



Thank you for taking the time to have a look at this free sample – which demonstrates just a very small part of the content of our popular **Database Glossary**.

This fantastic **120 page** glossary from Knowledge River covers the A-Z of the whole database world (literally) and contains **600 entries** (many with *extended practical examples*) covering:

- Relational theory (relations, domains, tuples, attributes, cardinality, degree etc)
- Relational algebra (SELECT, PROJECT, JOIN, CARTESIAN PRODUCT etc)
- Data management (data security, privileges, access control, data consistency etc)
- Database development life-cycles (requirements capture, data analysis etc)
- Data modelling (entity-relationship, relational and object-relational data models)
- Non-relational data models (navigational and hierarchical databases)
- Database Management Systems (DBMS, their functions and their architectures)
- Normalization and all its stages (functional dependencies, normal forms etc)
- Database optimization (indexes, clusters, cost-based and rule-based optimizers)
- SQL and all its keywords and constructs (static, dynamic and embedded SQL)
- Data warehousing (architectures, functions and processes)
- Distributed and replicated database systems (fragmentation, two-phase commits)
- XML (what it is and how it is imported and exported into the database world)
- Java (SQLJ, embedded SQL, application development, objects, methods etc)

This list is far from exhaustive. There are **600 technical terms** clearly described and supported by even clearer examples. So look up the technical term, read the detailed description and then digest the practical examples – you won't be puzzled any longer!

If you need further advice please email us at: feedback@knowledge-river.co.uk

Thank you once again for looking at this learning resource.



Copyright © Knowledge River Ltd 2008

All rights reserved.

Please turn over to view some sample content...

Abstract Data Type

Also known as an **ADT**. A data object or **data structure** that is defined by the *operations* that can be performed upon it (*what* it does) rather than any particular implementation used to create it (*how* it does them). For example, in computer science there are abstract data types (structures) called **stacks, lists, queues** and **trees**.

Each has a fixed set of operations but the way that these structures and operations are programmed (implemented) can vary and is not part of the specification of the **ADT**. ADTs are similar to objects in **object-oriented** (OO) programming and **databases** because they each have an interface to the outside world (a set of operations) and an internal (hidden) implementation.

Aggregate Function

One of a series of **SQL functions** that takes as its input a *set* of numerical data values (from a table column) and returns a *single* numerical value. The five commonest ones are:

- **AVG** (the arithmetic average of the data set)
- **COUNT** (the number of separate values in the set)
- **MAX** (the maximum or largest value in the data set)
- **MIN** (the minimum or smallest value in the data set)
- **SUM** (the summation of all the values in the set)

Binary Operator

Any **operator** that takes (as its input) two **operands** to produce a result. Examples include logical operators (AND, OR), the four arithmetic operators as well as database operators (**UNION, INTERSECTION, MINUS, JOIN, DIVIDE, and CARTESIAN PRODUCT**). For example:

```
SELECT name FROM staff  
UNION  
SELECT name FROM student
```

```
SELECT name FROM staff  
INTERSECT  
SELECT name FROM student
```

Compare to **unary operators**.

Compound Query

A single logical **query** comprising two or more distinct search elements. Also known as a composite query. For example:

```
SELECT *  
FROM Student  
WHERE age > 25                (element 1)  
AND subject = 'Computing'    (element 2)  
AND name LIKE '%Smith%';    (element 3)
```

Compare to **sub-queries** (or **sub-selects**)

Foreign Key

Used to represent the **relationships** from the original **entity relationship diagram**. Each relationship on the **ERD** will have a foreign key implementation whereby a **candidate key** (normally the **primary key**) of one relation is transplanted into the associated relation as a **foreign key**. This acts as logical linkage between the two relations – just like the relationship on the ERD links the two associated **entity types**.

For example, if there is a relationship X between entity types A and B on the ERD then when we move to the **relational model**, by placing the primary key of relation A into relation B (the precise placement of the foreign key is variable) it becomes a foreign key in B and retains the original logical connection between A and B (as originally specified by relationship X).

The relation which accepts the transplanted key is the **referencing relation** and the relation that donates the key is the **referenced relation**. Both primary and foreign keys can be single **attributes** or **sets** of attributes. The values of the foreign key must either be **NULL** or match those of the transplanted candidate (normally primary) key. If they do not, then the foreign key violates **referential integrity**.

GROUP BY

An **SQL** clause that is used to group or collect data into categories based on one or more columns (the **grouping columns**) and often used with **aggregate functions**. Often used with the **HAVING** clause too. See the example below:

```
SELECT Department, AVG(Grade), MIN(Grade), MAX(Grade)  
FROM University  
GROUP BY Department;
```

Inner Join

A relational operation performed on two or more **relations** or **tables** - but see **recursive (self) joins** - whereby the **rows** from those different tables are compared (normally for equality) on the basis of one or more named **columns** defined over a shared **domain** (which will be common to both relations or tables).

Compare to **natural joins**, **outer joins** and **unqualified joins (Cartesian product)**.

Nested Queries

Also known as a **sub-query**, this is a two (or more) part query consisting of an **inner query** (or **sub-select**) and an **outer query** (or **main query**).

For example:

```
SELECT *  
FROM Company  
WHERE ID IN  
  (SELECT Company ID  
   FROM Supplier  
   WHERE Product Code In ('123', '234', 345'));
```

The inner query is executed first before passing back any **intermediate results** to the outer query for final processing.

Partial Dependency

A problem within a **relation** whereby one or more **attributes** *outside* of the **primary key** are determined by only *part* of the primary key. For example, if the primary key of 'Treatment' is:

(Vet ID, Pet ID, Date, Vet Name, Pet Name, Type, Notes)

And we have the following dependencies:

FD: Vet ID \Rightarrow Vet Name

FD: Pet ID \Rightarrow Pet Name

These are classed as **partial** dependencies and are **bad**. They need to be removed by decomposing 'Treatment' down into three child relations (one for vets, one for pets and one for the actual treatment data – Type and Notes).

The attributes 'Type' and 'Notes' depend on the *whole* key (they are **fully functionally dependent** on the whole key) and are fine. A relation which has no partial dependencies is in **second normal form (2NF)**.

Role

A collection of individual **privileges** that are packaged together under a unique name and then allocated to individual **users**. This means different users with the same security needs can be given the same set of privileges easily and efficiently. For example:

```
CREATE ROLE demo;  
GRANT SELECT ON Table1 To demo;  
GRANT INSERT ON Table1 To demo;  
GRANT UPDATE ON Table1 To demo;  
GRANT DELETE ON Table1 To demo;  
GRANT demo to User1;  
GRANT demo to User2;
```

Synonym

Also known as an **alias** in some **DBMS**, this is a technique for simplifying access to **tables** and **columns** in *other* user's **schemas**. For example, if Tutor1 needs to access the column called Grade in the table called Student which lives in the schema of Tutor2 then Tutor1 must enter:

```
SELECT Tutor2.Student.Grade  
FROM Tutor2.Student;
```

This **qualifying** of the table and column names is very long-winded and prone to errors so by declaring a synonym:

```
CREATE SYNONYM Target  
FOR Tutor2.Student;
```

Tutor1 can now simply enter:

```
SELECT Target.Grade  
FROM Target;
```

And get the same results. Much cleaner and easier. All the above assumes Tutor1 has the appropriate **privileges** on the Tutor2 table.