Polygon`;
- a `Point` can have one or many `Line`; and
- a `Line` can have only exactly two `Polygon`.

Using subtypes in an OODB spatial database

The addition of the subtypes `SharedLine` and `UnsharedLine` of type `Line` can be shown in the class diagram.

Class diagram: Links in OODB spatial database



In the class diagram above, the triangle notation shows that the subtypes `SharedLine` and `UnsharedLine` inherit from the type `Line`.

Theory: OODBMS & ORDBMS 2

The class diagram enforces that:

- a `Polygon` can have one or many `Line`;
- a `Polygon` can have one or many `UnsharedLine`;
- a `Polygon` can have two or many `SharedLine`;
- a `Line` can have only one or two `Polygon`;
- a `Point` can have one or many `Line`;
- a `Line` can have only exactly two `Polygon`.

The class diagram from page 177 can be extended to show the new subtypes.

Class diagram: Class declarations in OODB spatial database	Explanation

Class `Polygon`
Attributes:
`polyId: integer`
`polyName: string`
`displayLevel: integer`
`itsLines: list of line objects`
Operations:
`new( )`
`store( )`
`retrieve (polygonId: integer) : polygon`
`delete( )`
`resize(scaleFactor: real)`
`getArea() : real`
	The type `Polygon` has the attributes `polyId` and `polyName` which are effectively equivalent to using column names in a relational database.
	
	The attribute `itsLines` acts as an association and is an addition to how this object type would be represented in a relational database. This is required as to allow the objects of object type `Polygon` to be associated with lines that surround the shape.
	
	The attribute `displayLevel` has been added.

Class `Line`
Attributes:
`lineId: integer`
`fromNode: Point`
`toNode: Point`
Operations:
`new( )`
`store( )`
`retrieve (lineId: integer) : line`
`delete( )`
`resize(scaleFactor: real)`
`getLength() : real`
	The type `Line` has the attributes `lineId`, `fromNode` and `toNode` which are effectively equivalent to using column names in a relational database.
	
	The attributes `polyLeft` and `polyRight` have been removed as they are not used in all subtypes of `Line`.

Class `Point`
Attributes:
`pointId: integer`
`xLocation: real`
`yLocation: real`
`itsLines: list of line objects`
Operations:
`new( )`
`store( )`
`retrieve (pointId: integer) : point`
`delete( )`
`move(x: real, y:real)`
	The type `Point` has the attributes `pointId`, `xLocation` and `yLocation` which are effectively equivalent to using column names in a relational database.
	
	The attribute `itsLines` acts as an association and is an addition to how this object type would be represented in a relational database. This is required as to allow the objects of object type `Point` to be associated with the lines on which it sits.

Theory: OODBMS & ORDBMS 2

Class **SharedLine**	
Attributes: polyLeft: Polygon polyRight: Polygon	The type SharedLine shares all of the same attributes and operations as the type Line, but adds the attributes polyLeft and polyRight.
Operations:	

Class **UnsharedLine**	
Attributes: polyOwner: Polygon	The type UnsharedLine shares all of the same attributes and operations as the type Line, but adds the attribute polyOwner.
Operations:	

It is possible for attributes to have normal data types, such as integer, but also to have their data type be an object type, such as Point.

The operations shown in the class diagram require definitions. Definitions include code that performs the operations and are sometimes written in object-oriented programming languages.

Theory: OODBMS & ORDBMS 2

Inheritance in an ORDBMS

Creating a type

Inheritance from a type can be controlled using different clauses at the end of the `CREATE TYPE` statement:

- **`FINAL` clause** – does not allow subtypes to inherit from the type; and
- **`NOT FINAL` clause** – allows subtypes to inherit from the type.

This option can be changed after the creation of the type by using an `ALTER TYPE` statement:

```
ALTER TYPE <object type name> NOT FINAL
CASCADE CONVERT
TO SUBSTITUTABLE;
```

If this is disallowed by the database, the object type can be force dropped using the statement below and re-created as `NOT FINAL`.

```
DROP TYPE <object type name> FORCE;
```

Example: Creating a `NOT FINAL` object type

Write an SQL statement to create the object type `Line` with the attributes: `lineID` as an `INTEGER`, `polyLeft` as a `Polygon` and `polyRight` as a `Polygon`.

Allow other object types to inherit from `Line`.

```
CREATE TYPE Line AS OBJECT (
      lineID INTEGER,
      polyLeft Polygon,
      polyRight Polygon )
      NOT FINAL;
/
```

In the example above, the statement creates the type `Line` with the `NOT FINAL` clause. This is important as subtypes will be added later that are required to inherit from the type `Line`.

Creating a subtype

Syntax: Creating a subtype

```
CREATE TYPE <subtype name> UNDER <object type name> (
      <attribute name> <data type>,
      <attribute name> <data type>,
      …
      <attribute name> <data type>
      MEMBER [PROCEDURE/FUNCTION] <operation name>[(argument(s))] RETURN <data type>
      MEMBER [PROCEDURE/FUNCTION] <operation name>[(argument(s))] RETURN <data type>
      …
      MEMBER [PROCEDURE/FUNCTION] <operation name>[(argument(s))] RETURN <data type> )
      [FINAL / NOT FINAL];
/
```

Theory: OODBMS & ORDBMS 2

Example: Creating a subtype

Write an SQL statement to create the object type `UnsharedLine`, a subtype of `Line`, with the attribute `polyOwner` as a `Polygon`.

Do not allow other object types to inherit from `UnsharedLine`.

```
CREATE TYPE UnsharedLine UNDER Line (
      polyOwner Polygon )
      FINAL;
/
```

In the example above, the subtype `UnsharedLine` will inherit all of the attributes and operations from the type `Line`. However, the attribute `polyOwner` is added. The `FINAL` clause ensures that no subtypes can inherit from `UnsharedLine`.

Example: Creating a subtype

Write an SQL statement to create the object type `SharedLine`, a subtype of `Line`, with the attributes `polyLeft` as a `Polygon` and `polyRight` as a `Polygon`.

Do not allow other object types to inherit from `SharedLine`.

```
CREATE TYPE SharedLine UNDER Line (
      polyLeft Polygon,
      polyRight Polygon )
      FINAL;
/
```

In the example above, the subtype `SharedLine` will inherit all of the attributes and operations from the type `Line`. However, the attributes `polyLeft` and `polyRight` are added. The `FINAL` clause ensures that no subtypes can inherit from `SharedLine`.

Overriding describes redefining an operation inherited from a type to customise its behaviour in a subtype. This is achieved by creating an operation in the subtype with the same name and signature of the operation in the type.

For example, the following line could be added to the `CREATE TYPE` statement for the `UnsharedLine` type in order to create an overriding member function named `getLength`:

```
      OVERRIDING MEMBER FUNCTION getLength RETURN REAL
```

For overriding to work correctly, it is important that the member function in the type is also named `getLength` and has the return data type `REAL`.

Theory: OODBMS & ORDBMS 2

Using subtypes in tables

As seen before on page 183, tables can use object types in column definitions.

Example: Creating a table with object type columns

Create a table named `StoredLines` with a `borderLine` column with data type `Line`, a `creationDate` with data type `DATE` and `author` column to take strings up to length `30`.

```
CREATE TABLE StoredLines (
      borderLine Line,
      creationDate DATE,
      author VARCHAR(30)
);
```

It is then possible to `SELECT` the attributes of an object type from a table by using an alias.

Syntax: Selecting object type attributes from a table

```
SELECT <alias name>.<column name>.<attribute>
FROM <table name> <alias name>;
```

The alias name may be arbitrary.

Example: Selecting object type attributes from a table

Write an SQL statement to select the `lineId` of `borderLine` stored in the table `StoredLines`.

```
SELECT n.borderLine.polyId
FROM StoredLines n;
```

In the example above, the alias name used is `n`.

SQL DML for subtypes in tables

When inserting values in to a table that uses object types that have subtypes in its column definitions, it is possible to use the subtype's constructor which takes the attributes of the object type as parameters and creates the object. This will insert a subtype in to the location of a type.

Syntax: Inserting data (object types)

```
INSERT INTO <table name> VALUES (
      <subtype constructor name>(<values>),
      ...
      ...
);
```

The name of the constructor is the same as the name of the object type.

Theory: OODBMS & ORDBMS 2

Example: Inserting data (subtypes)

Write an SQL statement to insert a record in to the `StoredLines` table which contains:

- **an `UnsharedLine` object with `lineID` 19, `polyLeft` NULL, `polyRight` NULL and `polyOwner` as the shape in the `StoredShapes` table with `polyID` 9 as the `borderline`;**
- **`2004-11-24` as the `creationDate`; and**
- **`Smith` as the `author`.**

```
INSERT INTO StoredLines VALUES (
     UnsharedLine(19, NULL, NULL,
                  ( SELECT n.shape
                    FROM StoredShapes n
                    WHERE n.shape.polyId = 9 ) ),
     '24-NOV-04',
     'Smith'
);
```

In the example above, the value for the column `borderLine` is created by the constructor function for the `UnharedLine` object type.

The constructor function for the `UnsharedLine` subtype takes the following parameters:

- `lineID` – 19;
- `polyLeft` – NULL;
- `polyRight` – NULL; and
- `polyOwner` – the value of the `polyName` attribute in the `shape` column in the `StoredShapes` table where the `polyId` attribute in the `shape` column is 9.

Example: Inserting data (subtypes)

Write an SQL statement to insert a record in to the `StoredLines` table which contains:

- **an `UnsharedLine` object with `lineID` 20, `polyLeft` NULL, `polyRight` NULL and `polyOwner` as a new object of type `Polygon` with `polyId` 9, `polyName` E and `displayLevel` 2;**
- **`2016-11-16` as the `creationDate`; and**
- **`Smith` as the `author`.**

```
INSERT INTO StoredLines VALUES (
     UnsharedLine(19, NULL, NULL, Polygon(9, 'E', 2)),
     '24-NOV-04',
     'Smith'
);
```

Calling member methods of subtypes

This is demonstrated in the last example on page 200.

Practical: OODBMS & ORDBMS 2

Subtypes

The following examples are a continuation from the examples shown on pages 186-187.

Example: Modifying a type

Change the object type called `personType` created on page 186 so that it is declared as `NOT FINAL`, as it is now going to have two subtypes.

PL/SQL Code

```
ALTER TYPE personType NOT FINAL
CASCADE CONVERT
TO SUBSTITUTABLE;
```

OR

```
DROP TABLE newCustomer;

CREATE TABLE newCustomer (
      person PersonType,
      product VARCHAR(15),
      cost DECIMAL(6,2)
);

CREATE OR REPLACE TYPE personType AS OBJECT (
      dateOfBirth DATE,
      firstName VARCHAR(5),
      lastName VARCHAR(15),
      MEMBER FUNCTION getAge RETURN INTEGER )
      NOT FINAL;
/
```

Example: Creating subtypes

Create two subtypes of the `personType`:

- **`studentType` that has an additional attribute `'degreeCourse'`; and**
- **employeeType that has additional attributes `'jobRole'` and `'roleStartDate'`, and an additional operation `'durationInRole'` that returns the years in the job role since starting the role (i.e. by taking the year from the current date and subtracting the year from the attribute `roleStartDate`).**

PL/SQL Code

```
CREATE OR REPLACE TYPE studentType UNDER personType (
      degreeCourse VARCHAR(20) )
      FINAL;
/

CREATE OR REPLACE TYPE employeeType UNDER personType (
jobRole VARCHAR(20),
      roleStartDate DATE,
      MEMBER FUNCTION durationInRole RETURN INTEGER )
      FINAL;
/
```

Practical: OODBMS & ORDBMS 2

Example: Creating a subtype body
Create a type body for `employeeType` to contain the implementation of the member function `durationInRole`.
PL/SQL Code

```
CREATE TYPE BODY employeeType AS
     MEMBER FUNCTION durationInRole RETURN INTEGER IS
          todaysDate DATE;
          duration INTEGER;
          BEGIN
               todaysDate:= SYSDATE();
               duration := EXTRACT( YEAR FROM todaysDate ) - EXTRACT( YEAR FROM
               roleStartDate );
               REURN duration;
          END durationInRole;
END;
/
```

Practical: OODBMS & ORDBMS 2

Subtypes in tables

The following examples are a continuation from the examples shown on pages 186-187.

Example: Inserting data in to a table with subtype columns

Insert into the table NewCustomer a row containing a `studentType` and a row containing an `employeeType`.

The columns for `NewCustomer` were:

```
Person PersonType,
Product VARCHAR(15),
Cost DECIMAL(6,2)
```

SQL Statement

```
INSERT INTO newCustomer VALUES (
      personType('26-OCT-60', 'Chris', 'Smith'),
      'Computer',
      534.99
);

INSERT INTO newCustomer VALUES (
      studentType('16-MAY-70', 'Alex', 'Reed', 'Computer Science'),
      'Printer',
      240.50
);

INSERT INTO newCustomer VALUES (
      employeeType('09-JUL-80', 'Scott', 'Meyers', 'IT Manager', '10-NOV-15'),
      'Mobile',
      250.00
);
```

Example: Selecting data from a table with subtype columns

Select the `getAge` member function for all rows and then the `durationInRole` member function for all rows.

SQL Statement

```
SELECT n.Person.getAge(), n.person.firstName, n.person.lastName
FROM NewCustomer n;

SELECT TREAT(n.Person AS employeeType).durationInRole(), n.person.firstName,
       n.person.lastName
FROM NewCustomer n;
```

In the example above the `TREAT` function must be used as the database would not recognise `n.person.durationInRole()` for the `personType` column.

This is because the operation `durationInRole` is exclusive to the subtype `employeeType` and therefore the column of type `personType` must be considered as a column of type `employeeType` in order for the operation to be accessed by the `SELECT` statement.