楽しく学ぶ SQL 入門

1. SQL の基礎

1) SQL(Structured Query Language)とは？

簡単に言うと、データベース(以下 DB)を操作するための言語です。

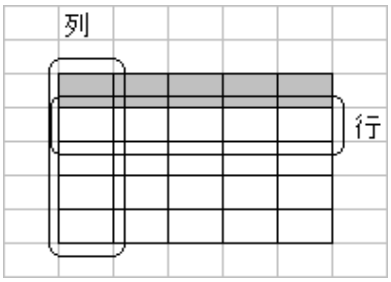
データの検索、入力、更新、削除が最も日常的に使用されますが、データを格納するためのものを作ったり、データを利用する方法を提供したり、果てにはデータベース自身の作成も SQL で行うことが可能となっています。

ANSI や ISO、JIS 等で標準規格化されていて、様々なデータベースに使用することが出来ます。

この業界で Web アプリケーションをやろうと思ったら、まず避けて通ることはできません。

2) RDBMS について

RDBMS は Relational DataBase Management System の略です。



リレーショナルデータベースのデータの格納場所は、右図のように 2 次元の形式を取り、縦軸を列(COLUMNS)と呼び、横軸を行(ROWS)と呼び、その交わるところをフィールドと呼びます。そしてそのフィールドごとに一つの値を格納することが可能です。

列には、データの種類(属性)を定義します。行には、レコード(一行の値)を挿入することが出来ます。

この形をしたデータの格納場所を表(テーブル)と呼びます。

3) KEY(制約)の種類

テーブル内の列には、列ごとに役割(KEY 又は制約と呼ぶ)を与えることが可能です。次にその名称と種類を挙げます。現時点では、さほど重要ではありませんが徐々に登場して参りますので紹介だけここで行っておきます。

・UNIQUE KEY ･･･指定した列に同一の値が入ることを禁止します。

・NOT NULL KEY ･･･指定した列に NULL 値(値が入らないこと)を禁止します。

・PRIMARY KEY ･･･UNIQUE と NOT NULL を合わせた制約です。

・FOREIGN KEY ･･･他のテーブルの列を参照して、参照先の列に存在しない値を禁止します。但し、参照先の列は、UNIQUE か PRIMARY である必要があります。

4) SQL の分類

SQL は、大きく分けて次の 3 つに分類することが出来ます。

・DML(Data Manipulate Language)･･･データ操作文

(例)	SELECT	・・・データ検索
	INSERT	・・・行新規挿入
	UPDATE	・・・値の更新
	DELETE	・・・行削除

・DDL(Data Definition Language)･･･データ定義文

(例)	CREATE	・・・オブジェクトの作成
	ALTER	・・・オブジェクトの変更
	DROP	・・・オブジェクトの削除

RENAME　・・・オブジェクト名の変更
GRANT　・・・権限付与
REVOKE　・・・権限削除
AUDIT　・・・監査
TRUNCATE　・・・表の切り捨て

・DCL(Data Control Language)･･･データ制御文(トランザクション制御)

(例) COMMIT　・・・更新の確定
ROLLBACK　・・・更新の取消
SAVEPOINT　・・・セーブポイントの設定

5)オブジェクトとは？

『オブジェクト』とは、一言で言ってしまうとデータベースの中にある物のことです。

例を挙げると、表(テーブル)、索引(インデックス)、表領域(テーブルスペース)、順序(シーケンス)、別名(シノニム)、ビュー、ユーザー、ロール etc..。

ここで記述した”ユーザー”が、DB に接続するために与えられた名前で、DB のオブジェクトはそれぞれのユーザーの所有物として扱われます。その際にそのオブジェクトの持ち主のことを『スキーマ』と呼びます。もし同一の DB 内で同じオブジェクト名を使用することになった場合は、スキーマ名をはっきりさせる必要があるので右記のように記述します。(例)スキーマ名.オブジェクト名

現時点では、このようなことがある程度の認識でかまいません。後にまた出現しますので、その時に思い起こしてください。

6)トランザクションとは？

トランザクションとは、データが確定からデータの確定までの期間を指します。これに関しても、実際に DCL(データ制御文)を活用する際に、詳細な説明を記述いたしまので、それまでは頭の隅からも追い出してもらって結構です。

2. SELECT 文の基礎

SELECT データを検索、取得するための SQL 文です。以下にその基本構文を示します。

SELECT 列名
FROM テーブル名
WHERE 条件

SELECT 句では、テーブルの列名を指定します。複数の列をしていする場合は、カンマで区切ります。

FROM 句では、テーブル名を指定します。WHERE 句では、検索の条件を指定します。複数の条件を指定する場合は、AND や OR といった論理演算子を使用します。この基本構文さえ覚えていれば大概の検索を行うことが出来ます。「SELECT 句を制する者は SQL を制する」らしいです。

SELECT 句と FROM 句は必須ですが、WHERE 句は、条件の必要な時にだけ記述します。なお、この世の中には様々なデータベースが活用されていますが (Oracle,DB2,SQLServer…etc) 、データベースによっては SQL 文の最後にセミコロン(;)の必要な場合がありますので、記憶しておきましょう。

それでは、次からは実際に SQL の手法を紹介して行きます。

(1) 特定の列の表示

ある特定の列を表示したい場合は、SELECT 句に列名を指定します。

以下に例を示します。

```
SQL> SELECT sportname FROM sport;

SPORTNAME
--------------------
野球
サッカー
バレー
バスケ
ラグビー

5 行が選択されました。
```

SPORTテーブル

SPORTNO	SPORTNAME
01	野球
02	サッカー
03	バレー
04	バスケ
05	ラグビー

(実践)他のテーブルを使って自分で、SELECT 文を書いてみましょう。

(2) 複数の列の表示

複数の列を表示するには、列名をカンマで区切ります。以下に例を示します。

```
SQL> SELECT positionno,positionname
        FROM position;

PO POSITIONNAME
-- --------------------
01 ピッチャー
02 キャッチャー
03 ファースト
04 セカンド
05 サード
06 ショート
07 レフト
08 センター
09 ライト
99 ベンチウォーマー

10 行が選択されました。
```

POSITIONテーブル

SPORTNO	POSITIONNO	POSITIONN...
01	01	ピッチャー
01	02	キャッチャー
01	03	ファースト
01	04	セカンド
01	05	サード
01	06	ショート
01	07	レフト
01	08	センター
01	09	ライト
01	99	ベンチウォーマー

(実践)列を指定する順番を変えて実行してみましょう。

(3) 全件走査

ある表の全ての列を表示したい場合に、全列を指定していたのではものすごい手間になってしまいます。その際に＊(アスタリスク)を使用すれば、全列を指定した場合と同様の結果を得られます。

```
SQL> SELECT * FROM area;

AR    AREANAME
--    --------------------
01    北海道
02    東北
03    関東
04    北信越
05    東海
06    近畿
07    中国
08    四国
09    九州

9 行が選択されました。
```

AREAテーブル

AREANO	AREANAME
01	北海道
02	東北
03	関東
04	北信越
05	東海
06	近畿
07	中国
08	四国
09	九州

(実践)アスタリスクを使用した場合と全列を指定する場合の両方を試してみましょう。

(参考)実際のプログラムでは、アスタリスクを使用せずに全列をきちんと指定することを推奨します。何故ならテーブルが変更された時に列の順番が変わってしまうと、それに相当する SQL 文までも変更するはめになってしまうからです。

(4) 算術式

数値データは、SELECT 句において算術演算子を使用して計算をすることができます。優先順位は以下のように、なっています。

[乗算 , *]⇒[除算 , /]⇒[加算 , +]⇒[減算 , -]

普段、使っている計算と同様にカッコ（）で囲むことも可能で、優先順位も一番高くなります。

```
SQL> SELECT (1+4)*2-6/3 FROM dual;

(1+4)*2-6/3
---------------
          8
```

(参考)今回登場した、DUAL 表は、このようなちょっとした計算等の実践に利用するために存在します。筆者はダミーテーブルと呼んでいます。以下のような日付取得の練習にも使ったりします。いずれ、関数の説明にも出てきますが、どのように使うかは、あなた次第です。

```
SQL> SELECT sysdate FROM dual;

SYSDATE
-----------
00-02-29
```

(参考)SYSDATE とは、現在の時間を取得してくれるパラメータです。Java でも利用されていることでしょう。計算式は［給与*12］=[年棒]という使い方が多いですが、今回は給与の列を用意していませんので、実例は後程ということにします。

(5) NULL 値

値を格納するフィールドに値が存在しない場合には、そのフィールドにはデータが存在しないという意味の'NULL'という値が入ります。先程の計算式の除算で 0 で割るとエラーになりますが、NULL で割った場合は、商は NULL です。

```
SQL> SELECT 5/0 FROM dual;
SELECT 5/0 FROM dual
           *
1 行でエラーが発生しました。
ORA-01476: 除数がゼロです。
```

```
SQL> SELECT 5/NULL FROM dual;

5/NULL
----------
```

(参考)NULL の扱いには注意が必要です。もしプログラムが SQL 文に渡した変数に NULL が含まれていたら、どんな計算結果も NULL になってしまいますし、SQL 文がプログラムに返した値に NULL が、混じっていたらもしかすると、NullPointException が発生するかもしれません。そのための対策の一つとして、SQL 文で NULL が発生した際に利用する関数を紹介します。

NVL(NULL が入るかもしれない値 , 左側が NULL だった場合のみ変換する値)

```
SQL> SELECT NVL(5/NULL,0) FROM dual;

NVL(5/NULL,0)
-------------
            0
```

(参考)今回、用意されてるテーブルにも NULL 値が存在します。実際に NVL 関数を使用した場合と使用しない場合を比較してみましょう。

(6) 文字列の連結、別名

連結演算子（||）を使用して、列と列、リテラル文字と列を連結して表示することが出来ます。

「リテラル文字」とは、SELECT 句に含まれた文字、数字、又は日付のことを言います。文字及び日付のリテラルはシングルコーテーション（‘ ）で囲む必要があります。

SELECT 句に、列以外のデータや連結をした場合に、結果表示を見やすくするために[AS]キーワードを利用して別名をつけることも出来ます。又、[AS]キーワードを使わずにスペースを空けるだけでも別名を使用することができます。

(参考)この連結演算子（||）は、パイプとも呼ばれていますが、本来データの連結はプログラム上で行うのが普通なので、利用機会はまずないでしょう。筆者の経験では利用したのは、PL/SQL を使用した際のみです。

以下のテーブルを利用して例を示します。

Player テーブル				
チーム番号	ポジション番号	背番号	生年月日	選手名
TeamNo	PositionNo	UniformNo	BirthDay	PlayerName

```
SQL> SELECT PlayerName||'の背番号は'||UniformNo||'です。' AS 背番号 FROM player;

背番号
------------------------------------
斎藤隆の背番号は 11 です。
高橋由伸の背番号は 24 です。
古田敦也の背番号は 27 です。
坪井智哉の背番号は 32 です。
バンチの背番号は 42 です。
ディアスの背番号は 49 です。
・・・
（以下略）
```

3．データの制限及びソート

前章では、基本的な SELECT 文ということで、テーブルにあるデータをただ取得するというものでしたが、実際の業務ではテーブル内に何千万件というデータがあっても珍しくありません。その時に適切なデータを取得するためには、条件を指定する必要があります。そこで登場するのが WHERE 句です。

(1)比較演算子

WHERE 句で条件を指定するためには、基本的には比較演算子を利用します。

次にその例を表します。必要に応じて使い分けるようにしてください。

比較演算子	意味
=	等しい
! = ＜＞	等しくない
＞=	以上
＜=	以下
＞	より大きい
＜	より小さい
BETWEEN a AND b	a以上b以下の範囲
NOT BETWEEN a AND b	a以上b以下の範囲外
IN (list)	list内のいずれかと等しい
NOT IN (list)	list内のいずれとも等しくない
IS NULL	NULL値である
IS NOT NULL	NULL値でない
LIKE	文字パターンと一致する
NOT LIKE	文字パターンと一致しない

(参考)算術演算子に、優先順位があったように比較演算子にも優先順位がそんざいします。

複数の比較演算子を利用する際には、この優先順位に注意してください。

１、() カッコ　２、NOT　３、AND　４、OR

(2)AND 演算子

それでは、実際に幾つか例を挙げてみます。

```
SQL> SELECT UniformNo AS 背番号,PlayerName AS キャッチャー
       FROM player
      WHERE PositionNo = '02';

背 キャッチャー
-- --------------------
 8 谷繁元信
59 相川亮二
 9 村田真一
12 村田善則
27 古田敦也
39 矢野輝弘
39 中村武志
32 西山秀二

8 行が選択されました。
```

上記の結果は、登録されている「キャッチャーの名前」を検索したものです。

(3)BETWEEN 演算子

次に、「1975 年生まれの選手」を検索してみます。

```
SQL> SELECT UniformNo 背番号,PlayerName 選手名
  2    FROM player
  3   WHERE BirthDay BETWEEN '19750101' AND '19751231';

背 選手名
-- --------------------
14 森中聖雄
12 村田善則
55 松井秀喜
 7 今岡誠
40 塩谷和彦
13 岩瀬仁紀
47 野口茂樹
14 沢崎俊和
15 黒田博樹

9 行が選択されました。
```

(参考)[BirthDay]には、『日付型』のデータが入っているので、今回のような利用の仕方は例外的です。その『日付型』データ等の基本的な使用法に関しては次章で説明致します。

(実践)ここまでで、幾つか WHERE 条件を自分で考えて試してみてください。

(3)IN 演算子 , LIKE 演算子

続いて『IN』『LIKE』の実例です。

「外野手で背番号が 20 番代の選手を取得します」

```
SQL> SELECT UniformNo 背番号,PlayerName 選手名
  2    FROM player
  3   WHERE PositionNo IN ('07','08','09') AND UniformNo LIKE '2_';

背 選手名
-- --------------------
24 高橋由伸
24 桧山進次郎
23 関川浩一
```

(参考)LIKE に使用できる記号は二つあって、一つ目はアンダーバー(_)です。これが一つあるとそこに一文字入ることを意味します。上記の例では一桁目に 2 で、二桁目が自由、三桁目以降は存在しないということになります。

二つ目の記号は、パーセント(%)です。これは、「文字も桁数も自由」であることを意味します。欲するデータの桁数が決まっていない場合等に利用します。例えば「背番号に 7 が含まれる選手は？」等です。

(実践)上記参考を踏まえて、実際に検索を行ってみましょう。今回は LIKE 演算子に数字を用いましたが、文字も用いることは可能です。もちろん漢字も OK です。

(4)IS NULL

PitcherRecord テーブル						
チーム番号	背番号	投球回	自責点	勝利	敗北	セーブポイント
TeamNo	UniformNo	Inning	LosePoint	Victory	Defeat	SavePoint

NULL 値に関しては前章で説明をしたと思います。今回はその NULL 値の条件検索の例です。

「敗戦の経験がない投手を検索します」

```
SQL> SELECT UniformNo 背番号
  2    FROM PitcherRecord
  3   WHERE defeat IS NULL;

背番号
---------
      49
      50
      12
      23
```

(参考)以上までの条件検索のパターンは、実際のプログラムでも多用されますので、覚えておいて損はしません。ただし、同じデータを取得するにも様々なパターンが存在しますが、テーブルのデータ数が増えてくるとその方法によっては、明らかに検索に時間が掛かってしまうパターンもあります。ここでは詳しくは述べませんが、どんな方法で検索時間が掛かるのかを考えてみてください。

(5)行のソート

これまでのデータの取得は、テーブルに入っている順番で行われていました。ですが、それでは不都合が生じる場合もあります。今回紹介するのは検索時にデータをソート(並び替え)する方法です。

まず、その構文を示します。

SELECT 列名
FROM テーブル名
WHERE 条件
ORDER BY ソート基準 [**ASC**｜**DESC**]

ASC は昇順、DESC は降順を意味します。ORDER BY 句の記述場所は、常に SELECT 文の最後になります。ソート基準には、複数の列を指定することも可能です。その場合はカンマで区切ります。

それでは例を挙げましょう。お題は勝率の高い奴です。

```
SQL> SELECT UniformNo 背番号,Victory 勝数,Defeat 敗数
  2    FROM PitcherRecord
  3   WHERE Victory >= 10 AND Defeat <= 5
  4   ORDER BY Victory DESC;

背         勝数       敗数
-- ---------- ----------
47          12          5
53          11          4
13          10          5
```

(参考)ASC は省略することも可能です。今までの流れでは省略出来たとしてもキチンと記述するというルールがありましたが、今回に限っては特に気にする必要はありません。

(実践)ソート基準には、列の代わりに SELECT 句の列を数字で指定することも可能です。試しに上記の SQL 文の ORDER BY 句の「Victory」を「2」に変えて実行してみましょう。

```
SQL> SELECT UniformNo 背番号,Victory 勝数,Defeat 敗数
  2    FROM PitcherRecord
  3   WHERE Victory >= 10 AND Defeat <= 8
  4   ORDER BY 2 DESC, 3 ASC;

背         勝数       敗数
-- ---------- ----------
42          14          8
47          12          5
42          12          7
53          11          4
18          11          6
13          10          5
18          10          6
19          10          7
```

4. 単一行関数

SQL 文で利用できる関数には、単一行関数とグループ関数があります。この章ではその内の一つである単一行関数のよく使用されるものを紹介します。そして今まで具体的な説明のなかったデータ型についても触れていきましょう。グループ関数については、お決まりの後述ということになります。

(1)データ型

一般にテーブルの列に利用される型を以下に紹介します。

CHAR 型	固定長文字型
VARCHAR2 型	可変長文字型
LONG 型	可変長文字型
NUMBER 型	数字型
DATE 型	日付型
BOOLEAN 型	論理データ型

CHAR 型と VARCHAR2 型は、最もよく利用されるデータ型です。固定長と可変長の違いは読んで字のごとくです。CHAR(10)の列と VARCHAR2(10)の列に 8 文字の文字列を挿入すると VARCHAR2 型では、そのデータに 8 バイトが用意されますが、CHAR 型は常に 10 バイト用意します。

LONG 型は、一見数字のことかと思ってしまいそうですが、文字型です。ですが、筆者もこの LONG 型を利用したことは、片手で数えるほどしかありません。忘れていても問題ないでしょう。

NUMBER 型は、JAVA で言うところの Int 型と同じ数字型です。特筆することはありません。

DATE 型の示す日付型は、表記方法が様々ですが、基本は ’YYYY/MM/DD HH:MI:SS’ の 14 桁です。

BOOLEAN 型も LONG 型と同じくテーブルには、まず使用されることは、ないでしょう。ただ Oracle 独自の言語である PL/SQL には、多々登場しますので、頭の片隅には置いておいてください。

BOOLEAN 型に格納できるデータは、お馴染みの「TRUE」「FALSE」と「NULL」の三つです。

(2)数値関数

CEIL 関数	CEIL(n)・・・n 以上の最も小さい整数を戻す。
FLOOR 関数	FLOOR(n)・・・n 以下の最も大きい整数を返す。
MOD 関数	MOD(m,n)・・・m を n で割った余りを戻す。
POWER 関数	POWER(m,n) m を n 乗した値を戻す。
SQRT 関数	SQRT(n)・・・n の平方根を戻す。
ROUND 関数	ROUND(m,n)・・・ m を小数点以下 n 桁に四捨五入した値を戻す。
TRUNC 関数	TRUNC(m,n)・・・ m を小数点以下 n 桁に切り捨てた値を戻す。

(参考) ROUND 関数と TRUNC 関数の n に負の値(マイナス)を使用することも省略することも可能。

(実践)まず、使うことはないと思ってます。試したい人だけ試しましょう。

(3)文字関数

文字関数は、入力として文字を受け取り結果として文字や数値を戻す関数です。先程紹介した数値関数よりも、はるかに使用頻度が高いです。

LENGTH 関数　LENGTH(char)・・・char の文字数を戻す。

(参考)バイト数を戻して欲しい時は、**LENGTHB** 関数を利用します。

文字数のカウントを行った場合は、**CHAR** 型 と **VARCHAR2** 型では大きな違いが発生します。例えば **CHAR（10）**の列と **VARCHAR2（10）**の列があったとして、それぞれに[横浜優勝]

というデータをセットすると **CHAR** 側の文字数は６で、**VARCHAR2** 側の文字数は４になります。これは、固定長と可変長の違いですね。

SUBSTR 関数　SUBSTR(char , n , m)･･･char の n 番目の文字から m の長さの文字列を抜き出して戻す。m は省略可能。

INITCAP 関数　INITCAP(char)･･･各単語の最初の文字を大文字、残りの文字を小文字にして戻す。空白または、英数字以外の文字で区切られたものを単語とみなす。

UPPER 関数　UPPER(char)･･･char 文字列を大文字にして返す。

LOWER 関数　LOWER(char)･･･char 文字列を小文字にして返す。

CONCAT 関数　CONCAT(char1 , char2)･･･char1 と char2 文字列を結合して返す。

LPAD 関数　LPAD(char1 , n , char2)･･･char1 を n 桁になるように**左**に char2 を埋めて戻す。

RPAD 関数　RPAD(char1 , n , char2)･･･char1 を n 桁になるように**右**に char2 を埋めて戻す。

(4)日付関数

日付のデータは、算術演算子を利用して計算を行うことが出来ます。以下のようなパターンが存在します。

・日付　＋　数値　：　日付に数値分足した日付が戻される。
・日付　－　数値　：　日付に数値分引いた日付が戻される。
・日付　＋　日付　：　エラーが起こります。
・日付　－　日付　：　日付間の日数が戻される。
・日付　＋　数値／２４　：　日付に時間を足した日付が戻される。

それでは、例に習って日付の関数の紹介です。

ADD_MONTHS 関数　ADD_MONTHS(d , n)･･･日付 d に、n ヶ月足した値を戻す。

(参考)月末を引数に入れると結果は、月末になります。試してね。

LAST_DAY 関数　LAST_DAY(d)･･･指定した月の月末日を戻す。

NEXT_DAY 関数　NEXT_DAY(d , char)･･･次の char 曜日を迎える日付を戻す。

MONTHS_BETWEEN 関数 MONTHS_BETWEEN(d1 , d2)･･･日付 d1 と d2 の間の月数を戻す。

SYSDATE 関数　SYSDATE･･･現在の日時を戻す。

(5)変換関数

この章の初めに幾つかのデータ型を紹介しましたが、例えばですが数字は必ずしも数字型の列にしか格納できないのか？というとそうとは、限らなかったりします。データベースによって変わってくることなのですが、他のデータ型への値の格納が暗黙的に可能な場合もあります。

ここで紹介する関数は、その他のデータ型への格納を明示的に行う関数です。

TO_CHAR(日付型)関数　TO_CHAR(d , fmt)･･･書式モデル fmt で指定した d を文字型にして戻す。

TO_CHAR(数値型)関数　TO_CHAR(n , fmt)･･･書式モデル fmt で指定した n を文字型にして戻す。

TO_NUMBER 関数　TO_NUMBER(char , fmt)･･･書式モデル fmt で指定した char を数値型に変換して戻す。

TO_DATE 関数　TO_DATE(char , fmt)･･･書式モデル fmt で指定した char を日付型に変換して戻す。

(参考)今回の引数に登場したフォーマットについても、少しだけ紹介しておきましょう。
本当に少しだけですから、応用は自分達で利かせてください。

・数字型のフォーマット　：　[9999]･･･４桁の整数を示す。
[9999.99]･･･小数部分を２桁保有する。

・日付型のフォーマット　：　[YYYY／MM／DD HH：MI：SS]･･･日時を１４桁で表した場合。

・文字型のフォーマット　：　日付型、数字型の表し方と同様です。

(6)その他の関数

NVL 関数　　NVL(expr , val)･･･expr の値が NULL の場合に、val を戻す。

DECODE 関数　DECODE(expr , val1 , rtn1 , val2 , rtn2 , ･･･ , rtn z)･･･expr の値が val1 の場合 rtn1 を expr の値が val2 の場合は rtn2 を戻し、expr の値がいずれの val でもない場合に rtn z を戻す。但し、rtn z を省略すると NULL が代用される。

```
SQL> SELECT PlayerName  名前,
  2              DECODE( PositionNo ,
  3                                  '01' , '花形' , '02' , '女房役' ,
  4                                  '引き立て役' )  通称
  5       FROM Player
  6     WHERE TeamNo = '04';

名前                           通称
-------------------- ---------
今岡誠                         引き立て役
藪恵壹                         花形
桧山進次郎                     引き立て役
広沢克実                       引き立て役
矢野輝弘                       女房役
遠山奨志                       花形
・・・
（以下略）
```

(参考)この **NVL** や **DECODE** は使い方によっては、かなり便利ですので覚えておきましょう。

5．結合

さて、やって参りました結合です。今まで意味不明なコードを説明や例に利用してきましたが、その苦痛ともやっとおさらばです。結合とは読んで字のごとくでテーブル同士を結合することによって格納されているデータを見やすく、効率よく取得することが可能となる方法です。

結合は直積とも呼ばれ、基本的に複数のテーブルを FROM 句に指定することによって実現します。この際に、結合条件を WHERE 句に記述することになるのですが、WHERE 句の記述なくとも結合は、発生します。WHERE 句に結合条件を記述しない方法を直積演算と呼びますが、筆者は、これの使用を禁じ手としてます。理由は、・・・自分で考えてみましょうね。

それでは、早速ですが結合条件の指定方法を紹介して行きます。

(1) 等価結合

最も基本的な結合方法です。字の通り、等しい値で結合を行う方法です。

Area テーブル		Team テーブル				
地区番号	地区名	スポーツ番号	リーグ番号	地区番号	チーム番号	チーム名
AreaNo	AreaName	SportNo	LeagueNo	AreaNo	TeamNo	TeamName

```
SQL> SELECT Team.TeamName,Area.AreaName,League.LeagueName
2       FROM Team , Area , League
3      WHERE Team.AreaNo = Area.AreaNo
4        AND Team.LeagueNo = League.LeagueNo;

TEAMNAME                  AREANAME             LEAGUENAME
------------------------- -------------------- --------------------
横浜ベイスターズ          関東                 セントラル
読売ジャイアンツ          関東                 セントラル
ヤクルトスワローズ        関東                 セントラル
阪神タイガース            近畿                 セントラル
中日ドラゴンズ            東海                 セントラル
広島東洋カープ            中国                 セントラル
西武ライオンズ            関東                 パシフィック
オリックスブルーウェイブ  近畿                 パシフィック
近鉄バファローズ          近畿                 パシフィック
千葉ロッテマリーンズ      関東                 パシフィック
日本ハムファイターズ      関東                 パシフィック
福岡ダイエーホークス      九州                 パシフィック
```

(参考)FROM 句には、幾つでもテーブルを指定することが出来ます。そして列名を示す際には、その前にテーブル名を記述する必要が出てきます。何故書くのか？それは、その列がどのテーブルの列なのかが、分からなくなるからです。実は、テーブル名を書かずともエラーにはなりませんが、結合を行う際には必ず記述することを推奨します。

尚、FROM 句のテーブル名には、今まで列にほどこしていたような、別名を付ける事ができます。テーブル名が２０文字くらいに長くなることは普通にありますから SELECT 句や WHERE 句に長いテーブル名を書くのが嫌な場合は、別名を大いに利用しましょう。

(2) 非等価結合

これも名前の通り非等価な結合方法です。＝(イコール)の演算子以外の演算子を使用します。

従業員マスタテーブル			給与マスタテーブル		
従業員 No	従業員名	給与	等級	最低給与額	最高給与額

```
SQL> SELECT J.従業員名 , J.給与 , K.等級 , K.最低給与額 , K.最高給与額
2       FROM 従業員 J,給与等級 K
3      WHERE J.給与 BETWEEN K.最低給与額 AND K.最高給与額;

従業員名           給与         等級 最低給与額 最高給与額
---------- ---------- ---------- ---------- ----------
伊藤           180000          3     140001     200000
田中           195000          3     140001     200000
白井           260000          4     200001     300000
長谷川         210000          4     200001     300000
桜井           300000          4     200001     300000
田村           230000          4     200001     300000
藤原           385000          5     300001     999999
林             900000          5     300001     999999
```

(参考)この例を見てもらったら、非等価結合がどんなものかが分かると思う。でもまあ使用頻度のかなり低い結合方法ですので、頭の片隅で良いでしょう。

先程、説明したテーブル名の別名についてもここで使用していますが、FROM 句で別名を使用した場合は、必ず他の句でも使用するように。エラーになります。

(3) 自己結合(内部結合)

続いて自己結合ですが、これは一つのテーブル内で完結する結合です。

従業員テーブル		
従業員 NO	従業員名	上司

```
SQL> SELECT 部下.従業員NO,部下.従業員名,部下.上司,
2              上司.従業員NO,上司.従業員名
3       FROM 従業員 部下,従業員 上司
4      WHERE 部下.上司 = 上司.従業員NO;

 従業員NO 従業員名           上司     従業員NO 従業員名
---------- ---------- ---------- ---------- ----------
      7369 伊藤             7902       7902 桜井
      7566 松岡             7839       7839 林
      7654 高木             7698       7698 藤原
      7902 桜井             7566       7566 松岡
      7934 田村             7782       7782 高橋
      ･･･(以下略)
```

(参考)このように、一つのテーブル内に同じ意味を表す列（従業員 NO と上司）が存在する場合に自己結合を利用します。今回 FROM 句のテーブル名の別名をそれぞれ部下,上司としたのは、もちろん見やすくするためです。

余談ですが、自己結合を利用する際には、**「CONNECT　BY」**句も利用できます。まあ無理に使うことはないので、説明はしませんが興味があれば調べてみてください。

(4) 外部結合

ここまでに紹介してきた結合方法では、指定した列の他方の表の値が **NULL** の場合は、列が検索されて来ません。そこで登場するのがこの外部結合です。他方の値が NULL 又は存在しない場合でも検索することが可能になります。

従業員テーブル			部署テーブル		
従業員 NO	従業員名	部署 NO	部署 NO	部署名	場所

```
SQL> SELECT D.部署 NO,D.部署名,D.場所,E.従業員 NO,E.従業員名
  2    FROM 部署 D,従業員 E
  3   WHERE D.部署 NO = E.部署 NO(+);

   部署 NO 部署名          場所            従業員 NO 従業員名
---------- -------------- ------------ ---------- ----------
        10 経理           東京               7782 高橋
        10 経理           東京               7839 林
        20 研究開発       名古屋             7788 藤井
        20 研究開発       名古屋             7566 松岡
        30 営業           大阪               7499 白井
        30 営業           大阪               7698 藤原
        40 管理           東京
        ･･･(以下略)
```

この結果から分かることは、「管理部には誰もいない」ということです。部署テーブルは部署の種類データを格納しているテーブルですから、部署 NO で結合を行った際に値が存在しないということは、考えられません。反対に従業員テーブルには、全ての部署の人間がいるとは限りません。例えば新しい部署が発足したりすると起きそうな現象です。

そのような時に、『値が存在しないかもしれない』側の結合条件に（＋）を付加する結合を外部結合と呼びます。

6．グループ化

この章では、平均や合計等の行のまとまりごとの情報を算出する方法を紹介します。

(1)グループ関数

DISTINCT　　引数の異なる値だけを処理します。

ALL　　引数の重複する値を含めて全てを処理します。(省略時は ALL です)

ALL のオプションに関しては、まず記述する機会は、ないでしょう。今回一応存在だけは紹介しますが、とりあえず忘れましょう。逆に DISTINCT オプションは利用価値がかなりあります。列の値を表示する時に重複をカットする役割を持ちます。

では、実際に DISTINCT を使ってみます。

```
SQL> SELECT DISTINCT TeamNo FROM Player;

    TEAMNO
    ----------
        01
        02
        03
        04
        05
        06
```

もし上記の SQL 文で DISTINCT オプションを使用しなかったらどうなるでしょうか？

当然 Player テーブルのレコード数が全て表示されることになります。

AVG 関数　　AVG(n)･･･n の平均値を戻す。

もう皆さんならこの関数をどのように使うべきかは分かると思います。

BatterRecord テーブル						
チーム番号	背番号	打数	安打	本塁打	打点	盗塁
TeamNo	UniformNo	BatCount	HitCount	HomeRun	HitPoint	Steel

(実践)安打や本塁打の平均をだしてみましょう。

COUNT 関数　　COUNT(expr)･･･expr が NULL でない行数を戻す。但し、アスタリスクを指定すると重複値及び NULL 値を含めて行数を戻す。

これは、はっきり言って便利な関数です。主な使用目的は行数を数えることです。先程の DISTINCT オプションを組み合わせると、指定した列の種類を数えることも出来ます。

MAX 関数　　MAX(expr)･･･expr の最大値を求める。

MIN 関数　　MIN(expr)･･･expr の最小値を求める。

SUM 関数　　SUM(n)･･･n の合計値を戻す。

この３つの関数の使用法は、AVG 関数と全く同じと考えましょう。

(実践)とりあえず、列を一つ指定してそれぞれの関数を試しましょうね。

```
SQL> SELECT COUNT(*) FROM BatterRecord;

  COUNT(*)
  ----------
        56
```

見ての通り、これはテーブル名のレコード数が 56 行あることを示しています。どんな時に使用するのか？それは、仕事で使うまでのお楽しみということにします。でも覚えてね。

続いて COUNT 関数と DISTINCT オプションを組み合わせたパターンです。

```
SQL> SELECT COUNT(DISTINCT TeamNo) FROM BatterRecord;

COUNT(DISTINCTTEAMNO)
-------------------------------------
                    6
```

結果を見てわかることは、この BatterRecord テーブルの TeamNo 列の値は 6 種類存在するということです。分かりますよね？もし分からないのなら今度筆者を見た時に、文句言ってください。分かるまで責任を持って何時間でも付き合いますので。

(2)GROUP BY 句と HAVING 句

本来グループ化のメインテーマはグループ関数ではなく、この GROUP BY 句と HAVING 句です。テーブルからデータを取得する際に、グループ化したい･･･例えばチーム毎に打率や防御率を表示、よくあるパターンでは、部署やクラスごとの成績を表示したい！なんて時に使います。

それでは、基本構文を示します。

SELECT 列名
FROM テーブル名
[**WHERE** 条件]
GROUP BY 列のリスト
[**HAVING** 条件]
[**ORDER BY** 列のリスト]

GROUP BY 句と HAVING 句は順不同です。どちらを先に書くのかは趣味の世界。もしくは、参加しているプロジェクトの規定に従いましょう。それと以前説明しましたが、ORDER BY 句は一番最後に記述します。

GROUP BY にセットするのは、SELECT 句に用意した列の内 GROUP 関数を使用していない列。そう考えてもらって構いません。つまり逆に考えると SELECT 句に GROUP 関数を使用しなければ **GROUP BY** 使用することはないということです。

まあ論より証拠なので、例を挙げてみましょう。

チーム別に選手の最多本塁打を本数の多い順に並べてみます。

ちょっとレベルが上がりますが、ちゃんとついて来てね。

Team テーブル				
スポーツ番号	リーグ番号	地区番号	チーム番号	チーム名
SportNo	LeagueNo	AreaNo	TeamNo	TeamName

```
SQL> SELECT T.TeamName , MAX(B.HomeRun)
  2    FROM BatterRecord B , Team T
  3   WHERE B.TeamNo = T.TeamNo
  4   GROUP BY T.TeamName
  5   ORDER BY MAX(B.HomeRun) DESC;

TEAMNAME                                 MAX(B.HOMERUN)
---------------------------------------- --------------
読売ジャイアンツ                                     42
ヤクルトスワローズ                                   36
広島東洋カープ                                       30
中日ドラゴンズ                                       23
横浜ベイスターズ                                     20
阪神タイガース                                        5
```

さて、皆さん上記の SQL 文が何をやっているのかは理解できたでしょうか？一応の所は、ここまでに最低一度は、説明を行ったものばかりです。

ここで注意することは、GROUP BY 句には SELECT 句で GROUP 関数を指定しなかった列全てを指定すること。ORDER BY 句には、SELECT 句に記述した列を GROUP 関数ごとでもきちんと記述すること。FROM 句で用意した別名は、他の句において常に使用すること。

まあ、別名に関しては重複しない限りエラーのでることはありませんが、GROUP BY 句は、指定しなかった場合エラーが発生します。

(実践)上記の SQL 文を、GROUP BY 句を外して実行してみましょう。

今回はチーム名を表示していますが、選手名も表示してみましょう。←難問です。

(参考)GROUP BY 句には、複数の列を指定することも出来ます。

HAVING 句には、グループ化をした値に条件を加えることができます。とりあえず例です。

```
SQL> SELECT T.TeamName,MIN(P.UniformNo)  最小背番号
  2    FROM Player P,Team T
  3   WHERE P.TeamNo = T.TeamNo
  4   GROUP BY T.TeamName
  5  HAVING MIN(P.UniformNo) > 1;

TEAMNAME                                 最小背番号
---------------------------------------- ----------
横浜ベイスターズ                                  2
阪神タイガース                                    7
読売ジャイアンツ                                  5
```

検索内容は、最小の背番号が１以上であるチームとその番号を取得。です。

今回は、SELECT 句に使用している列をそのまま HAVING 句に使用していますが、必ずしも同じにする必要はありません。必要に応じて使い分けてください。

(実践)HAVING 句の記述を WHERE 句内で記述してみましょう。エラーになるはず！

さて、ここでちょっとした間違い探しです。下の SQL 文を実行するとエラーになるのですが、その原因を考えてみてください。

```
SQL> SELECT T.TeamName,MAX(B.HitCount)
2       FROM Team T,BatterRecord B
3     WHERE T.TeamNo = B.TeamNo
4     GROUP BY T.TeamName
5     ORDER BY B.Steek DESC;
```

(解答)勘の良い人は、気づいたかもしれませんが、ORDER BY 句は、必ずしも SELECT 句に宣言した列を使用する必要はありません。全然 OK なのです。ですがその SQL 文に GROUP BY 句が混在している場合は、必ず ORDER BY で使用した列を GROUP BY にも記述しましょう。

(参考)HAVING 句は、その性質上プロジェクトで使用を禁じられることが多々あります。
ですから必ずしも覚える必要は、ないのかもしれません。では何故使用を禁じられるのか？
開発で使用するデータベースには、当たり前のように何千件もデータが存在します。しかし、HAVAING は条件指定でデータを取得するために、全件走査を行ってしまうのです。よって使用するためには、細心の注意が必要になりますよね。よって、そんな注意をするくらいなら使用を禁止してしまおうという訳です。

7. 副問い合わせ

副問い合わせとは、他の SQL コマンドの句で使用される問い合わせのことです。まあ簡単言えば複雑な問い合わせを行うためのテクニックと思っていてください。

結合とは二つ以上のテーブルから検索を行う際に使用するテクニックでしたが、この副問い合わせも二つ以上のテーブルを利用します。もっと簡単に言えばSQL文の中にSELECT文を埋め込むことです。

どこに埋め込むことが可能なのかは下記の通りです。

SELECT 文 **FROM** 句

SELECT 文 **WHERE** 句

SELECT 文 **HAVING** 句

UPDATE 文 **SET** 句

UPDATE 文 **WHERE** 句

DELETE 文 **WHERE** 句

※UPDATE 文、DELETE 文は次章に登場します。

(1)単数列副問い合わせ(単一行副問い合わせ、複数行副問い合わせ)

それでは、結合と比べながら一つ例を挙げてみましょうか。

Area テーブル		Team テーブル				
地区番号	地区名	スポーツ番号	リーグ番号	地区番号	チーム番号	チーム名
AreaNo	AreaName	SportNo	LeagueNo	AreaNo	TeamNo	TeamName

チームが存在する地区の名称を取得したいとしましょう。結合を利用するこうなります。

```
SQL> SELECT A.AreaName
  2    FROM Area A , Team T
  3   WHERE A.AreaNo = T.AreaNo;
```

結果は、実行してもらうとわかりますが値が重複することでしょう。ではその重複を解消したい時はどうするのか？みなさんの頭の中にはグループ化が思い浮かぶかもしれません。

```
SQL> SELECT DISTINCT A.AreaName
  2    FROM Area A , Team T
  3   WHERE A.AreaNo = T.AreaNo;
```

そうですね。このようにすれば重複を解消することはできます。ですが、副問い合わせを利用するならこんな感じなります。

```
SQL> SELECT AreaName
  2    FROM Area
  3   WHERE AreaNo IN (SELECT AreaNo FROM Team);

AREANAME
--------------------
関東
東海
近畿
中国
九州
```

結果は、ご覧の通りでちゃんと重複が解消されていますよね。SQL 文を見てもらえば中でどんな処理が行われているのかは、理解できると思います。

もしわからなかったら･･･、今までの章を見直しましょう。それでもわからなかったら･･･、まあ筆者の責任でしょうね。わかるまで相手しますので、しつこく質問して下さい。

(参考)以前にも少し書きましたが、『全件走査』つまりテーブルの中身を全て見るという行為は、かなり良くないです。今回利用しているような数十件程度のデータなら問題ありませんが、データ量が多い時に『全件走査』を行うと結果が返ってくるのにかなりの時間がかかります。

皆さんは、インターネットの画面遷移を何秒なら我慢して待つでしょうか？おそらく５秒待ったら「遅い」と考えるのではないでしょうか？実際の仕事では、このようなパフォーマンスを重視して SQL 文を書く必要がありますので、これからは SQL 文を書くたびに極力『全件走査』をしていないかどうかを気に留めておいてくださいね。

(2)副問い合わせに使用する演算子

副問い合わせの優れたところは、前もって条件を指定することによって外側の SQL 文の検索範囲を狭めることにあります。前回は IN 演算子を利用した例を挙げましたが、他の演算子も使用可能です。ただし内部の SQL 文が複数の列を戻す場合は、[< , > , = , !=, <= , >=]といった演算子は、使用不可です。何故なら１対１の演算子だからです。よって戻される件数がわからない場合は、[IN , NOT IN , ANY , ALL]等を活用しましょう。

それでは、一つ例を挙げましょうか。

```
SQL> SELECT P.PlayerName
  2    FROM Player P , BatterRecord B
  3   WHERE B.HitPoint > (SELECT AVG(HitPoint) FROM BatterRecord)
  4     AND P.TeamNo = B.TeamNo
  5     AND P.UniformNo = B.UniformNo;

PLAYERNAME
--------------------
谷繁元信
石井琢朗
中根仁
鈴木尚典
佐伯貴弘
清原和博
マルティネス
高橋由伸
清水隆行
松井秀喜
古田敦也
･･･以下略します。
24 行が選択されました。
```

検索内容は、[リーグ内の平均の打点以上をマークしている選手名の取得]です。

(実践)この文を参考にして、打点ではなく平均アベレージ以上をマークしている選手名を挙げてみてください。余裕があればそれぞれの選手名にチーム名も付加してください。

(参考)内側の SQL 文の検索結果が 0 件の場合は、外側には NULL を戻すことになります。

(3)複数列副問い合わせ

続いて内部の SQL 文が複数列戻す場合を紹介します。といっても単数列との違いは、使用する演算子だけなのでそんなに違いを意識することはありません。

```
SQL> SELECT P.PlayerName
  2    FROM Player P , BatterRecord B
  3   WHERE (B.TeamNo,B.UniformNo) IN (SELECT TeamNo,UniformNo
  4                                      FROM BatterRecord
  5                                     WHERE HitCount/BatCount > 0.3)
  6     AND P.TeamNo = B.TeamNo
  7     AND P.UniformNo = B.UniformNo;

PLAYERNAME
--------------------
金城龍彦
石井琢朗
種田仁
石井義人
中根仁
･･･以下略します。
14 行が選択されました。
```

どうでしょう？単に演算子が代わって、内側の SQL 文の戻り値が複数になっただけでこれまでの副問い合わせと何ら変わらないことが分かりましたか？それでは・・・。

(実践)IN の代わりに NOT IN を使ってみましょうか。上記の結果とは逆の結果が検索されることでしょう。その次に、NOT IN を使って内部からの戻り値に NULL が混じるような SQL 文を書いてみてください。結果は・・・、面白いことになります。筆者に言えることは、NULL が戻り値に混ざる可能性がある場合は、注意が必要だってことです。

(4)FROM 句の副問い合わせ

FROM 句内での副問い合わせは、そのビューを一時的に作成することと同様です。ビューに関しての説明は、後の章で行うので簡単にいうと、架空のテーブルです。実際に例を挙げましょうか。

```
SQL> SELECT UniformNo , PlayerName
2      FROM ( SELECT PositionNo , UniformNo , PlayerName
3               FROM Player
4              WHERE TeamNo = '01' )
5    WHERE PositionNo = '01';
```

結果と説明は、紙面の都合で省略します。というよりビューの章で同じことします。

8. データの挿入、更新、削除、確定

さて、やっと SELECT 文を抜け出しました。章が一つ終わってもまた SELECT か～っていうくらいもう正直言って筆者も飽きてます。そう今までは、テーブル内のデータを検索することしかしませんでしたが、この章ではそのデータを作成したり更新したり抹消したりします。

このテキストの最初に書きました DML(Data Manipulate Language)文というのを覚えているでしょうか？この章ではデータを操作する文の残り３つを紹介します。

(1)INSERT 文･･･データの挿入

まずは、基本構文を示します。

INSERT INTO 表名 [**(** 列名 **,** 列名 **,** ･･･**)**]
VALUES (値 **,** 値 **,** ･･･**)**

[列名]を指定した順に[値]を指定します。もちろん双方の指定する数は同じでないといけません。そして[列名]は省略することが可能です。しかしその時は、そのテーブル内の全列に対して[値]を指定しなければなりません。複数の[列名]や[値]を指定する時は、カンマも必須です。

この INSERT 文の特徴は、レコードを作成することなんですが、レコード作成という性質上一つずつしか作成出来ないという事ですね。SELECT 文は、複数件の選択が可能です。UPDATE も DELETE も複数件を扱うことが可能です。ですが INSERT だけは、一つずつです。

それでは、例を挙げてみましょう。

```
SQL> INSERT INTO Player(TeamNo, PositionNo, UniformNo, BirthDay ,PlayerName)
  2  VALUES('07','01','18',TO_DATE('1979-01-01','YYYY-MM-DD'),'松坂大輔');

1 行が作成されました。
```

行っている内容は、Player テーブルに‘松坂’のデータを挿入というだけです。値がない場合は、NULL もセット出来るし、日付に今の時間を使用したい時は、SYSDATE 関数も使えます。その他には特に何の説明はいらないと思います。テーブルの列の属性に合わせてデータをセットするだけです。属性って何？って思う人は、･･････テキストを最初から読み直しましょう。

(参考)今までは、検索のみだったのであまり注意しませんでしたが、列には制約というものがあります。NULL を挿入させない制約もありますし、UNIQUE 制約や PRIMARY 制約の場合は、同じ KEY 項目を持つデータを挿入させません。ですから上記の INSERT 文を連続で発行するとエラーが発生します。試してみてください。

もう一つ付け加えると、上記の例では全列に値をセットしているわけではありません。じゃあセットしなかった列はどうなるか？自動的に NULL がセットされます。(例外もあります。後述。)実際の仕事では、必ず列の指定を記述することを推奨します。

(2)UPDATE 文･･･データの更新

まずは、基本構文を示します。

UPDATE 表名
SET 列名 = 値 [, 列名 = 値, 列名 = 値, ･･････]
[**WHERE** 条件]

SET 句に変更を行いたい列とその値を指定します。複数の列を指定する場合はカンマで区切ります。WHERE 句では、変更を行いたい行の条件を指定します。指定方法は、SELECT 文と同様です。よってその条件に該当する複数の行が変更されることもあります。そして WHERE 句は省略することも可能ですが、その場合はテーブル内の全行が変更対象となります。

それでは、例を示します。先程挿入した‘松坂’を筆者は横浜ベイスターズに引き抜きます。

```
SQL> UPDATE Player
  2     SET TeamNo = '01', UniformNo = '99'
  3   WHERE TeamNo = '07'
  4     AND UniformNo = '18';

1 行が更新されました。
```

WHERE 句にて、‘松坂投手’を限定で指定し SET 句にてチームコードを横浜ベイスターズのコードにセットし直しました。他には、この SQL 文について説明はいらないでしょう。

(参考)勘の良い人は気づいてるかもしれませんが今回変更を行っている列は、テーブル内の行を一意に判断する列(KEY 項目)です。実は、このような KEY 項目の変更はデータベースによっては行えないことがありますので、少々注意が必要です。筆者が好んで使用する 0racle でもバージョンによっては不可能です。

もう一つ気をつけるとすれば、UPDATE 文は、1 行も更新できなくてもエラーは発生しません。ですが、実際の業務で UPDATE 文を発行する時は、必ず何行かは更新したいはずです。よって業務で UPDATE を発行した時は、1 行も更新しなかった場合を明示的にエラーとして扱ってやる必要があるということです。

(3)DELETE 文･･･データの削除

まずは、基本構文を示します。

DELETE FROM 表名

[**WHERE** 条件]

WHERE 句で指定した条件に一致する行を全て削除する SQL 文です。WHERE 句は、省略することが可能です。但しその場合はテーブル内の全行が削除されてしまいます。そしてデータベースによっては、WHERE 句と一緒に FROM 句すらも省略することが可能です。

今回は、例を省略します。皆さんの創造力におまかせということです。

(参考)以前にも書いたかもしれませんが、実際の仕事でプログラムする SQL 文は、SELECT,INSERT,UPDATE だけです。DELETE 文は、まず書くことはないと思います。何故なら！情報が命のこの業界でその肝心のデータを消してしますような SQL 文を書くわけないでしょう？例外としては履歴がちゃんとあるときや、ワークテーブルの場合くらいですが、どちらもバッチ処理に限られます。バッチ処理に関しては今回の教育目的から外れてるという理由の元に勝手ながら省略します。すいません。

9. トランザクション制御

先に断って起きますが、SQL をかじった程度でトランザクションまで理解するのは少々きついと思います。意味不明だと思ってしまったら、この章は飛ばしてしまってください。ですが、業務においては重要度が高いだろう（特に排他制御）という筆者の判断から勝手に付け加えることにしました。ですが文章において表現するのが難しい所もありますので、その時は教育担当に質問を投げてください。

トランザクションとは、データが確定からデータの確定までの期間を指します。ではデータが確定しているとは、どういうことか？一言でいうと誰が見ても同じ状態のことです。

DML(データ操作)文を発行するとテーブル内のデータが変わります。ですが実はこの時点ではこのデータは確定していません。実は、変更を行った本人にしか見えない状態となります。じゃあ他人にも見えるように変更を反映させたいときはどうするか？そこで DCL(データ制御)文の登場というわけです。

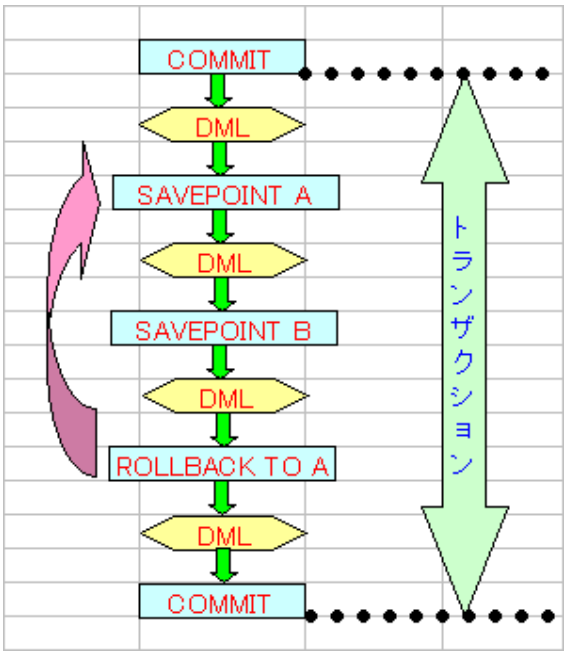
COMMIT ・・・更新の確定
ROLLBACK ・・・更新の取消
SAVEPOINT ・・・セーブポイントの設定

この COMMIT を使用すれば、自分の行った変更を他人にも閲覧できるように確定することが出来るのです。逆に ROLLBACK を使用すると、自分の行った変更を他人の目が触れる前に取り消すことが出来ます。そして SAVEPOINT は、ROLLBACK するところを指定するのに利用します。

右にそのトランザクションの一例を挙げていますが、要は変更を行った際は最後に必ず COMMIT を発行しましょうということです。

一つのトランザクションが終了すれば、次のトランザクションがすぐにスタートします。その時にその時点で確定されているデータは閲覧もしくは操作が可能になります。

これによってデータの整合性が保たれ、誰がテーブル内を変更していても同じ状態を閲覧することが出来ます。これを「読み取り一貫性」と呼びます。



(参考)DML 文では、明示的に COMMIT を発行しない限りデータは確定しませんが、DDL(データ定義)文では、COMMIT を発行しなくとも暗黙的にデータが確定されます。これを「暗黙のコミット」と呼びます。

10. テーブル

以前、データベースのオブジェクトというものの種類を幾つか挙げたと思いますが、この章ではその中の一つであるテーブル(表)について記述します。

テーブルは、データベースの中で実際にデータを格納しているオブジェクトです。この章でそのテーブルの作成、変更、削除について記述しますが、テーブルの構成はプロジェクトの設計に左右されますし、そう滅多に使用する SQL 文でもありませんので、無理に覚える必要はありません。

(1)テーブルの作成

```
CREATE TABLE 表名
        ( 列名 データ型 [ DEFAULT 式 ]
        [ , 列名 データ型 ] , , )
        [ TABLESPACE 表領域名 ]
        [ 記憶領域管理パラメータ ]
```

列に DEFAULT オプションを使用することによって、その列にデフォルトのデータを指定することが出来ます。以前 INSERT 文の説明時に「列名を指定しなくても値がセットされる」という例外があると書いてあったと思います。その例外がこの DEFAULT オプションです。

TABLESPACE は、テーブルを格納する領域のことですが、現時点でそこまで理解してもらう必要はありませんので説明は省略します。同じように記憶領域管理パラメータに関しても全く理解する必要がないので、説明を省略します。どうしても知りたいなら直接筆者に聞いてくれても構いませんよ。

それでは、簡単な例を示します。

```
SQL> CREATE TABLE    Position (
  2                  SportNo          VARCHAR2(2) ,
  3                  PositionNo       VARCHAR2(2) ,
  4                  PositionName     VARCHAR2(20)       NOT NULL,
  5                  UpdateTime       DATE    DEFAULT    SYSDATE    NOT NULL,
  6        PRIMARY KEY( PositionNo , SportNo ));
```

今回のテキストにも登場している Position テーブルの CREATE 文です。VARCHAR2 とは、文字型のことです。データベースによっては、様々なデータ型がありますので必ずしも文字型を指定する時に、VARCHAR2 を利用するわけではありません。そしてその直後に記述している(20)は、バイト数を示します。つまり全角の文字なら 10 文字の値しかこの列にセット出来ないことを示します。

基本構文には、記述しませんでしたがテーブルの列には制約を指定することも出来ます。**NOT NULL** 制約は、上記の通り列ごとに指定します。もし指定しなければその列には、**NULL** をセットすることが可能になります。PRIMARY KEY 制約は、指定した列の値によって行が一意に限定され(UNIQUE KEY 制約)、尚且つ NULL を許さない(NOT NULL 制約)という役割を合わせ持つ制約です。

DEFAULT オプションには、式のほかに、リテラルや SYSDATE 等の SQL 関数をセットすることが出来ます。

最後にもう一つ付け足すなら、一人のユーザーが同じ名前のテーブルを２つ以上作成することは出来ません。エラーが発生します。

(2)テーブルの変更(列の追加)

ALTER TABLE 表名

ADD (列名 データ型 [, 列名 データ型] **)**

列の追加には、ADD コマンドを使用します。列を追加した場合は DEFAULT オプションを使用しない限りその列には、NULL がセットされます。この新規の列は、既存のテーブルの最後に追加されます。

(3)テーブルの変更(列の変更)

ALTER TABLE 表名

MODIFY (列名 データ型／NOT NULL 制約
[, 列名 データ型／NOT NULL 制約] **)**

列の変更には、MODIFY コマンドを使用します。列の変更を行うにあたって幾つかの制限があります。以下にそれを挙げておきましょう。

- 列のサイズ(バイト数)を大きくするのは常に可能。
- 列のサイズを小さくするには、そのテーブルにデータが全くないか、変更を行う列のデータが全て NULL であること。
- NOT NULL 制約を追加する時は、そのテーブルにデータが全くないか、変更を行う列のデータに NULL が存在しないこと。
- MODIFY 句を使用して変更可能な制約は、NOT NULL 制約のみ。
- 列の DEFAULT 値を変更した場合は、今後のデータ挿入時より有効となる。

(4)テーブルの変更(列の削除)

ALTER TABLE 表名 **DROP COLUMN** 列名

列の削除を行うにも幾つか制限があります。以下に挙げます。

- １度に削除できる列は１列のみ。
- 最後の１列は削除できない。
- １度削除した行は復元できない。(暗黙のコミットのため)
- 列にデータが格納されていても削除は可能。

(5)テーブル名の変更

ALTER TABLE 元の表名 **RENAME TO** 新しい表名

今までのように ALTER を利用してテーブル名は変更できますが、実はもう一つ似たような記述方法がありますので、以下にその構文を記述します。

RENAME 元の表名 **TO** 新しい表名

両方とも実行内容は全く同じなので、どちらを利用しても構いません。

(6)テーブルの削除

DROP TABLE 表名 [**CASCADE CONSTRAINTS**]

テーブルの削除を行うにあたって、エラーの発生するパターンが一つだけあります。それは参照整合性制約(FOREIGN KEY)です。これを回避するためには、その制約を先に解除するか、もしくは、CASCADE CONSTRAINTS 句を利用して制約を同時に削除するかの２通りです。

(7)制約の変更(NOT NULL 制約は除く)

制約の変更を行うためには、1 度制約を削除してから新たに追加を行う必要があります。さらに、既にテーブル内にデータが存在する場合には、そのデータが追加する制約を満たしていないとエラーとなってしまうので注意が必要です。

ではまず、制約の追加の構文です。

ALTER TABLE 表名 **ADD CONSTRAINT** 制約名 **制約(** 列名 [, 列名 , ･･･] **)**

制約の追加には列の追加と同様に ADD コマンドを使用します。以下に例を示します。

```
SQL> CREATE TABLE 社員(
  2                社員 No            VARCHAR2(20),
  3                社員名             VARCHAR2(20));

表が作成されました。
```

```
SQL> ALTER TABLE 社員 ADD CONSTRAINT PK_社員 No PRIMARY KEY(社員 No);

表が変更されました。
```

少々意味不明かも知れませんね。最初の CREATE 文は今回の制約の説明用に作ったテーブルで見ての通り作成時点では何の制約も追加してません。次の ALTER 文にて制約を追加していますが、【PK_社員 No】というのは、今回追加した PRIMARY KEY 制約の名称で、適当な名称を付ける事ができます。以前 CREATE 文を作成した時に制約名というものは何も記述しませんでしたが、その時はデータベースが自動的に制約名を付加してくれていたのです。

続いて制約の削除の構文です。

ALTER TABLE 表名 **DROP CONSTRAINT** 制約名

```
SQL> ALTER TABLE 社員 DROP CONSTRAINT PK_社員 No;

表が変更されました。
```

特に説明はいらないと思いますが、前回追加した【PK_社員 No】という制約を指定して削除を行いました。

(8)制約の有効と無効

データの挿入を行う際に、制約が邪魔になることがたまにあります。かといって毎回追加と削除を行うのが嫌な場合は、制約を一時的に無効にし後に有効に戻すことが可能です。

まず制約を無効にする構文です。

ALTER TABLE 表名 **DISABLE CONSTRAINT** 制約名 [**CASCADE**]

続いて制約を有効にする構文です。

ALTER TABLE 表名 **ENABLE CONSTRAINT** 制約名

今回は例を記述しませんが、特に問題ないだろうと筆者は勝手に思ってます。CASCADE オプションを指定した場合は、それに依存する整合性制約も同時に無効にすることが出来ます。

もう一つ挙げるなら、制約を有効にする場合はテーブル内のデータが制約を満たしていないと制約の有効化は失敗するということですね。

(9)既存テーブルからのテーブル作成

テーブルの作成方法には、1から作成する方法以外に既存テーブルから抽出して作成する方法があります。以下に構文を示します。

CREATE TABLE 表名 **AS** SELECT 文

この構文の SELECT 文には、今まで使ってきた SELECT 文をそのまま使用できます。もちろん複数のテーブルからも作成することも可能です。以下に例を示します。

```
SQL> CREATE TABLE 一覧表 AS
  2  SELECT S.SportName スポーツ名, L.LeagueName リーグ名,
  3         A.AreaName 地区名, T.TeamName チーム名
  4    FROM Sport S , League L , Area A , Team T
  5   WHERE T.SportNo = S.SportNo
  6     AND T.LeagueNo = L.LeagueNo
  7     AND T.AreaNo = A.AreaNo;

表が作成されました。
```

(参考)このように、それぞれの列に別名を付けることも可能です。元になったテーブルの値は反映されています。但し、制約に関しては NOT NULL 制約のみ反映されますので、他の制約は後から追加してあげる必要があります。

11. ビュー

ビューとは、副問い合わせの説明で少しだけ触れましたが、架空のテーブルのことです。テーブルには実際のデータが格納されますが、正規化やセキュリティの関係上でビューを通したほうがデータを閲覧しやすい、又は管理しやすいために利用されます。

例えば、今回 SELECT 文を書く時にチーム名や選手名を参照するだけでも常に結合等を行う必要がありましたが、正直それがまどろっこしいと思うこともあるでしょう。最初から一つに固めていれば苦労はないのにと思うかも知れません。そもそもあれだけテーブルがバラバラになっていたのは、正規化のためなのですが、今回は正規化の説明は致しません。

(1)単一ビュー＆複合ビュー

ビューは、基本的にテーブルと同じような扱い(DML 文の発行)が可能ですが、次に挙げる 2 種類のビューによって少々性質が変わってきます。

単一ビュー

- 一つのテーブルから作成される。
- 関数を含まない。
- グループを含まない。
- ビューからの追加、更新、削除が可能である。

複合ビュー

- 複数のテーブルから作成される。
- 関数を含む。
- グループを含む。
- ビューからの追加、更新、削除が不可能である。

（筆者が見た参考書には、条件付きで可能と書いてあったが、筆者は見たことありません）

(1)ビューの作成

CREATE [**OR REPLACE**] [FORCE | NOFORCE]
VIEW ビュー名 **AS** SELECT 文 [**WITH READ ONLY**]

ビューの作成は、【既存表からのテーブル作成】と全く同じ方法を取ります。ただ一つ違うことは、ビューには変更を行う時に ALTER 文が存在しません。そのため、代わりに OR REPLACE オプションを使用して置換を行うことが出来ます。このオプションは同じビュー名が存在する時に効果を発揮しますので、常に記述していても問題はありません。筆者は常に記述することを推奨します。

WITH READ ONLY オプションは、そのビューを読み取り専用にしたい時に付加します。よって複合ビュー作成した時は、自動的に読み取り専用になりますのでその時は不要ですね。

FORCE オプションを指定すると元になるテーブルが存在しないか、元の表を参照する権限がなくとも強制的にビューを作成します。NOFORCE オプションは、その逆でビューの作成者が元になる表をちゃんと使用できる状態の時のみビューを作成可能とします。省略した場合のデフォルトは NOFORCE です。でもまあ深く気にすることは止めときましょう。筆者もこのオプションは気にしたことがありませんので、ありからず。

ちなみに、例も【既存表からのテーブル作成】と同じなので省略します。

(2)ビューの削除

DROP　VIEW ビュー名

ビューの削除に関しては、特に何の制限もありませんので、説明も要らないでしょう。

またまた省略です。

(3)インラインビュー

FROM 句内で行う副問い合わせは、ビューを一時的に作成することと同様だと説明したことがあると思います。覚えてるかな？その時は、下記の例を使用しました。

```
SQL> SELECT UniformNo , PlayerName
2       FROM ( SELECT PositionNo , UniformNo , PlayerName
3                    FROM Player
4                   WHERE TeamNo = '01' )
5     WHERE PositionNo = '01';
```

結果は、もう試されてると思いますが、チームコード=‘01’のデータを内側のテーブルから取得し外側の条件でポジションコード=‘01’のデータを選択しただけです。

じゃあ一体この方法にどんな利点があるのでしょうか？

一つ目は、内側の SELECT 文でデータを絞り外側で結合やグループ化等の【全件走査】を行う場合のパフォーマンスの向上を図るためです。

二つ目は、データの統計等を取得したい際に上位何件だけ欲しい、又は下位何件だけ欲しいという条件の場合に有効です。以下にその例を示します。

```
SQL> SELECT チーム名, 勝利, 敗北
  2    FROM (SELECT TeamName チーム名, SUM(Victory) 勝利, SUM(Defeat) 敗北
  3            FROM Team T, PitcherRecord P
  4           WHERE T.TeamNo = P.TeamNo
  5           GROUP BY TeamName
  6           ORDER BY SUM(Victory) DESC)
  7  WHERE ROWNUM <= 4 ;

チーム名                                                 勝利        敗北
-------------------------------------------------- ---------- ----------
中日ドラゴンズ                                             67         61
横浜ベイスターズ                                           60         62
読売ジャイアンツ                                           60         38
阪神タイガース                                             50         55
```

どうでしょう、検索内容は理解出来ましたか？ROWNUM というのはテーブル内のデータの行数を指します。つまり今回の条件では【テーブル内のデータを上から順に４つだけ取得する】ということを示しています。ランキングの取得には、かなり使える方法と思います。理解出来たら、打撃成績や投手成績の上位 10 位を選出してみてください。

さて、これで残念ながら SQL の基礎講座は終了です。基礎といってもこれだけで仕事には充分通用しますので自信を持ってもらって結構です。皆さんのこれからの精進に期待します。

筆者：阪上靖弘