# Fundamentals of Databases notes

Todd Davies

January 13, 2015

## Introduction

Databases are core, if largely invisible, components of modern computing architectures in both commercial and scientific contexts. The management of data has evolved from application-specific management of myriad files to organisation-wide approaches that see data as one of the most important assets of modern organisations and, as such, a key factor in their ability to compete and thrive. At this organisation-wide scale, database management systems (DBMSs) are the crucial piece of software infrastructure needed to achieve the desired results with consistent quality and robust efficiency. A modern DBMS is a thing of wonder and embodies in its internal construction and in its wide usability many advances in algorithms and data structures, programming language theory, conceptual modelling, concurrency theory, and distributed computing. This makes the study of databases a data-centric traversal of many of the most exciting topics in modern computing.

## Aims

The aim of this course unit is to introduce the students to the fundamental concepts and techniques that underlie modern database management systems (DBMSs).

The course unit studies the motivation for managing data as an asset and introduces the basic architectural principles underlying modern DBMSs. Different architectures are considered and the application environments they give rise to.

The course unit then devotes time to describing and motivating the relational model of data, the relational database languages, and SQL, including views, triggers, embedded SQL and procedural approaches (e.g., PL/SQL).

The students learn how to derive a conceptual data model (using the Extended Entity Relationship paradigm), how to map such a model to target implementation model (for which the relational model is used), how to assess the quality of the latter using normalisation, and how to write SQL queries against the improved implementation model to validate the resulting design against the data requirements originally posed. For practical work, the Oracle DBMS is used.

The course unit also introduces the fundamentals of transaction management including concurrency (e.g., locking, 2-phase locking, serialisability) and recovery (rollback and commit, 2-phase commit) and of file organisation (e.g., clustering) and the use of indexes for performance.

Finally, the course unit addresses the topic of database security by a study of threats and countermeasures available. In the case of the former, these include potential theft and fraud as well as loss of confidentiality, privacy, integrity and availability. In the case of the latter these primarily include mechanisms for authorization and access control, including the use of views for that purpose. The course unit also addresses the topic of legal frameworks that give rise to obligations on the part of database professionals by introducing exemplars such as the 1995 EU Directive on Data Protection and the 1998 UK Data Protection Act.

## Additional reading

# Contents

# 1 An introduction to Database Management Systems

DataBase Management Systems (DBMS's) are a type of middleware that provide a layer of abstraction for dealing with databases. It is nearly always unnecessary to write software from scratch that interfaces with a database, since a lot of database operations will share a significant amount of logic.

Henceforth, a lot of the functionality required of applications that make use of a database is placed into a DBMS, which application developers can make use and save time. The DBMS acts as a service, that is well implemented and is able to enforce good practices and advanced techniques such as concurrency, sharding, recovery management and transactions.

Some advantages of using a DBMS include:

- It decouples data inside a database from the application using it. Either can be re-written at any time so long as they still provide/use the same interface.
- Since the data is decoupled from the application, using a DBMS (in theory) lowers the development cost of the application.
- Most DBMS are scalable, concurrent, fault tolerant, authorisation control (often role based for organisations).

Even though the DBMS aims to provide a layer of abstraction for a user application, there are several layers of abstraction within the DBMS itself. These are:

| | |
|---|---|
| **Physical** | Deals with the file(s) that is written to the storage medium that will hold the database. Needs to know about file formats, indexing, compression, etc. |
| **Logical** | Mainly concerned with mapping the raw data into database 'concepts' such as tables, views etc. It is here that the formal specification of the database is defined, commonly used models include *relational, XML based and document based* |
| **View** | Ensures that only authorised people can view the data. |

If the database is using a relational format, then it will be defined by a schema. A schema dictates how the database is formatted; what tables there are, and what datatypes their columns take. An instance of a database is the content (data) inside of the database at a particular point in time. There is a certain isomorphism between relational databases and imperative programming languages; a schema would be akin to the declaration of variables (i.e. their names and types), while the instance would be their values at a particular point in the program's execution.

Irrespective of what logical model a database uses, most DBMS use between one and three languages to interface with a user/application. These are:

- Data Definition Language - used for specifying schema.
- Data Manipulation Language - used for mutating the data in the database.
- Data Query Language - used to access data in the database.

Often DBMS languages will be both a DML and a DQL, and sometimes a DDL too! One such example is SQL, does all of the above!

# 2 Relational Algebra and SQL

Relational algebra is designed for modelling data stored in relational databases (i.e. tables) and defining queries on it. It can perform unary operations (such as growing, shrinking and selecting from tables), or binary operations (union, intersection, difference, product, join).

## 2.1 Selection $\sigma$

The $\sigma$ operator can select rows that meet a certain criteria from the table.

**Alcohol-Selection**

| Type | Strength | Colour |
|---|---|---|
| Wine | 11 | Red |
| Beer | 4.2 | Yellow |
| Wine | 12.8 | White |
| Port | 18 | Carmine |
| Ale | 11 | Red |

If we run $\textit{Fine-Wines} := \sigma_{Type=Wine}(\textit{Alcohol- Selection})$, we'll end up with:

**Fine-Wines**

| Type | Strength | Colour |
|---|---|---|
| Wine | 11 | Red |
| Wine | 12.8 | White |

## 2.2 Projection $\pi$

The $\pi$ operator can select rows instead of columns. If we do $\textit{Anonymous-Drinks} := \pi_{Strength,Colour}(\textit{Alcohol-Selection})$:

**Anonymous-Drinks**

| Strength | Colour |
|---|---|
| 11 | Red |
| 4.2 | Yellow |
| 12.8 | White |
| 18 | Carmine |

Notice that both the 11% Ale and the 11% Red Wine have the same strength and colour values. Consequently, the projection operator combines those rows into one so the same result isn't displayed twice.

Projection can also be used to do simple arithmetic, $\textit{Test} := \pi_{Strength+Strength->DStrength,Colour}(\textit{Anonymous-Drinks})$:

**Test**

| DStrength | Colour |
|---|---|
| 22 | Red |
| 8.4 | Yellow |
| 25.6 | White |
| 36 | Carmine |

## 2.3 Product $\times$

**Shops**

| Name | Dodginess | Price |
|---|---|---|
| Ali's | High | Medium |
| Tesco | Low | Medium |
| New Zeland Wines | X.High | Low |

We could do a cross product with the Shops and the Alcohol-Selection tables $\textit{Grog-Shops} := \textit{Shops} \times \textit{Alcohol-Selection}$:

**Grog-Shops**

| Name | Dodginess | Price | Type | Strength | Colour |
|---|---|---|---|---|---|
| Ali's | High | Medium | Wine | 11 | Red |
| Ali's | High | Medium | Beer | 4.2 | Yellow |
| Ali's | High | Medium | Wine | 12.8 | White |
| Ali's | High | Medium | Port | 18 | Carmine |
| Ali's | High | Medium | Ale | 11 | Red |
| Tesco | Low | Medium | Wine | 11 | Red |
| Tesco | Low | Medium | Beer | 4.2 | Yellow |
| Tesco | Low | Medium | Wine | 12.8 | White |
| Tesco | Low | Medium | Port | 18 | Carmine |
| Tesco | Low | Medium | Ale | 11 | Red |
| New Zeland Wines | X.High | Low | Wine | 11 | Red |
| New Zeland Wines | X.High | Low | Beer | 4.2 | Yellow |
| New Zeland Wines | X.High | Low | Wine | 12.8 | White |
| New Zeland Wines | X.High | Low | Port | 18 | Carmine |
| New Zeland Wines | X.High | Low | Ale | 11 | Red |

## 2.4 Renaming $\rho$

The notation for renaming columns is pretty simple; $\textit{Drinks} := \rho_{Name,Strength,Hue}(\textit{Alcohol-Selection})$

**Drinks**

| Name | Strength | Hue |
|---|---|---|
| Wine | 11 | Red |
| Beer | 4.2 | Yellow |
| Wine | 12.8 | White |
| Port | 18 | Carmine |
| Ale | 11 | Red |

## 2.5 Join $\bowtie$

If we had:

**People**

| Name | Drinks |
|---|---|
| Alice | Wine |
| Bob | Beer |

We could join it with the Grog-Shops table, using $\textit{Fave-Shops} := \textit{People} \bowtie_{People.Drinks=Grog-Shops.Type} (\textit{Grog-Shops})$

**Fave-Shops**

| Person.Name | Grog-Shops.Name | Dodginess | Price | Type | Strength | Colour |
|---|---|---|---|---|---|---|
| Alice | Ali's | High | Medium | Wine | 11 | Red |
| Bob | Ali's | High | Medium | Beer | 4.2 | Yellow |
| Alice | Ali's | High | Medium | Wine | 12.8 | White |
| Alice | Tesco | Low | Medium | Wine | 11 | Red |
| Bob | Tesco | Low | Medium | Beer | 4.2 | Yellow |
| Alice | Tesco | Low | Medium | Wine | 12.8 | White |
| Alice | New Zeland Wines | X.High | Low | Wine | 11 | Red |
| Bob | New Zeland Wines | X.High | Low | Beer | 4.2 | Yellow |
| Alice | New Zeland Wines | X.High | Low | Wine | 12.8 | White |

If two tables have a column of the same name, then they can be joined naturally without specifying which columns to join explicitly.

### 2.6 Distinct $\delta$

The $\delta$ operator will ensure that no rows are duplicated.

### 2.7 Chaining operators

Just like in normal algebra, you can chain operators:

$\delta(\pi_{Strength,Colour}(\textit{Alcohol-Selection}))$

Gives:

| Strength | Colour |
|---|---|
| 11 | Red |
| 4.2 | Yellow |
| 12.8 | White |
| 18 | Carmine |

## 3 SQL Syntax

### 3.1 Selecting rows

The SQL command to select rows from a table is:

```
SELECT <column-name>
FROM <table-name>;
```

If you only want distinct values from a column, use `DISTINCT`:

```
SELECT DISTINCT <column-name>
FROM <table-name>;
```

If you want all of the columns, use `*`:

```
SELECT *
FROM <table-name>;
```

You can also do derivations:

```
SELECT salary/2
FROM <table-name>;
```

In order to specify which rows you want from the table, or you want to join two tables together, use the `WHERE` clause:

```
SELECT *
WHERE <condition>
FROM <table-name>;
```

```
SELECT *
WHERE table1.columnx = table2.columny
FROM table1, table2;
```

Use `AND` to chain multiple conditions in a `WHERE` clause:

```
SELECT *
WHERE x > 100
AND table1.x = table2.y
FROM <table-name>;
```

#### 3.1.1 Where conditions

You can use the standard $=, <, >$ signs for comparisons between columns, but SQL also lets you use `LIKE`, `BETWEEN` and `IS NULL`:

```
SELECT *
FROM <table-name>
WHERE name LIKE '
```

```
SELECT *
FROM <table-name>
WHERE salary BETWEEN 10000 AND 12000;
```

```
SELECT *
FROM <table-name>
WHERE previous-convictions IS NULL;
```

You can also compare tuples:

```
SELECT *
FROM t1, t2
WHERE (t1.parent, t1.age) = ('Janet',
t2.age);
```

### 3.2 Set operations

The three main set operations are `UNION`, `EXCEPT` and `INTERSECT`, their meaning is rather self evident given their names. An example usage may be:

```
(SELECT *
FROM <table-name>)
UNION ALL
(SELECT *
FROM <table-name>);
```

### 3.3 Joins

We've already looked at the most common join:

```
SELECT *
WHERE table1.columnx = table2.columny
FROM table1, table2;
```

But you can also do it manually:

```
SELECT *
FROM table1 JOIN table2
USING (<column-name>);
```

Or with a `NATURAL JOIN`, which is automatic, but can only be applied when columns have identical names:

```
SELECT *
FROM table1 NATURAL JOIN table2
```

### 3.4 Renaming columns

You can use the `FROM` clause to rename tables:

```
SELECT *
FROM table1 as a, table2 as b
where a.col > b.col;
```

### 3.5 Ordering and other operations

`GROUP BY` can be used to sort rows by a column value:

```
SELECT *
FROM <table-name>
GROUP BY height;
```

Other operators such as `AVG` and `COUNT` can also be used:

```
SELECT AVG(salary)
FROM <table-name>;
```

```
SELECT COUNT(DISTINCT salary)
FROM <table-name>;
```

```
SELECT COUNT(*)
FROM <table-name>;
```

```
SELECT dept-name, AVG(salary)
FROM staff
GROUP BY dept-name
HAVING AVG(salary) > 30000;
```

## 4 Designing databases

Entity-Relationship (ER) Modelling is a simple, high-level conceptual modelling approach that focuses on data requirements rather than business logic. A conceptual model is what we aim to produce in the design phase of database development. It describes the format of the database, they types of entity that are used, the relationships between entities and constraints on the data.

We can translate a conceptual model into a logical schema, which is often just a language such as SQL. We can use the logical schema directly to create and manipulate our database.

### 4.1 Entity Relationship Modelling

There are three basic constructs in ER modelling; entity types, attribute types and relationship types.

Attribute types consist of the following:

Simple or Composite
: A simple type is atomic, whereas a composite type is made of an amalgamation of multiple other types.

Single or Multi valued
: Types can either have single values (such as an integer), or they can have a value from discrete set of allowed values (multi-valued).

Stored or Derived
: Stored attributes are ones where the value of the attribute is directly stored by the database (e.g. licence start date), while Derived attributes are computed from the values of other attributes (e.g. licence end date = start date + 1 year).

Null valued
: Can be null.

Complex-valued
: Made of arbitrarily nested composite or multivalued attributes.

Remember, ER models don't have primary or foreign keys, they only have plain keys. Relationships are modelled explicitly in ER rather than implicitly using keys.

Keys are important in ER modelling. Entities with a key are bound by uniqueness constraints; each key must be unique within the entity. There can be more than one key in an entity type, but there can also be no key, and if this is the case, then the entity is said to be **weak**.

**Carnality ratio** constraints specify the ratio of one entity to another in a relationship. It can be $1:1, 1:n, n:1, n:m$.

**Participation constraints** specify if an entity depends on another entity. If a participation is **total**, then every entity in the total entity must participate in some relationship, if the participation is **partial**, then some entities may not participate in any relationship. For example, student:program is total:partial.

**Weak entities** can only be properly identified if they are related to some other entity type, an **owner**. An **identifying relationship** is one where a weak entity has a total participation constraint with it's owner.

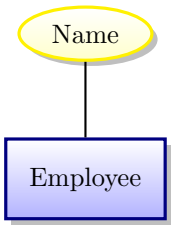
When we're writing a requirements specification, **bold** words are entity types (e.g. **employees, departments and projects**), *italic* words are attribute types (e.g. *name, location, phone number*), and underlined words are relationship types (e.g. employees belong to a department).

### 4.1.1 ER Notation

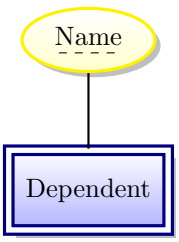
**Entity type**


**Weak entity type**

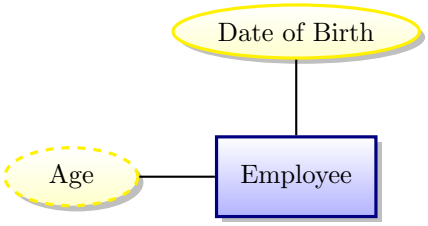

**Attribute**



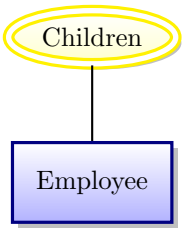

**Weak attribute**



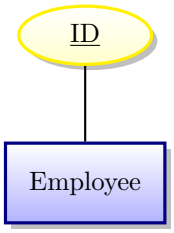

**Derived attribute**



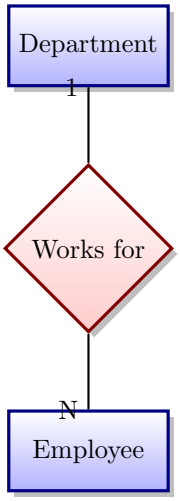

**Multivalue**



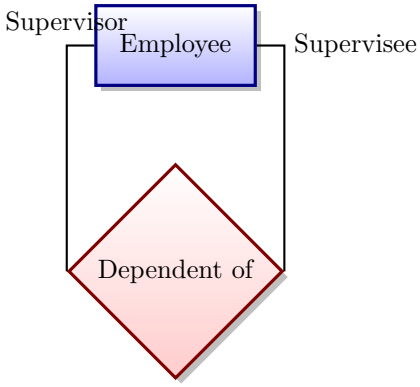

**Key**



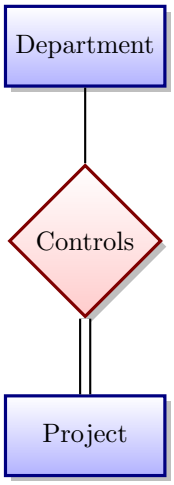

**Relationship (plus carnality ratio)**



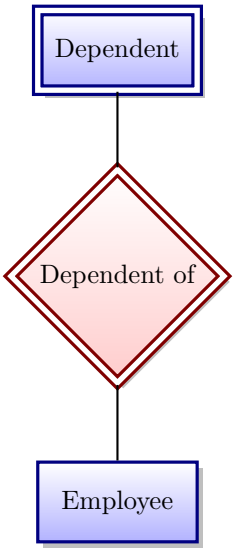

**Recursive relationship**




**Participation constraint**



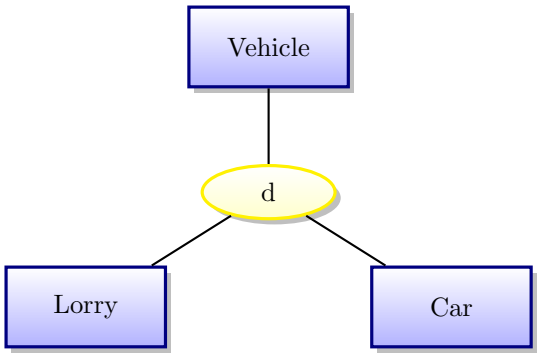

**Identifying relationship**




### 4.1.2 Specialising entities

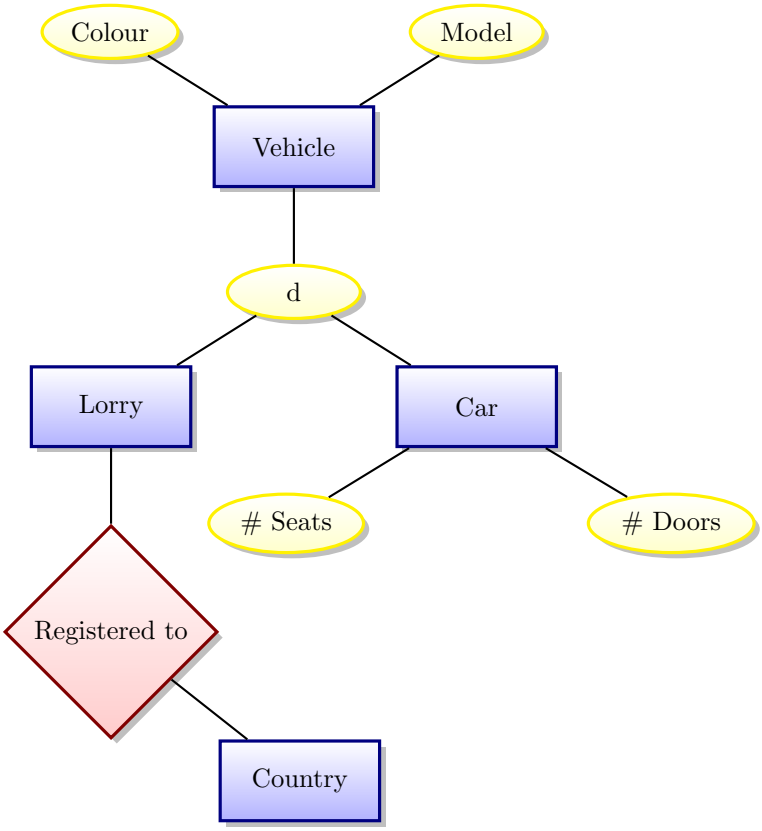
It's often efficient to refine entities into sub-entities if there are fields that are relevant for some members and not others. For example a vehicle entity could have car and lorry sub-entities. One way of doing this is top-down conceptual refinement, which is a fancy way of saying, start with an entity, identify it's subclasses, create sub-entities for them, and then repeat for the newly created sub-entities.

I've not got time to work out how to do ∪ arrowheads now, please draw them on the diagram!




An alternate way to specialise is to do a bottom-up approach. Start with lots of sets, and define a common superset of them. Here, you're generalising rather than specialising.

Each subentity can have it's own attributes and relationships of course:




Entity subtypes can be user, attribute or predicate defined based on whether the user has defined them to be a subtype, their attributes define them to be subtypes or if a predicate has (e.g. if country in EU $\implies$ EU Member).

In the examples, the attribute 'd' is used to create a subtype. This stands for disjoint, since a car can't also be a lorry. There is also 'o', which is used when an entity could belong to multiple subtypes.

# 5 Converting an ER diagram into a relational language

I refer you to Alvaro's sixth lecture, for this part, since it'd take too long to draw all the diagrams you need. If you're reading this and feeling helpful, you write this section and send me a pull request!

# 6 Normalization

Normalizeation is the process of decomposing unsatisfactory (i.e. potentially improvable) relations by creating smaller relations from them. The keys and functional dependencies of the relation determine it's normal form.

3NF and BCNF (Boyce-Codd Normal Form) are the targets we try and work towards.

The normal form indicates a particular quality level of the database schema. 1NF is a relation in its first normal form, where every field contains only atomic values. 2NF, 3NF and BCNF are normal forms that are defined in terms of keys and functional dependencies of a relational schema.

The main technique that we use to try and refine schema is **decomposition**. This is the idea of taking a relation ABCD, and decomposing it into two relations; AB and BCD. Decomposition is guided by the functional dependencies that hold over the relation.

An example would be a table of Employees. If there was a column for office id and another for work address, we would have to update all of the employees who worked at a specific office if the office location moved. If there was a separate table of offices, we could simply update the office location in that table, and because of a relation between Employee and Office, we wouldn't have to update Employee at all!

A key is one or more attributes in a relation where each key is unique for that relation. There may be more than one valid key (candidate key) for each entity, if this is the case, then one of them is selected to be the primary key, and the others are said to be secondary keys.

## 6.1 Functional Dependencies

Functional dependencies are a different way of solving the employee-office problem mentioned a few paragraphs ago. A functional dependency is basically a constraint between two attributes in the entity, the notation would be Office_ID $\rightarrow$ Office_Location, if we were to create an FD between the office ID and it's location.

If an FD is in place, the determinant (the bit before the arrow) determines the bit after the arrow. Determinants of the same value will *always* map onto the same value.

The trouble with functional dependencies is what happens when you want to update the derived value, or insert a new data item that doesn't have a determinant.

# 7 'Advanced SQL'

When it was conceived, SQL was not Turing complete. In order to get encapsulate more logic, programmers would use database API's, or embed SQL commands into another programming language that was Turing complete, where a compiler will resolve them into code in the host language. This was called *Programmatic SQL*, but it was hard to do well, since SQL is a declarative language, and most host languages wouldn't be (they'd be imperative, object oriented or even functional), so the programming styles (and other, more fundamental stuff like type systems) would clash.

Another solution is to extend SQL to make it Turing complete. Oracle did this to create Procedural Language/SQL, which is what we have the privilege of using on the course.

## 7.1 Procedural Language/SQL

### 7.1.1 Anonymous Block

This is an unnamed block of PL/SQL. This one will print out the number of students in the student table, and log any errors if it can't.

```
DECLARE
  amount NUMBER := 0;
  error_code NUMBER;
  error_message VARCHAR2(255);

BEGIN
  SELECT COUNT(*) INTO amount
  FROM STUDENT;

  DBMS_OUTPUT.PUT_LINE(amount);

EXCEPTION
  error_code := SQLCODE
  error_message = SQLERRM
```

```
    INSERT INTO errors VALUES(error_code, error_message);

  END;
```

### 7.1.2 Cursors

If we run an SQL command that could return an arbitrary number of rows, we need to use a cursor. This allows us to access one row at a time and iterate over them. It must be declared and opened before use, and closed after it is finished with. Cursors can be used to update content in tables as well as just viewing them.

Anybody familiar with Android development will probably have come across cursors.

```
CURSOR getStudentsDoingCourse(myCourseName Student.courseName%TYPE) IS
  SELECT *
  FROM Students
  WHERE Students.courseName = myCourseName
  ORDER BY Student.id;
```

### 7.1.3 Subprograms; Procedures and Functions

Subprograms are named blocks that can be called with parameters. They aid in abstraction and increase the reusability and maintainability of the code.

Functions always return a single value, whereas procedures don't return any value. In order to return a value, functions use the `RETURN` keyword.

Here is an example procedure:

```
CREATE OR REPLACE PROCEDURE getAverageGrade(sID IN Students.ID%TYPE) AS
  var_name Students.name%TYPE;
  var_grade NUMBER;
  var_count NUMBER;
  var_temp NUMBER;

  CURSOR getStudentGrades(sId StudentGrades.ID%TYPE) IS
    SELECT *
    FROM StudentGrades
    WHERE StudentGrades.ID = sID;

  CURSOR getStudentName(sId Students.ID%TYPE) IS
    SELECT name
    FROM Students
    WHERE Students.ID = sID;

  var_error_code NUMBER;
  var_error_message VARCHAR(255);

  BEGIN

    OPEN getStudentGrades(sID);

    LOOP
      FETCH getStudentGrades INTO var_temp;
      EXIT WHEN getStudentGrades%NOTFOUND;

      var_grade := var_grade + var_temp;
      var_count := var_count + 1;
    END LOOP;
```

```
    IF getStudentGrades%ISOPEN
      THEN CLOSE getStudentGrades;
    END IF;

    var_grade := var_grade / var_count;

    OPEN getStudentName(sID);
    FETCH getStudentName INTO var_name;

    DBMS_OUTPUT.PUT_LINE(var_name || ': ' || var_grade || '%');

    IF getStudentName%ISOPEN
      THEN CLOSE getStudentName;
    END IF;

  EXCEPTION
    WHEN OTHERS
    THEN
      var_error_code := SQLCODE;
      var_error_message := SUBSTR(SQLERRM, 1, 255);
      DBMS_OUTPUT.PUT_LINE(var_error_code || ': ' || var_error_message);

      IF getStudentName%ISOPEN
        THEN CLOSE getStudentName;
      END IF;

      IF getStudentGrades%ISOPEN
        THEN CLOSE getStudentGrades;
      END IF;
  END;
```

To execute a procedure, do `EXECUTE getAverageGrade(8955942)`.

### 7.1.4 Triggers

Triggers are constructs that react to certain conditions. They obey an **event-condition-action (ECA)** model.

Triggers can be fired by insertions, deletions or updating rows in a table when tables are created, altered or dropped, as well as when error messages are generated. Triggers can fire before, after or instead of the triggering event, and the `action` is a block that is executed if the trigger fires.

Row level triggers execute the action that for each row that has changed in a table, while statement level triggers only execute once per fire event. Triggers can be made to cascade, if the effects of one cause another to fire. It is possible that this could result in non-termination.

```
CREATE TRIGGER name
  (BEFORE | AFTER | INSTEAD OF)
  (INSERT | DELETE | UPDATE [OF columns])
ON tableName
  // (refer to the old/new values of an update)
  [REFERENCING (OLD | NEW) AS (oldName | newName)]
  // Row-level or Statement-level?
  [FOR EACH (ROW | STATEMENT)]
  [WHEN condition]
  triggerAction (e.g. call procedure)
;
```

# 8 Transactions, concurrency and recovery

These are all very important features of a DBMS system, and are some of the greatest advantages of using a DBMS instead of rolling your own system as a programmer.

## 8.1 Transactions

A transaction is a action or series of actions carried out by an application on the database. It can be seen as a logical unit of work on the database, and one level of abstraction sees client applications as just a series of transactions with intervals of non-database work in between.

Each transaction moves the database from one state to another (or possible the same state). Each state is consistent, however, in between states, the database may be inconsistent (i.e. while processing occurs on the database, constraints may be violated).

Transactions can have two outcomes, success and failure. If the transaction is successful, then it will be committed, and the database will enter a new state. If the transaction fails, then it will be aborted and the database is rolled back to the state it was in before the start of the transaction. Once a transaction is committed, it cannot be aborted, but an aborted transaction can always be restarted later for another attempt.

Transactions have four basic properties, called ACID:

**Atomicity**
: A transaction is either fully successful, or it's failed; there's no in between. This is sometimes called an 'all or nothing' contract.

**Consistency**
: The transaction must transform the database from one consistent state to another.

**Isolation**
: Partial effects of incomplete transactions are not visible to other transactions. All transactions are in effect, sandboxed.

**Durability**
: The effects of a committed transaction are permanent. If a later transaction fails, then it will not affect previous transactions.

## 8.2 Concurrency

Concurrency control is important since it allows multiple operations on the database to take place simultaneously without interfering with each other.

Without concurrency control, if two transactions were to execute in parallel, they could affect each other and produce incorrect results. Problems can include:

**Lost update problem**
: If two users update the database at once, the one that finishes first will be overridden by the one that finished second. This is mitigated by stopping concurrent operations from reading objects/tables while another operation is writing to it.

**Uncommitted dependency problem**
: If one operation reads a piece of data while another operation writes to it, then the first could read data that is in an intermediate state which is inconsistent. To prevent this from occurring, then we must stop reads from occurring on objects that haven't been committed.

**Inconsistent analysis problem**
: If one transaction reads several values, but they are updated by another transaction part way through the read, then some of them will be inconsistent with the others. To avoid this, again stop one transaction from reading while another is updating and vice versa.

### 8.2.1 Serializability

We want to be able to schedule transactions so that they don't interfere with each other. The easy way of doing this, is to have no concurrency, but this would severely limit performance. Serializability guarantees sequences of execution that will ensure consistency.

Lets define some terminology; a *schedule* is a set of of reads and writes to the database by concurrent transactions, a *serial schedule* is when the reads and writes must be executed consecutively with no interleaved operations, and a *non-serial schedule* is one where any operations can be interleaved.

To make the database as efficient as possible, we want to find as many non- serial schedules as we can so that we can get parallel computation. These are called serializable schedules.

The exact order of the reads and writes in the database is important; the transactions need to execute in a serial manner if one transaction is writing while another is reading or writing to the same data item. We can use precedence graphs to test for these conflicts.

In order to create a **precedence graph**, you create a node for each transaction and then create a directed edge from $T_i$ to $T_j$ if:

- $T_j$ reads the value of an item written by $T_i$
- $T_j$ writes a value into an item after it has been read by $T_i$
- $T_j$ writes a value into an item after it has been written by $T_i$.

When you've drawn a precedence graph, you must run a cycle finding algorithm on it (such as Floyd's cycle finding algorithm). If a cycle is present, then the transactions must run in a serial manner.

### 8.2.2 Concurrency control techniques

The two most basic control techniques are locking and timestamping. Both techniques are pessimistic, which means that they delay a transaction in case it conflicts with another transaction.

Locking is when data items are locked to prevent other transactions from viewing or updating them (depending on the type of lock obtained). Transactions must obtain a shared lock on a data item when it wants to read, and an exclusive lock when it wants to write.

In order to ensure that assigning locks doesn't violate isolation and atomicity rules for the database, locks are assigned (usually) using the two-phase-locking (2PL) protocol.

The **Two Phase Locking** protocol, as you might expect, has two phases; the growing phase and the shrinking phase. In the growing phase, the transaction may acquire locks but cannot release them, and in the shrinking phase, it can release locks but cannot acquire them. If all transactions follow 2PL, then the schedule is serializable.

If transactions depend on each other (due to locks being released before a transaction ends), then **cascading rollback** can occur. If T3 depends on T2, which depends on T1 and T1 is rolled back, then both T3 and T2 must also be rolled back. This can be stopped by ensuring that locks are only released at the end of transactions (since they have then been committed), which is called rigorous 2PL.

Deadlock can occur if two transactions are waiting from a lock from each other. If this happens, then one or more of the transactions should be aborted and restarted. Deadlock can be detected using timeouts (only waiting $n$ seconds to acquire a lock).

### Recovery

You can keep a transaction log of all actions executed on a database. This log is stored in a file on (often) a backup system. If the database becomes corrupted, you can simply replay the

transactions in the log and get the database back to the state it was in.

Shadow paging is another recovery technique, where a copy of the page a transaction is taking place on is created. If the transaction fails, you can simply replace the page with the copy and it will be the same as before.

# 9 File organisation and indexing

Most (certainly the ones in this course) databases store their data on a secondary data store, such as magnetic disk. Data is organised onto files on the disk, and each file is composed of multiple records, each with one or more fields. Typically, a field corresponds to an attribute, a record to an entity, and a file to an entity type.

When data is needed (to be modified or read), the database maps the logical location of the data in the database onto a physical location on the disk.

The physical record is the unit of transfer bewteen main memory (RAM) and the secondary memory. When the DMBS wants to read or modify a part of the stored database, then the nececarry page is loaded into primary storage. Each 'unit' of physical database is usually referred to as a *page* or, less commonly block.

Files are oragnised in three different ways:

**Heap files**
When the records stored in the file are in no particular order. New records are inserted at the end of the file, which is efficient, but means that finding them again later requires an $O(n)$ linear search (which is only really kind of efficient if you're selecting most of the records in the file, such as when doing `SELECT * FROM <tablename>`). Deleting records requires more time than finding them, since the file must be written back to disk. Perodic reorganisation (defragmenting) is required to compact the file and fill in the blank spaces that were left by deletion.

Good for when you're doing bulk loading of data, a relation requires only a few pages, indexes are used, and every record is almost always accessed.

**Sequential files**

When the files are ordered on a given field. This field also serves as a key, called the ordering key. As you might expect, this vastly increases the speed of finding a record, since we can use binary search. However, it also increases the complexity of insertion; because elements must be kept in order, if there's no free space at a location of the insert we must shift all of the other records over (and potentially cascading this shift to other pages).

Q: Where would you insert a record in a page if you wanted to maximise the chance of cascading? A: Near the start.

One way to minimise the effect of cascading is to have an unordered 'overflow' pool at the end of the file. Periodically, the DBMS will merge in this pool with the rest of the file. This reduces the effectiveness of the search, since we'll need to do a linear search of this section if we don't find whatever we're looking for with a binary search of ordered bit.

**Hashed files**

A hash function determines at what logical address records reside at in the file.

Each file is organised into $M$ equal sized buckets (read as partitions). One of the fields in the file is designated as the hash key, and then records are assigned to buckets based on the hash of this key. This means that search is very efficient on this key.

One problem with hashed files is that when a bucket is full, where do we put more records? This is the same problem that datastructures such as HashMaps face, and it is mitigated in similar ways too:

*Open addressing*
: This is when if a bucket is full, a linear search for the next blank space will take place, and it is here that the item will be inserted.

*Unchained overflow*
: Here, a seperate area is designated for collision records. This is also organised in terms of buckets.

*Chained overflow*
: If you're a Java whizz, you might have met this before; overflow items are kept in a linked list structure, where each item points to the next. This means searching is a linear operation, but lets you cut down the amount of records you'd need to search otherwise. With this method, every record must have an additional field to hold a pointer to the next record.

*Multiple hashing*
: This is when a second field is used as a secondary hash key if the first produces a collision, aiming to produce a new address for the record. In general, the new hash function is used to place colliding records in the overflow area.

These techniques can be combined in order to provide a comprehensive solution (for example, having multiple hashing, then a chained overflow when the second hash function fails).

When updating a record in a hashed file, you've got to relocate the record if the hash key changes. This can make updating slow.

Using a hash is good when what you're searching for will exactly match the hash key, but not good when you're pattern matching, getting a range of values, updating frequently, or the search parameters aren't exactly on the hash key (i.e. other fields or part of the hash key).

The steps taken by the DBMS to manipulate the pages is called the *access method*.

## 9.1 Indexing

An index is a datastructure that allows the database to find records more quickly and thus improve query response time. It is similar to the index in a book, and is usually ordered on one key (the search key).

An index can be either dense or sparse; the former has a value for every search key (so every record), while the latter has one for only some keys.

There are three types of index; primary, clustering and secondary. Each file can have one primary or one clustering index, but many secondary indices. The different types of indices are listed here:

**Single level indices**

*Primary indices*
: These indices are defined on a single file, that contains records ordered by a key. The primary index is *sparse*, containing an entry for each disk block that can then be searched sequentially.

*Clustering indices*
: Here, the keys are ordered like in primary indexing, however, the index contains each distinct key value, and points to the start of that key in the list (since theye'll probably be lots of repeated keys). This is another *sparse index*.

*Secondary indices*
: Secondary indices are *dense* because they use keys that the file is not ordered on. This means that they must point to every record individually. They speed up indexing

since they themselves are ordered, so you can do a binary search on them. They are used to complement a primary/clustering index, and there can be more than one of them per file.

**Balanced trees**

If a tree has the same depth from root to leaf for all it's branches, then it's called a balenced tree. The order (also known as the degree) of the tree is the maximum number of branches per parent node, with larger orders creating shallower trees (as would be expected). The access time is dependent on the breadth of the tree, since it's $O(n \log_o n)$, with $o$ being the order of the tree.

B+-Trees are similar, except their leaves form a linked list. This means they can be traversed in $O(\log_b n)$ time, where $b$ is the order of the tree.

B+-Trees are very versatile, since they support pattern matching well, as well as getting a range of or part of a key. Since B+-Trees are dynamic (their structure changes as they are used), their performance is kept optimal, and they also allow for efficient retrieval in the order of the access key.