基本ルール

2018年12月17日 月曜日　　11:18

SQL文に書き込まれたデータそのものを特にリテラルという。
‘’で括られたリテラルは、基本的に文字列情報として扱われる。
‘2018-02-03’のように‘’で括り、一定の形式で記述すると日付情報として扱われる。

代表的なデータ型(各DBMS製品で異なる)
　数値：整数-INTEGER
　　少数-DECIMAL, REAL
　文字列：固定長-CHAR　空きは空白で埋める　e.g. 郵便番号や社員番号など
　　可変長-VARCHAR　入力に合わせた領域を確保
　日付と時刻：DATETIME, DATE, TIME (日付はDBMS間で違いが比較的大きいので注意)

SELECT：列名 FROM テーブル名 WHERE その他の修飾(検索結果の加工)
UPDATE：テーブル名 SET 列名 ＝ 値 WHERE
DELETE：FROM テーブル名 WHERE
INSERT：INTO テーブル名(列名) VALUES(値)

SELECT, UPDATE, DELETE, INSERTはDMLと総称される。

SQLのルール
　記述の途中での改行可能
　予約語は大文字、小文字の区別はされない。列名などに使用は不可
　文中にコメントの記述可能

SELECT文

2018年12月17日 月曜日　　12:48

データベースとデータのやり取りをするにあたって、最も頻繁に使用される。
テーブルから目的のデータを指定して取得することがその役割。

SELECT 列名・・・
　FROM テーブル名
(WHERE修飾)
(その他の修飾)

ASによる別名の定義

SELECT文における列名やテーブル名の指定では、それぞれの記述の後ろに
「AS ＋ 任意のキーワード」をつけることで別名を定義できる。

SELECT ＊による全列検索は便利だが、データベースの設計変更で列が増減すると、
検索結果も変わり、予期しないバグの原因になることがある。

UPDATE文

2018年12月17日 月曜日　　12:56

UPDATE文は、すでにテーブルに存在するデータを書き換えるための命令

UPDATE テーブル名
　　SET 列名＝値,・・・・
(WHERE修飾)

WHERE句のないUPDATE文は全件更新になる

DELETE文

2018年12月17日 月曜日　　12:59

DELETE文は、すでにテーブルに存在する行を削除するための命令。
行を丸ごと削除する機能であるため、特定の列だけを指定することはほぼない。

DELETE FROM テーブル名
(WHERE修飾)

DELETE文では列名を指定する必要がない。

WHERE句のないDELETE文は全件削除になる

INSERT文

2018年12月17日 月曜日　　13:03

INSERT文は、テーブルに新しいデータを追加するための命令。
テーブルの行を指定しくみ(WHERE)はないが、どこにどのようなデータを追加するのかを指定する構造になっている。

INSERT INTO テーブル名(列名1, 列名2, ・・・・, 列名n)
　　VALUES (値1, 値2, ・・・・, 値n)

INTOに続けて、データを追加するテーブル名を記述する。
さらにテーブル名の後ろに、括弧で括ってデータを追加する列名を指定する。
そのテーブルのすべての列に値を追加する場合には、省略可能。

VALUES句には記述した列名に対応するデータの値を指定する。
列名を省略した場合は、テーブルの全ての列について値を指定する必要がある。

4大命令の分類法

2018年12月17日 月曜日　　13:11

検索系：SELECT
更新系：UPDATE DELETE INSERT

検索系の命令はDBのデータを書き換えることはなく、実行結果は表の形になる。
更新系の命令はDBのデータを書き換える。結果は成功か失敗のみで表が返されることはない

既存系：SELECT UPDATE DELETE
新規系：INSERT

既存系は、すでにDBに存在するデータに対して何らかの処理を行う。
検索や更新、削除は既存のデータに対して行う処理であり、対象行を指定(WHERE句)を利用可能

新規系は、DBに存在しないデータについての指定をする。
既存データに対する処理はしないので、WHERE句の利用はできない。

WHERE句-1

2018年12月17日 月曜日　　14:49

WHERE句の基本

処理対象行の絞り込みに用いる → WHEREを指定しないと全ての行が処理対象になる

SELECT UPDATE DELETE で使用可能 → INSERTでは使用できない

WHEREの後ろには条件式を記述 → 絞り込み条件に沿った正しい条件式を記述する

WHERE句には、必ず結果が真か疑となる条件式でなければならない

<>は左右の値が正しくないを意味する。PG言語の!=にあたる。

NULL：そこに何も格納されていない未定義であることを示す。0や空白文字とは異なる。
→ NULLは演算子では判定できない。NULL判定はIS NULL 、非NULL判定はIS NOT NULL

条件式の結果は常に真か疑のどちらかであるが、厳密にはUNKNOWN(不明、計算不能)がある。

条件式にDATE型を用いる場合には、じこくじょうほうについて時刻情報について注意が必要
2013年3月以前のデータを抽出するために 日付 <= '2013-03-31' とした場合、時刻を指定していないためDBMSによっては2013-03-31 00:00:00 と解釈される可能性がある。その結果、2013-03-31の00:00:01から23:59:59までのデータが条件に合致しないことになる。
こういう場合は、翌日より過去という条件(2013-04-01)で指定する。

WHERE句-2

2018年12月17日 月曜日　　15:09

LIKE演算子

パターンマッチングに利用。部分一致の検索が容易に行える。

%：任意の0文字以上の文字列

_：任意の1文字

パターン文字を検索するにはESCAPEを使用。 e.g. hoge LIKE ‘%hoge$%’ ESCAPE ‘$’

BETWEEN演算子

ある範囲内に値が収まっているかを判定

IN演算子

値が括弧内に列挙した複数の値の何れかに合致するかを判定。

＝では1つの値しかひかくできないが、INでは一度に複数の値との比較が可能。

e.g. WHERE hoge IN (‘foo’, ‘bar’) ← 否定の場合は NOT IN

ANY/ALL演算子

複数の値と大小を比較したい場合に用いる。

必ず直前に比較演算子をつけて利用する。これにより複数の値との比較を一度に行う。

DBMSによっては、副問い合わせでしか使えないことがある。

ANYは値リストと比較して何れかが真なら真、ALLは全て真なら真

そのほかにもAND OR NOT演算子が存在する。使い方はPG言語と同じ

優先順位はNOT＞AND＞ORの順。条件式に括弧をつけると評価順位を引き上げ可能

g

主キーと必要性

2018年12月17日 月曜日　　15:29

主キー

値を指定することで、ある1行を完全に特定できる役割を担う列のこと
必ず何らかのデータが格納される(NULLではない)
他の行と値が重複しない
全てのテーブルは主キーとなるような列を必ずもつべき

自然キー：氏名、社員番号のように自然に登場し、主キーの役割をはたせる列

人口(代替)キー：入出金IDのように管理目的のためだけに人為的に追加された列

複合キー：単独では重複の可能性があっても、複数組み合わせれば重複する可能性がなくなる場合があり、それらを主キーとして扱ったもの

検索結果の加工

2018年12月17日 月曜日　　11:15

DISTINCT：SELECT文に付加すると、結果表の中で内容が重複している行があれば除去
e.g. SELECT DISTINCT foo FROM bar

ORDER BY：SELECT文の最後に記述すると、指定した列の値を基準として、検索結果を並べ替えて取得することができる。ASCは昇順、DESCは降順。デフォは昇順。
文字列の並べ替えは、文字コード、アルファベット順などになる。
e.g. SELECT hoge FROM foo ORDER BY hoge ASC
SELECT * FROM foo ORDER BY bar DESC, hoge DESC ← 列番号での指定も可能

LIMIT：先頭から数行だけ取得する。OFFSETで途中から取得可能。
e.g. SELECT hoge FROM table LIMIT 取得行数 (OFFSET 先頭から除外する行数)
LIMIT 3 OFFSET 10 (11行目から14行目まで)

集合演算子-1

2018年12月17日 月曜日　　17:22

集合演算：SELECTにより抽出した結果表を1つのデータの集合ととらえ、その結果同士を足し合わせたり、共通部分を探したりといった様々な演算を行う仕組み。

UNION：和集合。2つのSELECT文をUNIONでつないで記述すると、それぞれの検索結果を足し合わせた結果(和集合)を返す。
e.g. SELECT 文1 UNION(ALL) SELECT 文2 ← ALL付加で重複行をすべてそのまま返す

集合演算は、選択列リストに記述した列の組み合わせで計算される。
そぞれぞれの検索結果の列数が異なっていたり、データ型がバラバラだったりすると、DBMSは1つの結果表にまとめることができない。そのため、それぞれのテーブルの列数とデータ型を一致させておく必要がある。
列数とデータ型さえ一致していれば、全く異なるテーブルや列でも抽出可

1つのテーブルに格納されたデータを複数の異なる条件で抽出したい場合にもUNIONを活用できる。各々SELECT文を用意しておき、UNIONで1つのSQL文としてまとめることで、SQLの実行回数を抑えることが可能。

集合演算子でORDER BY を使うときの注意点
　ORDER BY は最後のSELECT文に記述する
　列番号以外による指定(列名やASによる別命)の場合、1つ目のSELECT文のものを指定

選択列リストの数が合わないSELECT文で、どうしても集合演算子を使いたい場合は、足りない方の選択列リストにNULLを追加することで、数を一致させることができる。

集合演算子-2

2018年12月17日 月曜日　17:37

EXCEPT：差集合。あるSELECT文の検索結果に存在する行からから別のSELECT文の検索結果に存在する行を差し引いた集合となる。oracleではMINUS
e.g. SELECT 文1 EXCEPT(ALL) SELECT 文2 ← ALLを付加で重複した行をそのまま返す

INTERSECT：積集合。2つのSELECT文に共通する行を集めた集合。
e.g. SELECT 列名 FROM table INTERSECT (ALL) SELECT 列名 FROM table

DBMSでは、ORDER BYやDISTINCT、UNIONなどには大量のメモリを消費する可能性があり、負荷のかかる作業なので、乱用は控える。

式と演算子

2018年12月17日 月曜日　　11:15

選択列リスト：SELECTのすぐ後ろに指定する。固定値や計算式の指定も可能

e.g. SELECT hoge + 100 AS 別名 FROM table

　　UPDATE table SET hoge ＝ hoge ＋ 100　　hogeが複数の場合は各々に適応

演算子は数値同士に使用した場合は四則演算と同じ結果になる。

日付に用いた場合は日数が増減する。

||は文字列同士を連結する

CASE演算子：列の値や条件式を評価し、結果に応じて好きな値に変換できる。

　　CASEからENDまでが一つの列としてカウント

e.g. CASE 評価する列や式 WHEN 値1 THEN 値1のときに返す値

　　・・・WHEN 値n THEN 値nの時に返す値　(ELSE デフォルト値)

　　END AS 別名

下記は最初に一致したWHENが採用される。一致しない時はELSEが採用

　　CASE WHEN 条件1 THEN 条件1のときに返す値

　　・・・WHEN 条件n THEN 条件nの時に返す値　(ELSE デフォルト値)

　　END AS 別名

関数-1(文字列)

2018年12月18日 火曜日　　12:20

関数は、DBMSごとの違いが大きく、互換性が少ない。

ユーザー定義関数：必要な処理を記述して作成した関数をSQL文から利用可能

ストアドプロシージャ

実行する複数のSQL文をまとめ、プログラムとしてDBMSに保存してDBの外部から呼び出すもの。DBとAP間のやり取りを少なくし、ネットワークの負荷軽減が可能

LENGTH関数

引数：文字列が格納された列

戻り値：文字列の長さを表す数値

TRIM関数：左右から空白を除去した文字列(charで追加された空白を除去できる)
LTRIM関数：左側の空白を除去した文字列
RTRIM関数：右側の空白を除去した文字列

REPLACE(置換対象の文字列, 置換前の部分文字列, 置換後の部分文字列)

→ 置換処理された後の文字列が返る

SUBSTRING(文字列を表す列, 抽出の開始位置, 抽出する文字数) → 抽出された部分文字列

SUBSTR(文字列を表す列, 抽出の開始位置, 抽出する文字数) → 抽出された部分文字列

抽出の開始位置は1以上の数字を指定(0は不可)。DBMSで字数指定かバイト数指定か異なる

関数-2

2018年12月18日 火曜日　　12:37

ROUND(数値を表す列, 有効とする桁数) → 四捨五入した値
有効桁数の値が正だと小数部の桁数、負は整数部の桁数を表す　　e.g. -2だと10の位

TRUNC(数値を表す列, 有効とする桁数) → 切り捨てた値
有効桁数の値が正だと小数部の桁数、負は整数部の桁数を表す

POWER(数値を表す列, 何乗するかを指定する数値) → 指定した回数乗じた結果

CURRENT_DATE → 現在の日付(YYYY-MM-DD)
CURRENT_TIME → 現在の時刻(HH:MM:SS)
上記2つとも引数不要なため()はつけない。

CAST(変換する値 AS 変換する型) → 変換後の値
e.g. CAST (hoge AS VARCHAR(20))　hogeをVARCHARに変換
数値として解釈できない文字列をINTEGERに変換しようとするとエラー

COALESCE(式や列1, 列や式2, ・・・, 列や式n)
→引数のうち、最初に現れたNULL出ない引数。引数は任意の数を指定可。
引数の型は全て一致させる。全ての引数がNULLだと戻り値はNULL
e.g. SELECT COALESCE(メモ, 'メモはNULL') AS メモ FROM table
上記はメモを表示し、メモがNULLだと'メモはNULL'に代替する

関数の動作確認のためSELECT文のFROM句を省略可能。oracleは不可

集計関数(グループ化)

2018年12月17日 月曜日　　11:15

・検索対象の全行をひとまとめに扱い、1回だけ集計処理を行う。
・集計関数の結果は、必ず1行になる。
・集計関数はSELECT文の選択列リストかHAVING句のみに記述可。WHEREには不可

SUM：各行の値の合計を求める。
MAX：各行の値の最大値を求める。文字列と日付も引数で指定可能
MIN：各行の値の最小値を求める。文字列と日付も引数で指定可能
AVG：各行の値の平均値を求める
NULLは無視。全行がNULLだと集計結果はNULL

計数

COUNT：行数をカウントする。　e.g. COUNT(DISTINCT hoge)
COUNT(*)は単純に行数をカウント、COUNT(列)は指定列の値がNULLである行を無視してカウント。
DISTINCTを指定することによって、その列で重複している値を除いた状態で集計を行う

グループ化：集計に先立って、指定した基準で検索結果をいくつかのまとまりに分ける

SELECT グループ化の基準列名, 集計関数
FROM テーブル名
(WHERE 絞り込み条件)
GROUP BY グループ化の基準列名

グループ化した集計関数のポイント

グループ化するにはGROUP BY 句に基準となる列を指定する
集計関数はデータの値をグループごとにまとめて計算する
集計関数の結果表の行数は、必ずグループの数と一致する
GROUP BY句にカンマ区切りで複数の列のを基準にしたグループ化が可能

HAVING：集計処理を行った後の結果表に対して絞り込みを行う
WHERE句を処理する段階では、集計が未完了なため集計関数はWHERE句に使用不可

SELECT グループ化の基準列名, 集計関数
FROM テーブル名
(WHERE もとの表に対する絞り込み条件)

GROUP BY グループ化の基準列名

HAVING 集計結果に対する絞り込み条件

SELECT文のグループ化の基準列名とGROUP BYのグループ化の基準列名は一致する必要がある。

集計テーブル

2018年12月18日 火曜日　　17:40

集計テーブルの利用

あるテーブルの集計結果を格納するための別テーブル(集計テーブル)を作成
集計関数を用いて集計処理を一回行い、結果を集計テーブルにINSERTする
集計結果が必要な場合は、すでに作った集計テーブルに格納されている計算済みの集計結果を利用する。
集計テーブルの内容は、最新の集計より古くなる可能性があるので、一定のタイミングで再度集計処理を実行し、最新の状態に更新することが不可欠

SELECT文で指定できるもの

SELECT 選択列リスト
FROM テーブル名
WHERE 条件式
GROUP BY グループ化の基準列名
HAVING 集計結果に対する絞り込み条件式
ORDER BY 並べ替え列名

副問い合わせ-1

2018年12月17日 月曜日　　11:15

副問い合わせ：他のSQL文の一部分として登場するSELECT文。()で括って記述
e.g. SELECT hoge FROM table WHERE foo = (SELECT MAX(hoge) FROM table)

データ構造：1つ以上のデータで形成されたもの
スカラ(単一の値)：昨日の京都の最高気温、自分の誕生日など
ベクター(配列・1次元に並んだ値)：過去12ヶ月の京都の最高気温、太陽系の惑星の名前など
マトリックス(表・2次元に並んだ値)：過去12ヶ月の各地の最高気温、九九の計算結果など

単一行副問い合わせ
検索結果が1行となる副問い合わせを指す。SELECT文の選択列リストやFROM、UPDATEのSET句、また1つの値との判定を行うWHERE句の条件式などに記述可能

```
e.g. UPDATE 家計簿集計
        SET 平均 =(SELECT AVG(出金額) FROM table WHERE 出金額 ＞0 )
     WHERE 費目 = '食費'

    SELECT 日付, メモ, (SELECT 合計 FROM table WHERE 出金額 ＞0 )
      FROM
    WHERE 費目 = '食費'
```

複数行副問い合わせ
検索結果が複数の行からなる単一列(n行1列を指す。実行の結果、複数の値が返ることもある。複数の値を列挙する場所で使用。IN、ANY、ALLなど。
WHERE句の条件式や、SELECT文のFROM句に記述可能

```
  e.g. SELECT  * FROM table
        WHERE 費目 IN (SELECT DISTINCT 費目 FROM table )

       SELECT  * FROM table
       WHERE 費目 = '食費'
          AND 出金額 < ANY(SELECT 出金額 FROM table WHERE 費目 = '食費' )
```

複数行副問い合わせでは単一の値の代わりに記述することはできない。
単なる等号や不等号では比較できない。ANY・ALL演算子を組み合わせることで比較が可能

NOT IN と ＜＞ALLは全ての値と一致しないことを判定する演算子

IN と ＝ANYはいずれかの値と一致することを判定する演算子

副問い合わせ-2

2018年12月19日 水曜日　　11:18

表形式となる副問い合わせ

検索結果がn行m列の表となる副問い合わせ(n, mは1以上)

SELECT文のFROM句やINSERT文などに記述することができる。

```
SELECT SUM(SUB, 出金額) AS 出金額合計
  FROM (SELECT 日付, 費目, 出金額
          FROM 家計簿
        UNION
        SELECT 日付, 費目, 出金額
          FROM 家計簿アーカイブ
         WHERE 日付 >= '2013-01-01'
           AND 日付 <= '2013-01-31') AS SUB
```

```
INSERT INTO 家計簿集計(費目, 合計, 平均, 回数)
SELECT 費目, SUM(出金額), AVG(出金額), 0
  FROM 家計簿
 WHERE 出金額 > 0
GROUP BY 費目
```

INSERTに副問い合わせを使用するとVALUES以降の記述に相当する

相関副問い合わせ：副問い合わせの内部から主問い合わせの表や列を利用する.
EXISTS演算子とともに使用する。

```
e.g. SELECT 列 FROM テーブル1
      WHERE EXISTS
     (SELECT * FROM テーブル2 WHERE テーブル1.列 = テーブル2.列)
```

副問い合わせ(NULLの対応)

2018年12月19日 水曜日　　11:09

NOT IN または ＜＞ALLで判定する副問い合わせの結果にNULLが含まれると、全体の結果もNULLになる。INではNULLが含まれていても等しい値が1つあれば結果を得られる。

副問い合わせの結果から確実にNULLを除外する方法

副問い合わせの絞り込み条件に、IS NOT NULL 条件を含める

COALESCE関数を使ってNULLを別の値に置き換える

e.g. SELECT * FROM table

WHERE 費目 IN (SELECT DISTINCT 費目 FROM table WHERE 費目 IS NOT NULL)

SELECT * FROM table

WHERE 費目 IN (SELECT COALESCE(費目, '不明)' FROM table)

行値式：複数の列の組み合わせによる条件式。oracleDBなど使用可能

e.g. WHERE(A, B) IN (SELECT C, D FROM ～)

表の結合

2018年12月17日 月曜日　　11:16

リレーション：2つのテーブルの行に情報としての関係がある
外部キー：他テーブルの関連行を示す値を格納し、リレーションを結ぶ役割を担う列

テーブルの結合

SELECT 選択肢リスト ← 両テーブルの列を指定可能
  FROM テーブルA
    JOIN テーブルB
      ON 両テーブルの結合条件

e.g. SELECT 日付, 名前 AS 費目, メモ
        FROM 家計簿
          JOIN 費目　　← 結合する他の表を指定
            ON 家計簿.費目ID = 費目.ID　←結合条件を指定

結合に関係する2つのテーブルは対等な関係ではない。
FROM句で指定したテーブル(左表)が主役であり、それにJOIN句で指定したテーブル(右表)の内容を必要に応じてつないでいく。
結合はテーブルを全て繋ぐのではなく、結合条件が満たされた行を1つひとつ繋ぐこと

右表の結合条件の重複
　繋ぐべき右表の行が複数あるとき、DBMSは左表の行を複製して結合する。
　結果表の行数は、もとの左表の行数より増える。

結合相手のいない結合
　右表に結合相手の行がない場合や、左表の結合条件の列がNULLの場合、
　結合結果から消滅し、結果表に現れることはない。

外部結合-1

2018年12月19日 水曜日　　12:59

SELECT ～ FROM 左表の名前
　　LEFT JOIN 右表の名前　← LEFT OUTER JOINと記述することも可能
　　ON 結合条件

結合相手の行がない場合や左表の結合条件がNULLの場合、選択列リストに
抽出される右表の列はすべてNULLとなる。結果的にすべての値がNULLである行を
新たに生み出して結合するので左表の全行を必ず出力する。

右外部結合：右表の全行を必ず出力する
完全外部結合：左右表の全行を必ず出力する

SELECT ～ FROM 左テーブル名
　　RIGHT[FULL] JOIN 右テーブル名　← RIGHT[FULL] OUTER JOINも記述可能
　　ON 結合条件

左外部結合、右外部結合、完全外部結合はいずれも本来結果表から消滅してしまう行も
強制的に出力する効果があり、これらを総称して外部結合という。
対して、結合すべき相手の行が見つからない場合に行が消滅する結合は内部結合という。

結合する表同士で同じ列名が存在する場合は、「テーブル名.」という表記を加え、
どのテーブルに属する列であるかを明示的に指定することができる。

外部結合-2

2018年12月19日 水曜日　　14:48

3つ以上のテーブルの結合も可能。一度に全てのテーブルが結合されるわけではなく、前から順に1つずつ結合処理が行われる。

```
e.g. SELECT 日付, 費目.名前  経費区分.名称
       FROM 家計簿
        JOIN 費目                          ← まずは費目を結合
         ON 家計簿.費目ID = 費目.ID    ←結合条件を指定
        JOIN 経費区分                      ←  その結果にさらに経費区分を結合
         ON 費目.経費区分ID = 経費区分.ID
```

副問い合わせの結果との結合

選択列リストや結合条件の指定のために、副問い合わせに別名を指定することが必要

```
e.g. SELECT 日付, 費目.名前  費目.経費区分ID
       FROM 家計簿
        JOIN (SELECT * FROM 費目
              WHERE 経費区分ID = 1
        )AS 費目                           ←副問い合わせの結果を結合
         ON 家計簿.費目ID = 費目.ID
```

同じテーブル同士を結合

同一テーブル同士を結合することを自己結合や再帰結合という。

選択列リストや条件式を記述するために、同じテーブルに別の名前をつける

```
e.g. SELECT A.日付, A.メモ, A.関連日付, B.メモ
       FROM 家計簿 AS A
      LEFT  JOIN 家計簿 AS B
            ON A.関連日付 = B.日付
```

非等価結合：結合条件は＝以外の演算子を用いた条件式も記述可能。DBMSの負荷は増大

トランザクション

2018年12月17日 月曜日　　11:16

トランザクション：1つ以上のSQL文をひとかたまりとしてあつかったもの
　　一部だけ実行は不可の途中分割不可能なものとして取り扱う
トランザクション制御：DBMSがトランザクションを扱うこと

DBMSは、トランザクションに含まれるすべてのSQL文について、必ず「全ての実行が完了している」か、「1つも実行されていない」のどちらかになるように制御する

コミット：トランザクション終了の際に仮の書き換えをすべて確定したことにする。
　　トランザクション中のSQL文では仮のもとして書き換えられる。
ロールバック：トランザクション中に異常発生のため中断した場合、すべての仮の
　　書き換えをキャンセルしてなかったことにする動作のこと。

BEGIN：開始の指示。この指示以降のSQL文を1つのトランザクションとする
COMMIT：終了の指示。この指示までを1つのトランザクションとし、変更を確定する
ROLLBACK：終了指示。ここまでを1つのトランザクションとし、変更の取り消しをする

BEGINとCOMMITの間で障害が発生した場合は、自動的にロールバックが行われ、
実行が取り消される。BEGINの前に戻る。

自動コミットモード：DBMSは1つのSQL文が実行されるたびに、自動的に裏でコミット
　　を実行する。ツールや環境により明示的に解除する必要がある。
MySQLでは SET AUTOCOMMIT = 0で解除

2フェーズコミット：各データベースに対して、トランザクションの確定準備と確定の
　　2段階指示を出し、トランザクションの整合性を維持する。

トランザクションの分離

2018年12月19日 水曜日　16:32

DBMSに対して複数の利用者が同時に処理を要求すると発生する副作用は次の3つある。

ダーティリード：まだコミットされていない未確定の変更を他人が読めてしまう
反復不能読み取り：SELECT文実行後に他人がデータを変更し、次回結果が異なること
　テーブルを複数回読み取る処理で整合性が崩れてしまう
ファントムリード：SELECT文実行後に他人がINSERTを実行し、結果が変わること
　1回目の検索結果の行数に依存する処理で問題が起こることがある

DBMSは上記問題を解決するために、トランザクション実行時に他のトランザクションから影響を受けないように分離して実行する。この制御を行うために内部でロックとよばれる仕組みを使う。
トランザクション実行中に読み書きしている行に鍵をかけ、他の人からの介入を防ぐ
ロック中は相手のトランザクションが完了するまで待たされることになる

どの程度厳密にトランザクションを分離するかをトランザクション分離レベルとして指定可能。多くのDBMSではREAD COMMITTEDがデフォルト

分離レベル	ダーティリード	反復不能読み取り	ファントムリード
READ UNCOMMITTED	恐れあり	恐れあり	恐れあり
READ COMMITTED	発生しない	恐れあり	恐れあり
REPEATABLE READ	発生しない	発生しない	恐れあり
SERIALIZABLE	発生しない	発生しない	発生しない

上に行くほど高速だが危険性は上昇する。以下は分離レベルの指定方法
SET TRANSACTION ISOLATION LEVEL 分離レベル名
SET CURRENT ISOLATION 分離レベル名

ロックの活用

2018年12月19日 水曜日　　16:51

明示的に取得できるロックの種類
行ロック：ある特定の1行だけをロックする
表ロック：ある特定のテーブル全体をロックする
データベースロック：データベース全体をロックする。
DBMSによっては、ページや表スペースなどもロック対象になる

排他ロック：他からのロックを一切許可しないため、主にデータの更新時に利用
共有ロック：ほかからの共有ロックを許す特性があり、データの読み取り時に利用

SELECT ～ FOR UPDATE (NOWAIT)：明示的な行ロックの取得

明治的なロックを取得しようとした時、他のトランザクションによって同じ行がロックされている場合は、ロックが解除されるまでトランザクションは待機状態になる。
NOWAITオプションを指定するとロックの解除を待たずにすぐさまロック失敗のエラーを返す。かけたロックは、コミットまたはロールバックによってトランザクションが終了されると解除される。

LOCK TABLE テーブル名 IN モード名 MODE(NOWAIT)：特定の表全体をロック
モード名をEXCLUSIVEで排他ロック、SHAREで共有ロックとなる。
取得された表ロックは、行ロックと同様にトランザクション終了に伴い解除される
SELECT文などの実行前に記述

デッドロック：お互いのトランザクションが相手のロック解除待ちになり処理が停止
　　対処法としてDBMSは片方のトランザクションを強制的に失敗させる

デッドロック予防として、トランザクションの時間を短くしたり、同じ順番でロックするようにする。

ロックエスカレーション
　多数の行ロックがかけられると自動的に表ロックに切り替わる

SQL命令の種類

2018年12月17日 月曜日　　11:16

データ操作言語(DML：Data Manioulation Language)

データの格納や取り出し、削除などを行うための命令

e.g. SELECT, INSERT, UPDATE, DELETE

データ定義言語(DDL：Data Definition Language)

データを格納するテーブルなどの作成や削除、各種設定を行うための命令

e.g. CREATE TABLE, ALTER TABLE, DROP TABLE

データ制御言語(DCL：Data Control Language)

DMLやDDLの利用に関する許可や禁止を設定するための命令

e.g. GRANT, REVOKE

GRANT 権限名 TO ユーザー名　　←権限付与

REVOKE 権限名 FROM ユーザー名　←権限剥奪

データベース管理者(DBA)が行う

テーブルの作成

2018年12月19日 水曜日　　17:50

テーブルの作成(基本形)

```
CREATE TABLE テーブル名(
    列1 列1の型名
    列2 列2の型名
        |
    列n 列nの型名
)
```

デフォルト値を含むテーブルの作成：NULLの代わりに指定したデフォルト値が格納される

```
CREATE TABLE テーブル名(
    列名 型名 DEFAULT デフォルト値,
)
```

テーブルの削除：DROP TABLE テーブル名

DMLに属するDELETEなどはロールバックによりキャンセルできるが、DDLに関してロールバックできるかどうかはDBMSにより異なる。oracleではDDLはロールバック不可

テーブル定義の内容を変更するには、ALTER TABLEを使用

列の追加(挿入される位置は原則として一番最後になる)

```
ALTER TABLE テーブル名 ADD 列名 型 制約
```

列の削除

```
ALTER TABLE テーブル名 DROP 列名 型 制約
```

全件のデータを高速に削除する

テーブルの全行を削除する場合に、TRUNCATE TABLEが利用されることがある

DELETEとほぼ同じだが、動作に次の違いがある。

DELETEはWHERE句で指定した行だけ削除できるが、TRUNCATEは必ず全行削除

DELETEはDMLだが、TRUNCATEはDDLに属する

DELETEはロールバックに備え、記録を残しながら借り削除するが、TRUNCATEは記録を残さずに行を削除する(ロールバック不可)

DELETEは記録を残すため低速だが、TRUNCATEは高速

TRUNCATE TABLE は厳密にはデータ削除ではなくテーブルの初期化の命令

テーブルを一度DROPして同じものをCREATEする動作イメージ

制約

2018年12月19日 水曜日　　18:06

多くのDBMSは制約を用いて、型よりもさらに強力な制限をかけることができる

NOT NULL制約

NULLの格納は許可されない。DEFAULT指定と組み合わせて利用されることが多い
INSERTで値を指定しないで実行するとエラーが発生

UNIQUE制約

列の内容が決して重複してはならない。NULLが格納された行が複数存在することは許可されている。NULLはNULLとも等しくないと解釈される。

CHECK制約

ある列に格納されている値が妥当であるかを細く判定する。
CHECKの後ろ括弧内に記述した条件式が真となるような値のみを許可

主キー制約

主キーの役割を担う列に設定する。この制約が付いている列は、NULLも重複も許されないだけでなく、インデックスとしても意味を持つ。

```
CREATE TABLE 家計簿 (
        ID  INTEGER  PRIMARY KEY,          ←単一の主キー
      日付  DATE  NOT NULL,
      メモ  VARCHAR(10)  DEFAULT  '不明'  NOT NULL,
  入金額  INTEGER  DEFAULT  0  CHECK(入金額 >= 0),
      名前  VARCHAR(40)  UNIQUE
      PRIMARY KEY(ID, 名前).        ←複合主キーも生成可能
)
```

外部キーと参照整合性

2018年12月20日 木曜日　　10:13

参照整合性：外部キーの参照先の行が存在していてリレーションが成立している状態
参照整合性の崩壊：外部キーの参照先の行がが存在していない。

外部キー制約
　　参照整合性が崩れるようなデータを操作しようとした場合にエラーを発生させ、
　　強制的に処理を中断させる制約

CREATE TABLE テーブル名(
　　列名 型 REFERENCES 参照先テーブル名(参照先列名) ← 宣言と同時に制約を設定

　　FOREIGN KEY (参照元列名) REFERENCES 参照先テーブル名(参照先列名)
　　↑テーブル定義の最後でまとめて制約を設定する場合
)

データベースの高速化

2018年12月17日 月曜日　　11:16

インデックス：データベースで作成することのできる検索情報

インデックスは、指定した列に対して作られる。
インデックスが存在する列に対して検索が行われた場合、DBMSは自動的にインデックスの使用を試みるため、高速になることが多い。
インデックスには名前をつけなければならない。重複しない範囲で自由に設定可能

CREATE INDEX インデックス名 ON テーブル名(列名)　　← インデックス作成(DDL)

DROP INDEX インデックス名 ON テーブル名　　← インデックス削除(MySQL)
DROP INDEX インデックス名　　← インデックス削除

高速化のパターン(WHERE, ORDER BY, JOINで頻繁に使用する列に効果大)

1.WHERE句による絞り込み

完全一致検索(全く同じ値であるかを検索)では、インデックスが使用され、高速に検索結果を得ることが可能。部分一致(任意の部分検索)や後方一致(末尾の部分一致)検索ではインデックスを使用できない。

2.ORDER BY による並び替え

並び替えを高速に行えるようにする効果もインデックスは備えているので、ORDER BYの処理が早くなる。

3.JOINによる結合の条件

結合処理は内部で並び替えを行うので、インデックスのある列を使うと高速になる

インデックスのデメリット
検索情報を保存するために、ディスク容量を消費する
データが変更されると、インデックスも書き換えるため、DMLのオーバーヘッドが増大
→検索性能は向上するが、書き換え時のオーバーヘッドは増加するので濫用しない

プラン：DBMSがどのような方法でアクセスすれば最も高速であるかを分析

EXPLAIN PLAN または EXPLAINを使用して、指定したSQL文を実行するプランを調べることが可能。インデックスを作成することによりSQLがどのくらい早くなるか目安を得たい場合にも有効。
e.g. EXPLAIN SELECT * FROM 家計簿 WHERE メモ = '不明'

データベースオブジェクト

データの管理や操作のために、データベース内に作成するテーブルやビュー、インデックスやシーケンスなどの総称

ビュー

2018年12月20日 木曜日　11:06

ビュー：結果表をテーブルのように扱える。実体は名前をつけたSELECT文。
　　実行されるSQL文は、一見するよりも負荷の高い処理になる可能性があるので注意

CREATE VIEW ビュー名 AS SELECT文 ← ビューの作成
DROP VIEW ビュー名 ← ビューの削除

```
CREATE VIEW 家計簿4月 AS
SELECT * FROM 家計簿
WHERE 日付 >= '2013-04-01' AND 日付 <= '2013-04-30'
SELECT * FROM 家計簿4月;
SELECT DISTINCT 費目ID FROM 家計簿4月;
```

採番：追加する行に独自の番号を振るため、適切な番号を取得すること
採番テーブル：既に採用した番号や最後に採番した番号を記録した専用のテーブル

マリアライズド・ビュー
　SELECT文による検索結果をキャッシュしているテーブルのようなもの
　データの実体を持つためディスク容量は消費するが、テーブルを経由することなく
　データを直接参照できるため高速に動作。インデックスの作成も可能。

単純な連番で良い場合、一部のDBMSでは連番を管理する機能が提供されている
CREATE TABLE文で列を定義する際に指定するだけで、連番が振られるように設定可能

SQL Server	IDENTITY修飾を付与
MySQL	AUTO_INCREMENT修飾を付与
PostgreSQL	データ型にSERIAL型を利用
SQLite	AUTOINCREMENT修飾を付与

シーケンス

2018年12月20日 木曜日　　11:25

シーケンス．OracleDB DB2 SQL Server PostgreSQLで利用可能

常に採番した最新の値を記録しており、シーケンスに指示をすることで現在の値(最後に採番した値)や次の値(次に採番すべき値)を取り出すことができる。

シーケンスから値を取り出すと、その操作はすぐに確定し、トランザクションをトールバックしてもシーケンスの値は戻らない。

これは1つのシーケンスが複数のトランザクションからの利用を考慮したため。

CREATE[DROP] SEQUENCE 費目シーケンス;　← シーケンス作成[削除]

SELECT 費目シーケンス.CURRVAL FROM DUAL;　← 現在の値取得

SELECT 費目シーケンス.NEXTVAL FROM DUAL;　← 次の値取得 DUALはダミーテーブル

最大値を用いた採番は利用しない

ロックを用いない限り、複数の人に同じ番号が採番されてしまう

→ロックによるパフォーマンス低下の懸念

最後に採番した行の削除後に採番すると、同じ番号を再利用してしまう

→主キーとして利用する場合の不変性の崩壊

e.g. SELECT MAX(ID) + 1 AS 採番 FROM 費目

データベースの安全利用

2018年12月20日 木曜日　　11:39

ACID特性：データを正確かつ安全に取り扱うためのシステムが備えるべき4つの特性

原子性：処理が中断しても中度半端な状態にならない
一貫性：データの内容が矛盾した状態にならない
分離性：複数の処理を同時実行しても副作用がない
永続性：記録した情報は消滅せず、保持され続ける

オフラインバックアップ：データベースを停止してからバックアップを行う
オンラインバックアップ：DBMSを稼動しながら整合性のあるバックアップデータを取得

データベースの内容は低頻度(日次、週次、月次など)
ログファイルの内容は高頻度(数分～数時間ごとなど)

データベースのログは、REDOログ(アーカイブログ)またはトランザクションログと呼ばれ、内容は「それまでに実行した全てのSQL文」が記載。
このログファイルを高頻度でバックアップすると、データ消失時にも消失直前の時点までデータを復元することができる。

バックアップからのデータの復元方法

1. 最期に取得したデータベースのバックアップを復元する。
2. ログに記録されているSQL文のうち、最後のデータベースバックアップ以降に実行されたものを再実行する。

ロールバック(実行した処理の取り消し)
　データベースの利用中にSQLの実行失敗やデッドロックなどで多々発生する。
ロールフォワード(まだ実行されていない処理を実行する)
　障害復旧時に行われる処理であるため、滅多に発生しない。

システムとデータベース

2018年12月17日 月曜日　　11:16

データベース構築のINPUTとOUTPUT
INPUT：要件の一覧表(顧客から聴取したもの)
OUTPUT：一連のDDL文(実行すれば必要十分名テーブルが作成されるもの)

概念設計

管理すべき情報は何かを整理。データベースやシステムに関することは考えず、
要件に登場する情報だけを大まかに把握する。
扱うべき情報に何があるかを明確にする。
情報間で関連がある場合、どのような関係であるかも併せて整理する。

論理設計

概念設計で明らかになった各情報について、RDBを使う前提で構造を整理して
詳しく具現化する。どのようなテーブル・列を作るかまでを明らかにする。
型や制約など、付随的な部分については考えない。

物理設計

特定のDBMS製品を使う前提に立ち、論理設計で明らかになった各テーブルに
ついてその内容を詳しく具現化する。
全てのテーブルの全ての列、型、インデックス、制約、デフォルト値などを確定する

この物理設計に基づいて、CREATE TABLEなどを含む一連のDDL文を作成し、
最終的にデータベース内にテーブルを作成することができる。

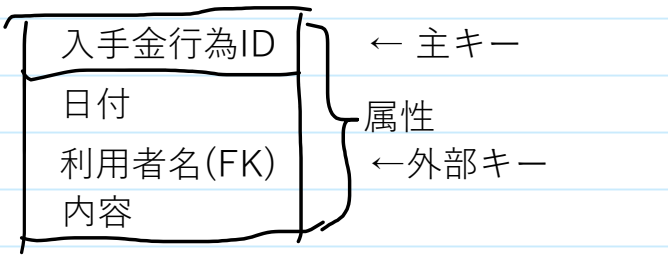
概念設計

2018年12月20日 木曜日　　12:31

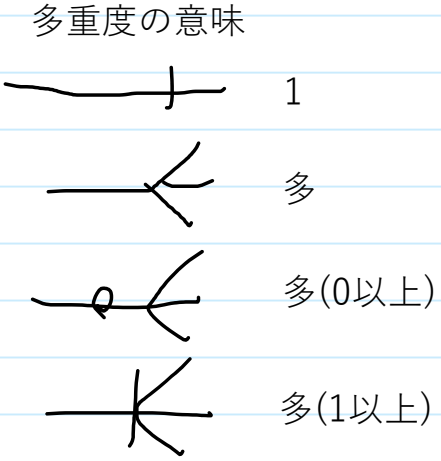
概念設計では要件を実現するために、抽象的な概念としてどのような情報の塊を管理しなければならないかを明らかにする。
この情報の塊をエンティティといい、通常エンティティは複数の属性を持っている。
形のあるものだけでなく、事実や行為など形のないものもエンティティになる。
エンティティ同士にどのような関係があるかも、この概念設計で明らかにする。

概念設計の成果は、ER図にまとめることが一般的。
ER図を用いることでエンティティ、属性、リレーションを俯瞰してみることが可能
ER図にはIEとIDEF1Xの2つの記述形式がある。

ER図記述のルール



多重度の意味



多重度・カーディナリティ：エンティティ同士の数量的な関係

1. 候補となる用語を洗い出す
   名詞を抜き出す、要件が実現されてる姿を仮定し、そこに登場する「人」「物」「事実」「行為」などの用語を書き出す。
2. 不要な用語を捨てる
   他の用語の具体例でしかないものを捨てる
   計算や集計をすれば算出可能な値は捨てる
3. 関連がありそうなものをまとめる
   同じ用語に関連するものを集める
   e.g. 日付、利用者、内容は何も入出金行為に関連する。
4. エンティティ名と属性名に分ける
   3.でまとめたグループの中で「〜の〜」という日本語が成立する場合、前者がエンティティ名に、後者がその属性名になる。

論理設計

2018年12月20日 木曜日　　14:16

概念設計で作成したER図はあくまでも概念の世界における理想的なエンティティ構造を表しているに過ぎないため、このままデータベースに格納できるとは限らない。
利用する予定のデータベースが扱いやすい構造にエンティティを変形する作業が論理設計
概念上のエンティティをリレーショナルデータモデルで扱いやすい形のテーブルに変形

多対多の分解

RDBは多対多の関係を上手く扱うことはできないので、2つのエンティティの対応を格納した中間テーブルを追加することにより、多対多を1対多の関係に変換する

主キーが備えるべき3つの特性

非NULL性：必ず何かしらの値を持っている。
一意性：他と重複しない。
不変性：一度決定したら値が変化することがない(主キーは一貫して同じ1行を指す)

正規化：矛盾したデータを格納できないように、テーブルを複数に分割していく作業
→テーブルを分割しないと内容に重複が多く、分かり辛い。
データ更新時には複数の関連箇所を正確に更新しなければならない。

整合性が崩れにくい優れたテーブルの設計の原則は「1つの事実は1箇所に」

正規化の手順(第1正規形)

2018年12月20日 木曜日　　14:35

正規形

正規化によってテーブルが適切に分割された状態。第1から第5正規形まで存在
通常のシステム開発が目的の場合は、第3正規形まで理解すれば問題はない

非正規形

簡単に整合性が崩れやすい。非正規形の構造はER図では表現できない。
特徴として「セルの結合を行なっている」「1つのセルに複数行書いている」など

第1正規形

テーブルの全ての行の全ての列に1つずつ値が入っていなければならないので、
繰り返しの列やセルの結合が現れてはならない。変形の手順は次の通り。

1. 繰り返しの列の部分を別の表に切り出す
   まず、元のテーブルから「繰り返しの列」の部分を別テーブルとして
   切り出し、切り出したテーブルに名前をつける。
2. 切り出したテーブルの仮の主キーを決める
3. 主キー列をコピーして複合主キーを構成する
   元のテーブルの主キー列を、切り出したテーブルにも加え、2.の仮の主キー
   とあわせて複合主キーを構成する。

関係従属性

ある列Aの値が決まれば、
自ずと列Bの値も決まるという関係
このとき、列Bは列Aに関係従属している
全ての非キー列は、主キーに
きれいに関係従属しているべき

繰り返し列とは1つの内容の隣で
同じ項目が繰り返し登場する列のこと

正規化の手順(第2正規形)

2018年12月20日 木曜日　　14:56

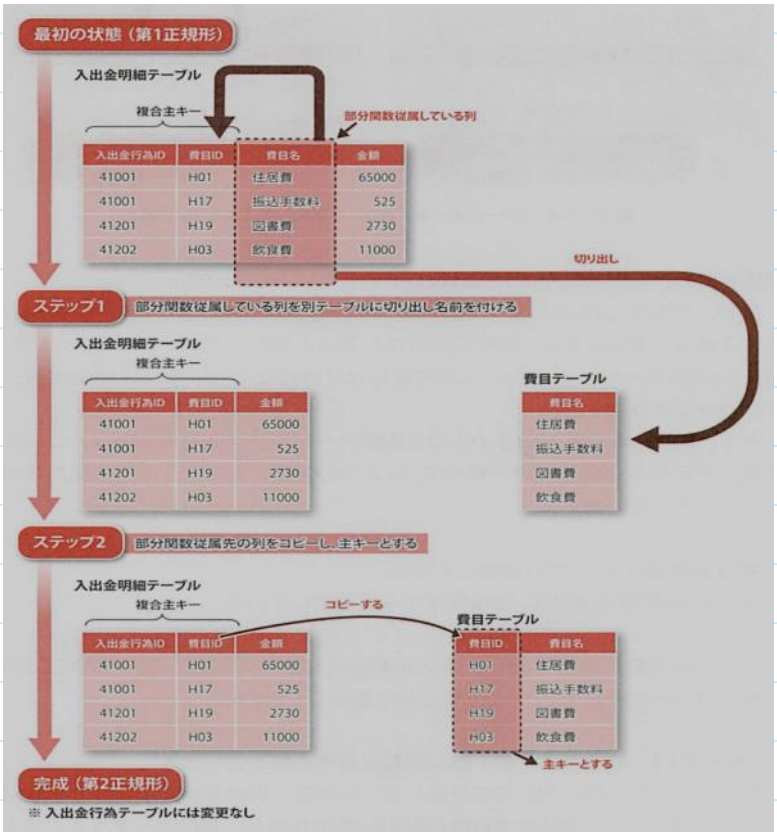
第2正規形への変形は、主キーに対する「きたない関係従属」の排除が目的
複合主キーを持つテーブルの場合、非キー列は複合主キーの全体に関数従属すべき
「複合主キーの一部の列に対してのみ関係従属する列」が含まれてはならない

部分関数従属

　　複合主キーの一部の列にしか関数従属しない状態
　　複合主キーをもたないテーブルは部分関数従属になる可能性はない

第2正規形では、全ての非キー列が複合主キーの全体への関数従属を求めている
変形の手順は次の通り

1. 複合主キーの一部に関数従属する列を切り出す
　　複合主キーの一部の列に関数従属している列を、別のテーブルとして
　　切り出して名前をつける。
2. 部分関数従属だった列をコピーする
　　切り出した列が関数従属していた列を、1.で作成したテーブルにコピーして
　　主キーとする。



正規化の手順(第3正規形)

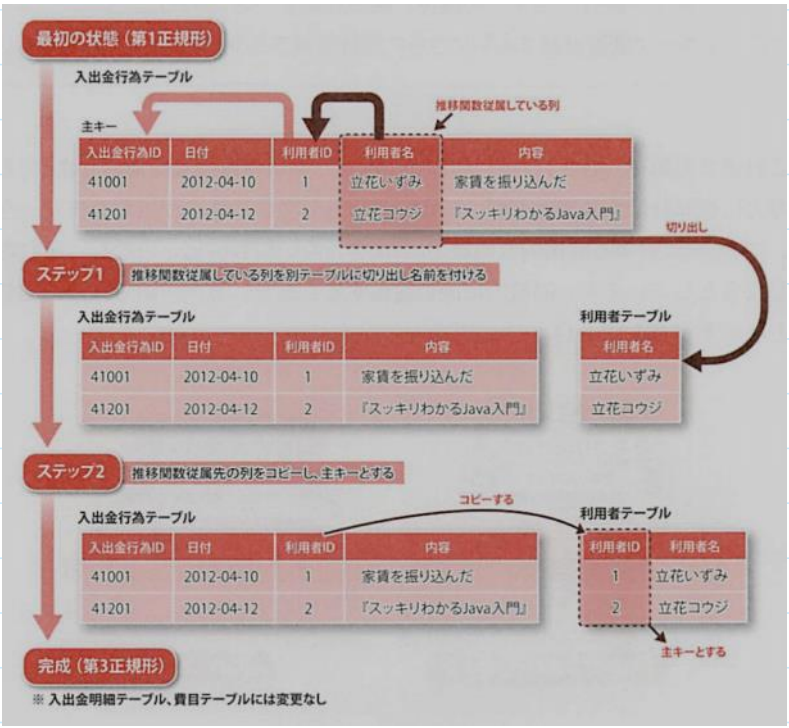
2018年12月20日 木曜日　　15:23

テーブルの非キー列は、主キーに直接、関係従属すべき。
主キーに関係従属する列にさらに関係従属する列は存在してはならない。

推移関数従属：間接的に関数従属すること

第三正規形では、推移関数従属を除去する。変形は次の通り。

1. 間接的に主キーに関数従属する列を切り出す
   間接的に主キーに関数従属している列を、別テーブルとして切り出し名前をつける
2. 直接関数従属先だった列をコピーする
   切り出した列が関数従属していた列を、切り出したテーブルにコピーして主キーとする。



正規化で排除するもの

第1正規形への変形：繰り返し列
第2正規形への変形：複合主キーの一部への関数従属(部分関数従属)
第3正規形への変形：間接的な関数従属(推移関数従属)

物理設計

2018年12月20日 木曜日　　15:45

論理設計後、どのDBMS製品を利用するかを確定した上で行うのが物理設計。
DBMS製品がサポートする型や制約、インデックス、利用するハードウェアなどの制約を考慮し、全テーブルについて詳細な設計を確定させる。
完成したデータモデルは、そのままDDLに変換できる内容になる。

物理名：最終的にDB内にテーブルが作られる際のテーブル名や列名
論理名：論理設計までの段階で利用してきた名前は論理名という

1. 最終的なテーブル名、列名(物理名)を決定する
    論理設計まではわかりやすいように日本語の使用が一般的だが、最終的にDB内にテーブルとして作成する場合は英語名をつけることも多い。

2. 列の型を決定する
    各列に対して指定する型を決定する。DBMS製品により型の種類や数値の精度が異なるため、マニュアルなどを参照しながら最適な型を選ぶ。

3. 制約、デフォルト値を決定する
    各テーブルや各列に対して、設定する制約を決定する。型と同じく、利用できる制約やデフォルト値はDBMS製品によって異なることがあるので物理設計の段階で決定する。

4. インデックスを決定する
    どの列にインデックスを設定するのかについても、物理設計で決定するDBMS製品のインデックス特性や、その列を利用する状況などを総合的に考慮して決定する。

5. その他
    利便性を考慮してレビューを作成したり、性能のためにあえて正規化を崩したり、巨大なテーブルを分割したりする作業が行われることがある。

正規化されたデータの利用

2018年12月20日 木曜日　　16:05