SQL Basics

An Overview of the Language of RDBMS

By Courts Carter

Structured Query Language (always written as SQL and pronounced "sequel") is the de facto standard for interacting with all Relational Database Management Systems (RDBMS). The rich set of instructions in SQL provides the power to query, add, modify, and delete data in an amazingly flexible manner, regardless of RDBMS vendor. In part its power is on offshoot of its simplicity. A very small language, SQL is declarative instead of procedural. This means you use SQL to tell the database only what you want done without prescribing how the DB should go about doing it.

This separation of function from implementation coupled with relational set theory, the underpinning concept of RDBMS, means you can accomplish a great deal in just a few terse commands.

DML, DCL, DDL, DUH-HUH?

SQL is a specification and doesn't belong to any single company – it's a standard developed by a committee whose body members represent a wide range of disciplines. They merely publish the SQL standard – it's up to software vendors to develop commercially viable products which comply with this spec. In general, few companies have met the standard 100%, but this is not necessarily bad. The committee's recommendations have, at times, lagged behind the innovations released by RDBMS manufacturers; largely because a spec can remain silent on contentious issues whereas users do not. Thus faced with a user community demanding ever richer features the vendors forged ahead, often in different directions, but always leaving the SQL committee in the dust. As a result today there are many subtle variations and flavors of SQL, but all share the same core functionality.

Speaking of basic functionality… you've probably heard "**ODBC**" bantered about. It stands for Open Database Connectivity and its only purpose is to be the go between for applications communicating with database engines. That is, an ODBC "driver" is a piece of software that understands the base set of standard SQL calls and can translate these to the exact calls implemented in Sybase, Oracle, Microsoft Access, or whatever databases you are using. ODBC was a standard put forth by Microsoft to simplify connecting all these databases which speak with accents.

While ODBC sounds (and is often) handy, it also embodies only the most basic set of SQL functionality and like we said, the standard is way behind on addressing many pressing business needs. Therefore, ODBC is less than ideal. Ideal is using a native driver – a custom piece of software "middleware" supplied by all database vendors along with their database server engines. Native drivers are smart and fully optimized to run with the vendor's particular flavor (extension) of SQL.

So now you probably want more acronyms (this is computer-speak so we must proliferate acronyms). Here ya go: SQL commands all fit within one of three functional categories DCL, DDL, or DML.

First, **DCL** is the Data Control Language. It provides a means for the granting access privileges to users and groups of users; it's the security layer built into an RDBMS. Next, **DDL**, Data Definition Language, is the set of commands used to create and modify the structure of a database. It is used to create tables and other structures found in the DB. Lastly is **DML**, Data Manipulation Language. This is really the only aspect you probably need to learn. It's the one that allows you to manipulate data. This article/presentation is all about the data manipulation commands available to us in SQL's DML. This is easy because there are only four. The are:

`SELECT`: analogous to read. It is used to query (ask questions of) the database. Ex:

"Give me everyone who is currently an employee and has green eyes or red hair".

INSERT: add data to the DB.

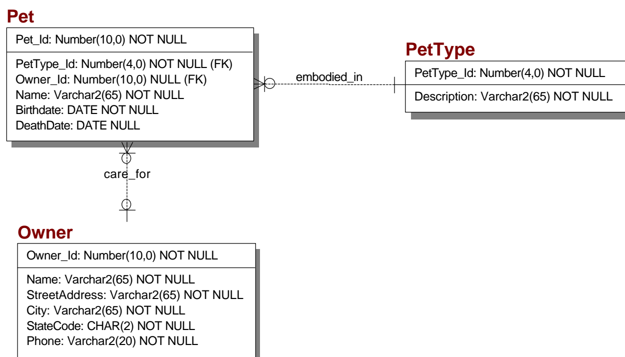
UPDATE: change existing data in the DB.

DELETE: remove data from the DB.

We shall go over each of these in some detail, but before doing this let's decide what the database we are using looks like.

Getting Started

What is the one topic everyone or nearly everyone can relate to? Pets! Puppies, kittens, and Vietnamese potbelly pigs. So let's use a pet database as our example. If you need to role-play pretend that you are the owner of a exclusive Beverly Hills kennel. I'm going to jump directly in with a model which has already been more-or-less normalized (if you are not sure what that means consider yourself lucky, you've obviously escaped our talk on normalization! Also, the DDL for creating these three tables is included at the end of this article and on the MIS website).



The Data Model

The data model shown here tells us that each Owner may "care for" one or more Pets. And each Pet must be of exactly one PetType. This second statement just means a Pet cannot be "cat-dog" (part cat and part dog). Both of these relationships are examples of **one-to-many** relationships.

✧ *What are the Primary Keys for these tables?*

✧ *What are the Foreign Keys?*

✧ *Which columns are required? How were you able to tell?*

The Data Model is amazing in its ability to lock you into one way of doing or even thinking about things. It's nigh on impossible to do anything which the model does not support. Take a moment to consider what this model tells you about the Business Rules. You should be able to answer the following:

✧ *How many Pets may a given Owner have?*

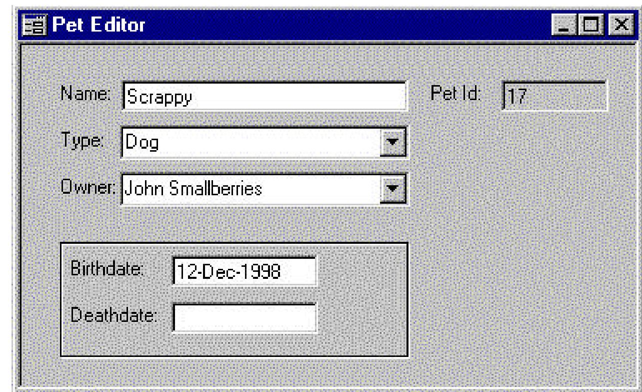
✧ *How many Owners may a Pet have?*

✧ *Can you have a Pet that is part dog and part cat?*

✧ *How many addresses and phone numbers may an Owner have?*

✧ *Will your kennel database be able to tell you whom the original owner of a Pet was if the Pet is sold, such as the case of a racehorse? Is this important to you?*

Are these assumptions accurate to your business needs?



Often the underlying Data Model reveals itself in the GUI front-end For example, an editor for Pet would probably look something like the one shown here. Are the controls what you expected? Can you see how they relate to the underlying data structures?

Inserts

To get going let's insert some data into these tables:

```
INSERT INTO Pet (Pet_Id, Name, Birthdate)
  VALUES (1, 'Gizmo', '03-Apr-93');
```

If we try this statement we get this error:

```
ORA-01400: cannot insert NULL into ()
```

The database is complaining because we failed to supply some required data. We look at the model and realize that we need to supply the Pet Type. So we try this:

```
INSERT INTO Pet (Pet_Id, Name, Birthdate,
PetType_Id )
  VALUES (1, 'Gizmo', '03-Apr-93', 1 );
```

This time we get an "integrity constraint" error. Recall that integrity constraints are rules the database uses to police the quality of its data. In this case the rule is a referential integrity constraint which essentially says that you cannot tell the database to refer to something which doesn't exist. Duh!

In looking at what might have gone wrong we realize that not only is PetType required, but the PetType_Id column appearing in the Pet table (a foreign key) refers to the Primary Key column on the PetType table (we can again see this in the data model). So before we can insert a Pet we need to populate the PetType table. Here it goes:

```
INSERT INTO PetType (PetType_Id, Description)
VALUES (1, 'Dog');

INSERT INTO PetType (PetType_Id, Description)
VALUES (2, 'Cat');
```

```
INSERT INTO PetType (PetType_Id, Description)
VALUES ( 3, 'Mouse');

INSERT INTO PetType (PetType_Id, Description)
VALUES ( 4, 'Monkey');

INSERT INTO PetType (PetType_Id, Description)
VALUES ( 5, 'Bird');

INSERT INTO PetType (PetType_Id, Description)
VALUES ( 6, 'Fish');

COMMIT;
```

✧ *What would happen if we left off the Commit?*

Now let's try adding Gizmo again:

```
INSERT INTO Pet (Pet_Id, Name, Birthdate,
PetType_Id )
VALUES (1, 'Gizmo', '03-Apr-98', 1);
```

This time we got no error. But we need to Commit the insert!

```
Commit ;
```

Why didn't we need to specify an Owner like we did the PetType? Great question! From the model we note that the relationship from Owner to Pet does allow Nulls. Why" yours is a benevolent kennel that accepts strays, so the Owner of a particular pet may not be known to us.

Nulls

A Null is a very special database construct. It means "information not given". Maybe an example is the best way to illustrate the concept of Nulls.

Let's say we have OwnerAge in the Owner table. We, being really diligent database designers, attach a constraint to this column which only allows ages greater than 0 and less than 150 (we checked Guiness to make sure we covered the extreme cases). Now what do we store in OwnerAge We don't know a person's age? Answer: we allow nulls.

What this means is that we do not have the person's age. We **cannot** make any claim as to why we do not have the age. It may be that the person refused to tell us or that we forgot to ask.

Nulls are physically stored in a separate column than the data they are attached to i.e. there is no magic number store in Age that means "unknown". As programmers we might have said that age must be between 0 and 150, or -1 if we do not know it. But consider what would have happened if we had asked the database for all Owners under the age of 20. The people with -1 in the age column would have been included and this is wrong because we just don't know this information. The actual answer would have excluded the unknown agers.

Nulls are really powerful and often misinterpreted – make sure you are never caught reading too much into missing data!

Let's continue adding data:

```
INSERT INTO Owner (Owner_Id, Name,
StreetAddress, City, StateCode, Phone)
VALUES (1, 'Bob Barker', '1234 Main Street',
'San Mateo', 'CA', ' 214 555 3424');

INSERT INTO Owner (Owner_Id, Name,
StreetAddress, City, StateCode, Phone)
VALUES (1, 'Fred Kuba', '542 3rd St.', 'San
Rafael', 'CA', '425 555 3822');
```

The first Insert statement worked fine, but the second line receives an error ("`ORA-00001: unique constraint violated`"). Why? We violated the unique constraint on the Primary Key for the Owner table. We tried to insert two rows with the same value. 1, for Owner_id Have we messed up the database'? No. We can rollback the entire transaction.

```
ROLLBACK ;
```

✧ *How many statements got rolled back?*

The unique constraint on the Primary Key protects the database in a way akin to assign to a credit card company ensuring that the same account number is not given to two different customers.

Lets fix the prior statements and commit them to the DB along with some more Pets (note that here the Owner_Id IS Included):

```
INSERT INTO Owner (Owner_Id, Name,
StreetAddress, City, StateCode, Phone)
VALUES ( 1, 'Bob Barker', '1234 Main Street',
'San Mateo', 'CA','415 555 3424');

INSERT INTO owner (Owner_Id, Name,
StreetAddress, City, StateCode, Phone)
VALUES ( 2, 'Fred Kuba', '542 3rd St.', 'San
Rafael', 'CA', '425 555 0822');

COMMIT;

INSERT INTO Pet (Pet_Id, Name, Birthdate,
PetType_Id, Owner_Id)
VALUES (2, 'Scrappy', '03-Apr-93', 1, 1);

INSERT INTO Pet (Pet_Id, Name, Birthdate,
PetType_Id, Owner_Id)
VALUES (3, 'Rufus', '28-May-90', 1, 1);

INSERT INTO Pet (Pet_id, Name, Birthdate,
PetType_Id, Owner_Id)
VALUES (4, 'Spot', '01-dec-95', 1, 2 );

INSERT INTO Pet (Pet_Id, Name, Birthdate,
PetType_Id, Owner_Id)
VALUES (5, 'Spot', '05-Oct-92', 1, 1 );

INSERT INTO Pet (Pet_Id, Name, Birthdate,
PetType_Id, Owner_Id)
VALUES (6, 'Candy', '05-Oct-88', 2, 1 );

COMMIT;
```

Insert owner data until we have the following loaded into the database.

Owners					
Owner_ Id	Name	Street Address	City	State Code	Phone
1	Bob Barker	1234 Main Street	San Mateo	CA	415 555 3424
2	Fred Kuba	542 3rd St.	San Rafael	CA	415 555 0822
3	Joan Small	88 Pine St Apt A	Los Angeles	CA	213 555 9004
4	Courtney Ay	193 43rd Avenue	Miami	FL	404 555 2001
5	Bird Parker	62 156th St	NY	NY	201 555 1892

Pets					
Pet_ Id	Name	Owner_ Id	Pet-Type_ Id	Birthdate	Death date
1	Gizmo	Null	1	03-Apr-98	Null
2	Scrappy	1	1	03-Apr-93	Null
3	Rufus	1	1	28-May-90	Null
4	Spot	2	1	01-dec-95	Null
5	Spot	1	1	05-Oct-92	Null
6	Candy	1	2	05-Oct-88	Null

We've now used the Insert statement is to insert new rows into some tables. Format syntax of the insert statement:

```
Insert Into tablename [(columnlist)]

{Values (valuelist) | selectstatement }
```

Here are a few other samples of the Insert statement:

1. Insert a new record into the Students table using values supplied:

```
Insert Into Students
VALUES ('12672656', 'Johnson Mary', '1313
Mockingbird Lane');
```

2. Insert records into the StudentHistory table taking records from the Students table:

```
Insert Into StudentHistory
  Select *
    From Students
    Where  ActiveRegistration = 'N';
```

Selects

Now that we have all this data in these tables let's ask the database some questions

1. What are all of the pet names in the table'!
2. What are all of the d o g names? Cats'?
3. Which pets are named 'Spot"' Which pets have names beginning with 'S'?
4. Which Pets belong to Bob Barker?

The Select statement is most commonly the one of interest to the vast majority of users - they want reports. With this statement you can do selections, projections, and joins. Selection means getting a subset of the rows in a table (get all of the data for all dogs in the table). Projection is selecting a just the columns of data which you are interested in (what are all of the pet names?). Join is the sexiest. It is the ability to connect multiple tables via some common attribute (what are all of the names of dog owners. i.e. get all of the dogs from the Pet table and join this with the Owner table).

What are all of the pet names in the table?

```
SELECT Name
  FROM Pet;

Name
---------------------
Gizmo
scrappy
Rufus
Spot
Spot
Candy
6 rows selected.
```

This is nice, but let's clean it up by asking that these be returned in ascending alphabetical order.

```
SELECT Name
  FROM Pet
  ORDER BY Name ASC;

Name
---------------------
Candy
Gizmo
Rufus
Scrappy
Spot
Spot
6 rows selected.
```

What are all of the dog names? Cats?

```
SELECT Name
  FROM Pet
  WHERE PetType_Id= 1
  ORDER BY Name ASC;

Name
---------------------
Gizmo
Rufus
Scrappy
Spot
Spot
5 rows selected.

SELECT Name
  FROM Pet
  WHERE PetType_Id = 2
  ORDER BY Name ASC;
Name
---------------------
Candy
```

```
1 row selected.
```

Mice?

```
SELECT Name
  FROM Pet
  WHERE PetType_Id = 3
  ORDER BY Name ASC;

Name
---------------------
0 rows selected.
```

Which pets are named ‘Spot“.’

```
SELECT Name, PetType_Id
  FROM Pet
  WHERE Name = 'spot';

Name  PetType_Id
----- ----------------
0 rows selected.
```

Why did we get no rows back? The where clause said ‘spot’ whereas in the data we have ‘Spot’ - we need a case-insensitive search. We’ll use the built-in function UPPER which converts strings to all upper case letters.

```
SELECT Pet_Id, Name, PetType_Id
  FROM Pet
  WHERE UPPER(Name) = 'SPOT';

Pet_Id Name  PetType_Id
------ ----- ----------
4      Spot  1
5      Spot  1
2 rows selected.
```

Which pets have names beginning with 'S'?

```
SELECT Pet_Id, Name, PetType_Id
  FROM Pet
  WHERE UPPER(Name) LIKE 'S%';

Pet_Id Name    PetType_Id
------ ------- -----------
2      Scrappy 1
4      Spot    1
5      Spot    1
2 rows selected.
```

The ' %' is a wildcard. What other cool things are there to know about the where clause?

The where clause represents one of the first filtering actions that occurs during the execution of an SQL statement Using the where clause we can build up complex selection criteria by joining sub-expressions with relational operators. Here is a list of options available for use in expressions associated with the where clause:

Owners

Operator	Type	Syntax or sample
+, -, /, *, %	Arithmetic operator	
<,>,=, >=,<=, != or <>, !>, !<	Comparison operators	
AND, OR	Relational operators	(Age > 25) Or (JobType= “technical”)
BETWEEN <exp1> AND <exp2>	Range operator	Age Between 35 And 45
IN (a,b,c…)	Set operator	PayCode IN (I. 7, 9 , 12)
NOT	Complement	PayCode Not in (2, 6)
LIKE	String matching operator	CourseName LIKE "BCS_"
%	String matching wildcard	Matches 0 or more characters
_	String matching wildcard	Matches any single character
[ ]	String matching wildcards	Matches any single char enclosed
[^]	String matching wildcards	Matches any single char not specified

The SQL select statement is probably the most used SQL statement. As its’ name Implies. it is used to select information from tables. The full syntax for the select statement is:

```
select [all|distinct] <select-list>
[into <target>]
[from <source>[holdlock][,...]]
[where <search-conditions>]
[group by [all] <aggregate-free-expression> [, .]
[having <search-conditions>]]
[order by {<column-name>[,...] | <column-number>[,...] | <expression>}
[asc|desc]]
[compute <row-aggregate>(<column-name>)[,...]
[by <column-name>(,...]]
[for browse]
```

Which Pets belong to Bob Barker?

This question can be approached in several ways One way is to first find out the Owner_Id for Bob Barker:

```
SELECT Owner_id
  FROM Owner
  WHERE Name = 'Bob Barker';

Owner_Id
---------------------
1
1 row selected.
```

Now we could execute this query:

```
SELECT Pet_Id, Name, PetType_Id
  FROM Pet
  WHERE Owner_Id = 1;
```

```
Pet_Id Name     PetType_Id
------ -------- ----------
2      Scrappy  1
4      Rufus    1
5      Spot     1
6      Candy    2
4 rows selected.
```

That got me my answer. but required two steps. In SQL you can join two tables on a given column Before explaining that take a look at this:

```
SELECT Pet.Pet_Id, Pet.Name, Pet.PetType_Id
  FROM Pet, Owner
  WHERE (Owner.Name = 'Bob Barker') AND
    (Pet.Owner_Id = Owner.Owner_Id);

Pet.Pet_Id Pet.Name Pet.PetType_Id
---------- -------- ----------------
2          Scrappy  1
4          Rufus    1
5          Spot     1
6          Candy    2
4 rows selected.
```

Joins allow you to link multiple tables based on shared values in a given column or set of columns. You can even select columns from any of the joined tables.

```
SELECT Pet.Pet_Id, Pet.Name,
    PetType.Description
  FROM Pet, PetType
  WHERE (Pet.Owner_Id = 1) AND
    (Pet.PetType_Id= PetType.PetType_Id);

Pet.Pet_Id Pet.Name  PetType.Description
---------- --------  --------------------
3          Scrappy   Dog
4          Rufus     Dog
5          Spot      Dog
6          Candy     Cat
4 rows selected.
```

Combining the prior two queries we could write:

```
SELECT Pet.Pet_Id, Pet.Name,
    PetType.Description
  FROM Pet, PetType, Owner
  WHERE (Owner.Name = 'Bob Barker') AND
    (Pet.Owner_Id = Owner.Owner_Id) AND
    (Pet.PetType_Id= PetType.PetType_Id)
  ORDER BY Pet.Name;

Pet.Pet_Id Pet.Name  PetType.Description
---------- --------- -------------------
6          Candy     Cat
4          Rufus     Dog
2          Scrappy   Dog
5          Spot      Dog
4 rows selected.
```

The database figures out how to search the tables and get our result, we just ask what we want with little concern for how the database gets it (sometimes. though. you will regret not thinking about how hard the DB will need to work to get your answer).

Keep trying select statements on your own. Start easy, thinking of real world questions that you might be asked to answer. Here is one to get you going: How many pets are between 3 and 6 years old'?

Updates

The Update statement is used to modify values of columns in a table. The syntax of the statement is:

```
Update tablename

Set assignmentstatement

Where whereclause
```

Here are a few samples of the Update statement:

1. Increase prices in the inventory table by 5%

```
Update Inventory
set Price = Price * 1.05
```

2. Set the number of credits to 5 if a course is over 80 hours:

```
Update CourseCredits
  Set Credits = 5
  WHERE CourseID IN
    (Select CourseId
      From Courses
      Where Hours> > 80);
```

Deletes

The Delete statement facilitates the removal of records from tables. The syntax o f the statement is:

```
Delete From tablename

Where whereclause
```

Here are a few samples of the Delete statement:

1 Delete records from the Students table if they are from Airdrie:

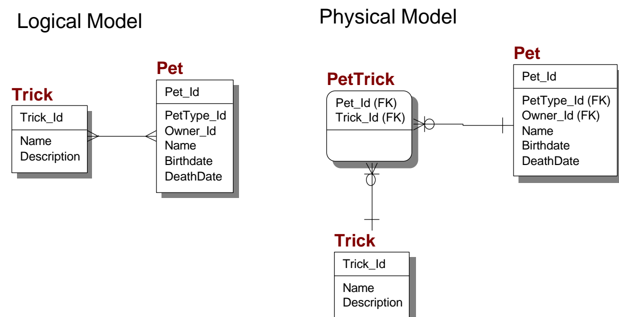
```
Delete FROM Student;
WHERE City Like 'Airdrie%';
```

2. Delete records from Students if they are not currently registered in a course:

```
Delete From Students
Where StatusCode IN
(SELECT StatusCode
From StatusCodes
WHERE description <> "Active")
```

More:

Many-To-Many Relationships



The Logical model shows a Many-To-Many relationship that is each Pet is capable of performing many Tricks, or read the other way any Trick can be performed by many (different) Pets. This is obvious since many dogs know how to fetch, many birds know how to say "hello", and many cats... well. many cats, er, bad example. But you get the idea.

The bad news is that it is physically impossible to implement a many-to-many (think about why this is so, and even if you were 'clever' and managed to do this it would probably violate First Normal Form!).

So instead of giving up we use what is called a resolver table which only contains the keys of the joined tables. Sometimes the resolver table will have its own unique attributes, often start and end dates.

Better Selects: Aggregate Functions, Grouping, and Outer Joins

So what if we want to know some totals, averages, or just a count? Of course we can do this,

```
-- how many pets are there in the database?
SELECT COUNT(*)
  FROM Pet;

COUNT(*)
--------
6
1 row selected.
```

This is useful, but I can well imagine that an even more common question would be "how many of each type of Pet is in the database?". To do this we could repeat the above query using a WHERE clause that limits the search by Pet Type, but there is a simpler way (note that it's simple once you understand it).

SQL allows you to group result sets into what? more sets or subsets. This is done with the Group By clause.

```
SELECT B.Description Description,
    COUNT(A.Pet_Id) PetCount
  FROM Pet A, PetType B
  WHERE (A.PetType_Id= B.PetType_Id)
  GROUP BY Description
  ORDER BY Description;

Description PetCount
----------- -----------
Cat          1
Dog          5
2 rows selected.
```

This is looking good, but wait... there are 6 Pet Types defined, but we only got 2 rows back in the above query Why is this'? It is because the tables are being glued together by PetType Id so if no Pet of a particular Pet Type has been defined in the Pet table this join will not have a row for it. We can resolve this by using an Outer Join. An Outer Join instructs the database to show all of the rows in one table even if the other table does not reference it. I hope I explain this better live than I just did in writing! Anyway here is what the query would look like as an outer join (note that the SQL standard for this is not consistently followed by all vendors, but all of them do have some way of flagging which table is to Included in its entirety. This example does this by placing "(+)" following the lucky table!).

```
SELECT B.Description Description,
    COUNT(A.Pet_Id) PetCount
  FROM Pet A, PetType B
  WHERE (A.PetType_Id(+)= B.PetType_Id)
  GROUP BY Description
  ORDER BY Description;

Description PetCount
----------- ---------
Bird        0
cat         1
Dog         5
Fish        0
Monkey      0
Mouse       0
6 rows selected.
```

Okay, now I'll introduce one more construct. Just as the WHERE clause works on single expressions the HAVING clause operates on aggregate expressions. So if I wanted all of the Pet Types with more than 3 Pets of that type one could:

```
SELECT B.Description Description,
    COUNT(A.Pet_Id) PetCount
  FROM Pet A, PetType B
  WHERE (A.PetType_Id= B.PetType_Id)
  GROUP BY Description
  HAVING (COUNT(A.Pet_Id) > 3)
  ORDER BY Description;

Description PetCount
----------- ---------
Dog         5
1 row selected.
```

The real world application of these features is awesome: getting all customers who have spent more than X dollars with us. our top buyers by dollars spent the most profitable geographic region. sales by Product. etc.

Views

What more is there with DML? Well, if you find that you are always doing the same table joins why not make a logical table that does this for you? It's called a view and it' pretty cool. You define a View with a select statement, and then you select from the View just as you can from any other table. A good candidate for a View is the join between Pet and Pet Type:

```
CREATE OR REPLACE VIEW v_PetPlusType AS
  SELECT Pet.Pet_Id PetId,
      Pet.Name PetName,
      PetType.Description PetType
    FROM Pet, PetType
    WHERE(Pet.PetType_Id=PetType.PetType_Id);
```

Note that you cannot have an Order By clause in a View definition (why?), but you may have one when you use the View in a Select statement, such as to get a list of Pets sorted by name.

```
SELECT * FROM v_PetPlusType ORDER BY PetName;
```

```
PETID PETNAME     PETTYPE
----- ----------- ------------
6     Candy       Cat
1     Gizmo       Dog
3     Rufus       Dog
2     Scrappy     Dog
4     Spot        Dog
5     Spot        Dog
6 rows selected.
```

It's even possible to create views that allow updates, inserts and deletes. Moreover, Views are really useful for hiding sensitive data from other users. You could create a view on an Employee table that hid the Salary column, then grant select on this View to everyone outside of the Human Resources department.

Also you can use Views to hide the ugliness of underlying table structures or to protect developers from tables which keep getting redefined. A View can be the API to the database adding stability and better maintainability to your database projects

Bibliography

SQL Resources

My favorite book which I haven't read, but did skim is:

The Practical SQL Handbook by Bowman, Emerson and Darnovsky. Published by Addison-Wesley Developer's Press. $39.95 380 Pages. Includes CD-ROM with a copy of Personal Sybase and example tables.

These I just saw and they looked alright:

Understanding SQL, by Gruber. Published by Sybes. $26.95 445 pages.

LAN Times Guide to SQL by Groff and Weinberg. Published by McGrall Hill. $29.95 610 pages.

Understanding the New SQL: A Complete Guide by Melton and Simon. Published by Morgan Kaufman.$44.95 394 pages.

SQL Self-Teaching Guide by Stephenson and Hartwig. Published by Wiley. $24 95 200 pages.

Teach Yourself SQL in 21 Days by Stephens et al. Published by SAMS Publishing. $39.99 180 pages

A Guide to SQL published by Pratt. A textbook, essentially.

For the more advanced student I recommend:

SQL For Smarties: Advanced SQL Programming by the demi-god Joe Celko. Published by Morgan Kaufman. $39.95 440 pages of SQL puzzlers and solutions that exercise every ounce of your SQL knowledge!

Other advanced:

A Guide to the SQL Standard. 4th edition. By Date and Danven published by Addison-Wesley. $39.76 400 pages.

SQL, Instant Reference published by Sybex. $19.99 300 pages

Data Modeling Resources

For those really into data modeling (not for the beginner) I recommend highly:

Data Model Patterns: Conventions of Thought by David C. Hay. Published by Dorset House Publishing. $39.95 240 pages. This book presents Data Models for commonly encountered business problems and walks the reader through how they might be designed and why. Really loved this book when It came out - I've bought three copies and had them all stolen. What higher praise is there'?

ERWin Methods Guide. This is part of the documentation that ships with ERwin (the tool I use for doing my Data Models). Its available as a pdf which I will gladly put out on the LAN somewhere if there is any interest. Cost: f:ree! Okay. ERWin costs over $3000, but I don't think we have to pay to circulate a piece of their documentation that has little to do with their product.

Oracle Resources

Understanding the Oracle Server a true insiders guide to the Oracle server presented for the newcomer-- really cool nuts and bolts sections on how the logical abstraction and physical implementation really interact and how these are configured and tuned by your DBA.

Building Intelligent Databases With Oracle PL/SQL Triggers and Stored Procedures. second edition. I had the first edition and found it really useful. The new one is fatter and has a new cover -- it must be great. For the person with an okay understanding of SQL.

Oracle SQL High-Performance Tuning "why is my query so slow?" "what is the effect of RAID5 on an RDBMS?", "What is the performance hit one incurs because of referential integrity ?". This book answers these and other pressing questions for the really curious.

DDL for creating the tables

```
CREATE TABLE Owner (
       Owner_Id Number(10,0) NOT NULL,
       Name Varchar2(65) NOT NULL,
       StreetAddress Varchar2(65) NOT NULL,
       City Varchar2(65) NOT NULL,
       StateCode CHAR(2) NOT NULL,
       Phone Varchar2(20) NOT NULL,
       PRIMARY KEY (Owner_Id)
);

CREATE TABLE PetType (
       PetType_Id Number(4,0) NOT NULL,
       Description Varchar2(65) NOT NULL,
       PRIMARY KEY (PetType_Id)
);

CREATE TABLE Pet (
       Pet_Id Number(10,0) NOT NULL,
       PetType_Id Number(4,0) NOT NULL,
       Owner_Id Number(10,0) NULL,
       Name Varchar2(65) NOT NULL,
       Birthdate DATE NOT NULL,
       DeathDate DATE NULL,
       PRIMARY KEY (Pet_Id),
       FOREIGN KEY (PetType_Id)
           REFERENCES PetType,
       FOREIGN KEY (Owner_Id)
           REFERENCES Owner
);
```

Courts Carter works in the I.S department of Cendant Software and is reachable at ccarter@cendantsoft.com. All of the scripts used in this article are available for download at the MIS web site.