リレーショナル
データベース指向開発

リレーショナルデータベース指向

リレーショナルデータベース(RDB)とは

最初に実装されたのは、 1970年にIBMのSystem R だと言われています。同時にSQL言語も実装されました。

データの保存形式は、イメージしやすいように例えると、エクセルのシートが複数あり、同じ種類のデータは同じシートの同じ列に格納され、データ件数分だけ行が追加されてる。そして各くシート（テーブル）は、キーとなる列によって、他のテーブルの列との関係が定義されている。

詳細： https://www.ibm.com/jp-ja/topics/relational-databases

リレーショナルデータベース指向とは

リレーショナルデータベース(RDB) で主だったものは、Oracle、SQLServer、Db2、PostgreSQL、MySQLなどがあります。

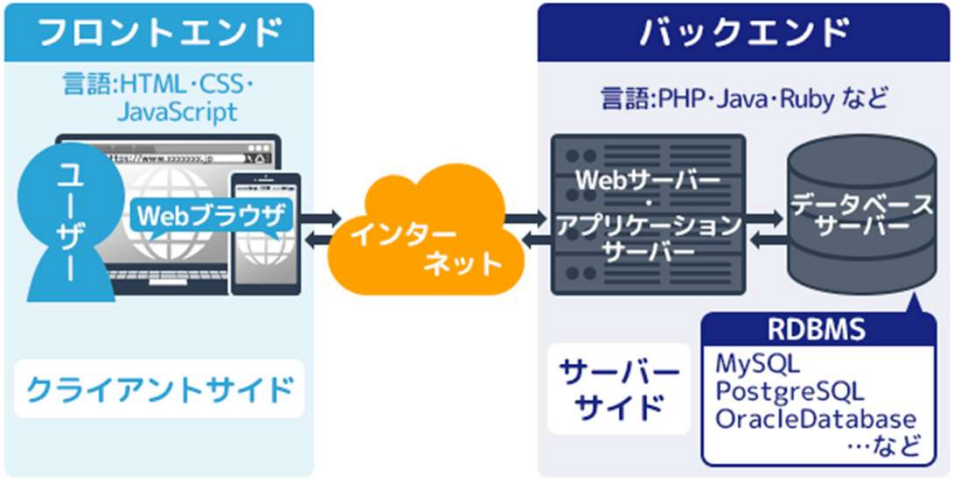
RDB指向(Oriented)とは、わたくしが命名した造語で、RDBの一般的な操作・管理言語であるSQL言語を中心として開発を行うことを言います。SQLにほぼすべての業務ロジックを埋め込む開発手法となります。

※略して、RDBOまたは、DBO

一般的なシステム開発とは

一般的なWebアプリケーション処理

右図のような、構成がWebシステムでの一般的な構成になります。その際の処理の流れの断片をとると以下のようになります。



1.ブラウザ上からユーザーがキーワードを指定して検索を実行する。

2. キーワード（パラメータ）がアプリケーション(AP)サーバーへ送付されて、パラメータを用いてSQL文を組み立てる。

3. APからRDBへSQLを問合せし結果をAPが取得する。

4. 問い合わせ結果をブラウザ側へ返す。

5.ブラウザ側で検索結果を表示する。

一般的なWebアプリケーション開発

大まかな開発担当

1.データベース設計開発。

　主に業務で扱うデータと、システム稼働する上で必要なデータを保存するためのテーブル設計開発

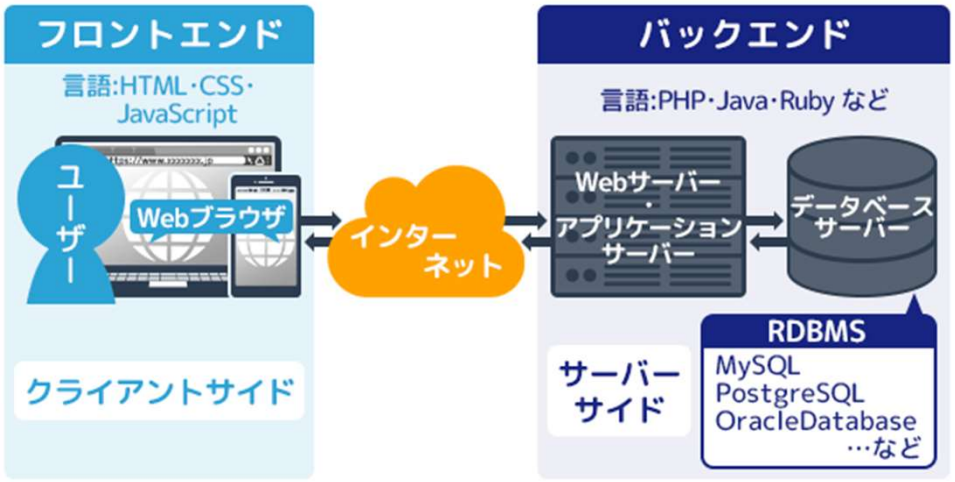
2. アプリケーション設計開発。

　主に、業務で扱うデータを加工処理する。

　RDBで取得したデータオブジェクトから求める形(フォーマット)のデータへ加工処理する。

3. クライアントサイド設計開発。

　APから取得したデータオブジェクトを画面へ出力するための形(フォーマット)へ加工処理する。



1. データベース設計開発

右ER図のようなテーブル、項目を設計

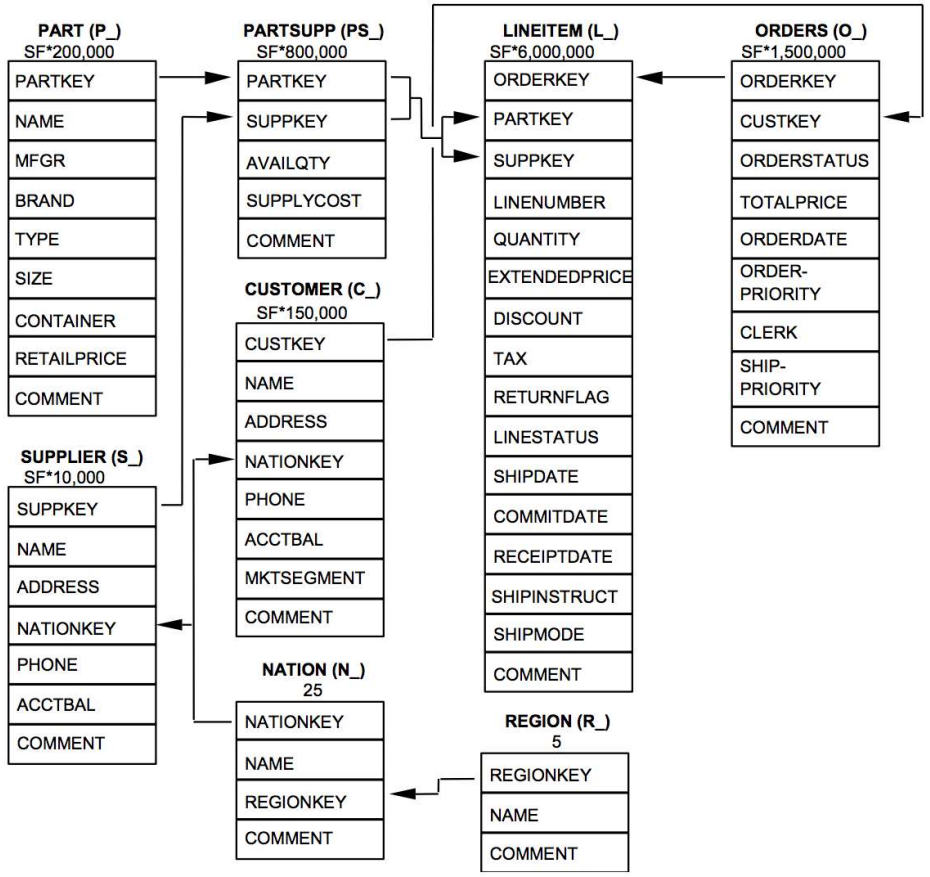
同じデータは同じテーブルの同じ列に保存される。列には型(整数、小数、文字列、時刻型、、、など)が存在する。

各テーブルには、テーブル同士の関係（※AテーブルのB列の値は、CテーブルのD列にあるいずれかの行と同じ、、、など)意味付けを定義する。

定義の情報は、設計書、あるいはDBの定義そのものだったりするが、多くの場合、AP側の実装に埋もれてしまっているのが実情だったりする。

SQLの設計書は数々のフォーマットなどが存在するが、複雑なSQLになるとSQLを文章で定義したりフォーマットに落とし込むのは不可能ではないかと思われる。

Figure 2: The TPC-H Schema



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

2. アプリケーション設計開発

クエリの発行、結果取得

右図のような、必要なデータを取得するための
SQL文を作成することを目的とする。結果はAP側
実装言語のオブジェクトとして取得する。

※C#オブジェクト DataSet

必要な分だけ上記の作業を繰り返す。

取得したすべてのオブジェクトを元に、ブラウザ
側へ渡す最終データ形式(※JSONなど)になるよう
なデータオブジェクトを作成処理。

```
SELECT
        L_RETURNFLAG,
        L_LINESTATUS,
        SUM(L_QUANTITY) AS SUM_QTY,
        SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE,
        SUM(L_EXTENDEDPRICE * (1-L_DISCOUNT)) AS SUM_DISC_PRICE,
        SUM(L_EXTENDEDPRICE * (1-L_DISCOUNT) * (1+L_TAX)) AS SUM_CHARGE,
        AVG(L_QUANTITY) AS AVG_QTY,
        AVG(L_EXTENDEDPRICE) AS AVG_PRICE,
        AVG(L_DISCOUNT) AS AVG_DISC,
        COUNT(*) AS COUNT_ORDER
FROM
        LINEITEM
WHERE
        L_SHIPDATE <= DATEADD(DAY, -90, TO_DATE('1998-12-01'))
GROUP BY
        L_RETURNFLAG,
        L_LINESTATUS
ORDER BY
        L_RETURNFLAG,
        L_LINESTATUS;
```

業務ロジックはここに埋め込まれる。

3. クライアントサイド設計開発

画面出力用データの取得

javascriptオブジェクトデータとしてAPからデータを取得して画面へ描画する。

※AP:C#側で作成したデータセットオブジェクトがjavascriptオブジェクトデータとなって取得される。

※データのみでなくAP側でHTMLタグ等を含めて画面出力用データを作成することもある。（※昔のwebでは一般的）

JSON

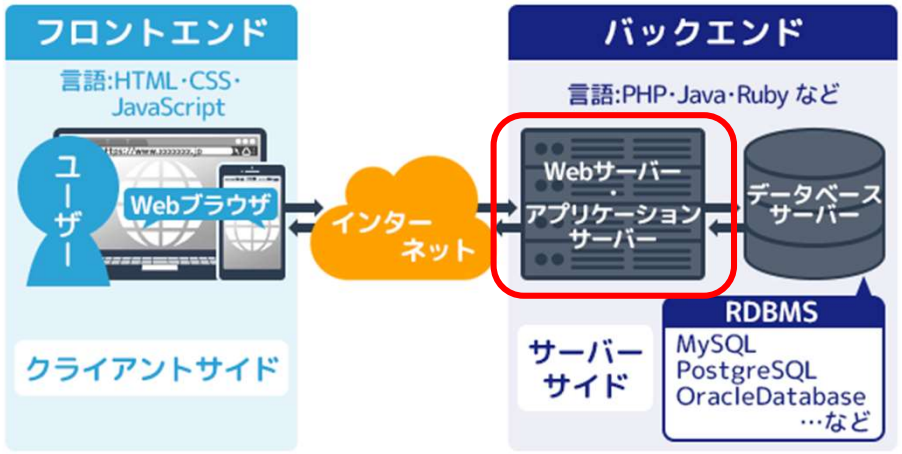
```
[
  {
    "name": "Molecule Man",
    "age": 29,
    "secretIdentity": "Dan Jukes",
    "powers": ["Radiation resistance", "Turning tiny", "Radiation blast"]
  },
  {
    "name": "Madame Uppercut",
    "age": 39,
    "secretIdentity": "Jane Wilson",
    "powers": [
      "Million tonne punch",
      "Damage resistance",
      "Superhuman reflexes"
    ]
  }
]
```

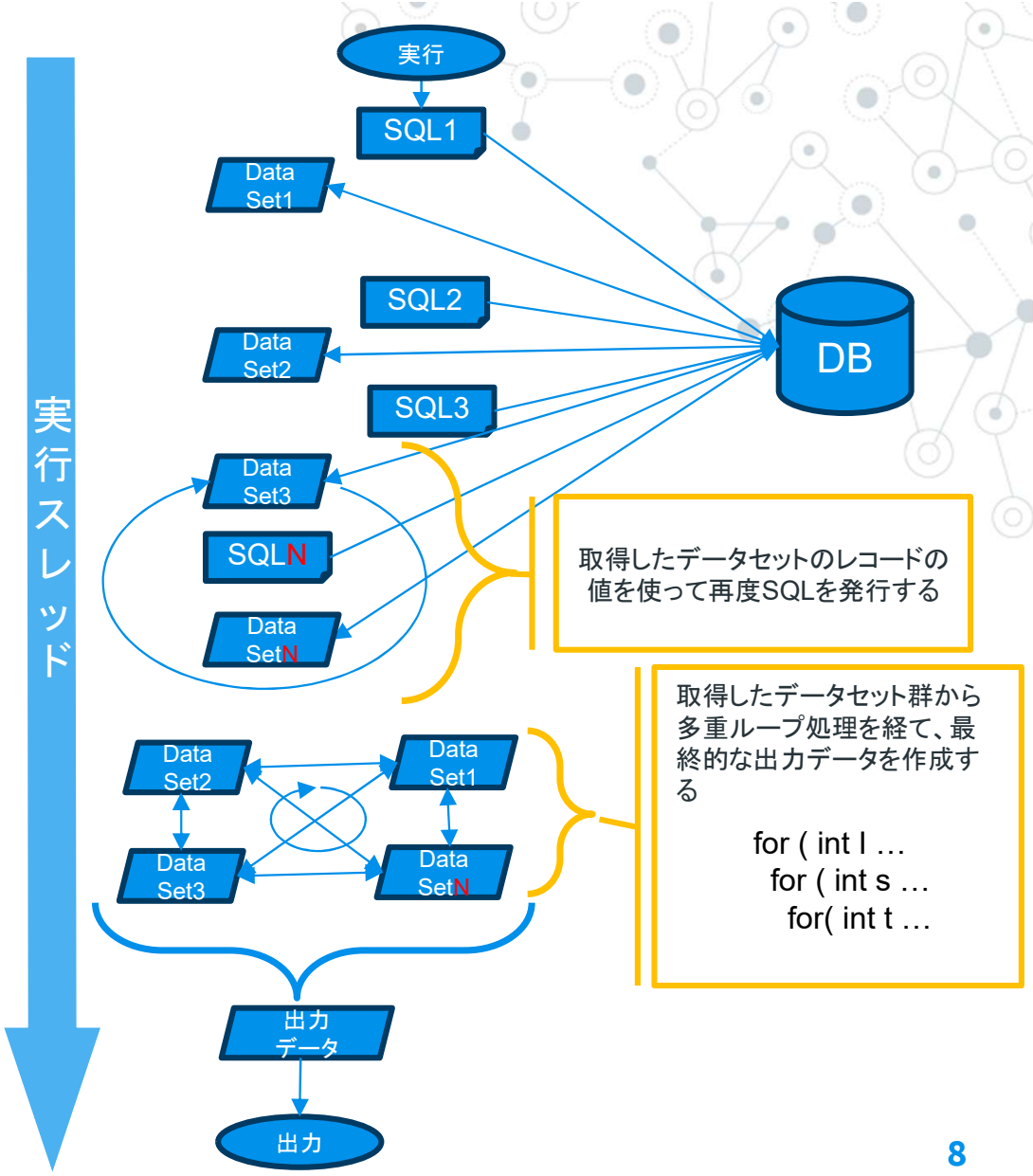
業務ロジックの実装

AP側でロジックを実装

開発において最も工数を割くのがAP側でのロジック設計開発です。右図は、AP側でのデータを取得し、クライアントへ返す為の出力データを作成するまでの流れです。

Webシステム上のAP側でのイベント処理は一般的に1スレッドで行われていることが多いと思います。右図ですと、SQLをDBへ投げて結果が返ってくるまでスレッドは待機状態になります。複数のSQLをDBへ投げる場合はその都度結果取得まで待機させるのが、もっともポピュラーな作りになります。





制御フロー1

C言語,Java言語,C#言語,JS言語,SQL言語

最初にプログラムを学習する際、制御フローを学ぶ。

順次処理　・・・書いた順に実行

分岐処理　・・・指定条件に従って実行順を変える

ループ処理・・・同じ処理を繰り返す

この3つを組み合わせれば
なんでもできる！ハズ、、、

```
printf(“Hello World!¥n”);
printf(“おはよう! ¥n”);
printf(“こんにちは！¥n”);
printf(“こんばんは! ¥n”);
```

```
Hello World!
おはよう
こんにちは！
こんばんは！
```

順番に出力される。

```
if(x==1){
   printf(“OK! ¥n”);
}else{
  printf(“NG! ¥n”);
}
```

```
//X=1の場合
OK!
//それ以外
NG!
```

条件によって出力が分岐される。

```
for (i = 1; i <= 3; i = i + 1){
  printf("こんにちは!¥n");
}
```

```
こんにちは!
こんにちは!
こんにちは!
```

指定回数繰り返す。

制御フロー2

SQL言語の共通認識

SQL言語を学ぶときに最初に行うことは、制御フローではなく、SELECT, INSERT ,UPDATE,DELETE,CREATE 所謂CRUDと言われているものである。そして、おそらく、学習過程に置いて、他のプログラム言語とは異なり、制御フローから学習することはないと思います。また、その後の学習の過程で制御フローは学習しないまま終えるのではないでしょうか？そのため、多くの技術者にとってSQL言語とは、RDBとのインターフェースに特化した特別な言語の為、制御フローがなくても特段疑問に思うことはないと思われます。

制御フローの学習がない！

```
Select * from table1 where …
```
データ取得

```
Insert Into table1 (col1,col2)
Values (data1,data2)
```
データ追加

```
Update table1
    set  col1 = data1
       ,  col2 = data2
```
データ更新

```
Delete from table1 where …
```
データ削除

```
Create table table1
          (col1 int ,
           col2 char
           . . .
           )
```
テーブル定義

制御フロー3

SQL言語で順次処理

順次処理とは任意のデータの順番を出力できること。

SQLで並び順を制御するのはOrder By句

ASC　昇順　1,2,3

DESC　降順　3,2,1

1,3,2 と並べるには・・・

大小だけでなく、任意の順番にレコードを出力できる。

```
Select * from table1
Order By col1 ASC
```

列col1 昇順

```
Select * from table1
Order By col1 DESC
```

列col1 降順

```
Select * from table1
Order By
Case col1
When 2 then 3
When 3 then 2
Else col1
End
```

列col1: 1,3,2の順

制御フロー4

SQL言語で分岐処理

分岐処理は順次処理のorder by句で使用したcase式になります。

Case式はwhere句でも使用可能です。

wehere句でcase式のサンプルとして、web画面から検索を行う場合、検索ボックス、選択項目で、入力、選択されたもののみを検索条件に含めてSQLを発行する際よく用います。

具体的には、SQL文を静的に作成する際使います。

多くの場合AP側でSQLを動的文字列として作成するが、CASE式によって静的に記述できる

```
Select
CASE    col1
WHEN 1 THEN 10
WHEN 2 THEN 20
ELSE id
END
 from table1
```

列col1の値により出力値を変える

```
SELECT
*
FROM emp
where
CASE
  WHEN @name <> “”
  THEN name =@name
  ELSE TRUE
END
```

氏名欄に入力があった場合、氏名で検索したい。
空白の場合は検索条件に含めない。

制御フロー5

SQL言語でループ処理

ループ処理は相関サブクエリを使って実現する。普通のサブクエリは、サブクエリだけを抽出して単独実行できます。一方相関サブクエリは、他のテーブルのカラムが含まれているため単独実行してもエラーになる。

サブクエリ内に他のテーブルのカラムが含まれるSQLを相関サブクエリという。

s1

s1_id	s1_name
10	a
20	b
30	c

t1

t1_id	t1_name
1	a
2	d
3	c

手続き言語のイメージ。

```
s1[3][2];
t1[3][2];
d1[3][3];

for(int i = 0; i < t1.lengh; i++){
 for(int s = 0; s < s1.lengh; s++){
      if(s1[s]["s1_name"] == t1[i]["t1_name"]){
        d1[i]["s1_id"]=s1[s]["s1_id"]
        break;
      }else{
       d1[i]["s1_id"]=null;
      }
    }
  d1[i]["t1_id"]=t1[i]["t1_id"];
  d1[i]["t1_name"]=t1[i]["t1_name"];
}
```

```
Select
 (select id from table1 limit 1)
,*
from table1
```

カッコ内サブクエリをのみで実行可能

```
Select
 (select id
    from table2 s1
  where s1.name = t1.name
    limit 1) s1_name
,*
From table1 t1
```

s1のnameとt1のnameが同じ場合s1のidを出力する。

d1

s1_id	t1_id	t1_name
10	1	a
null	2	d
30	3	c

制御フロー6

Select句に相関サブクエリを含むSQLの実行順

1. 親のFrom句が実行される。
2. 親のwhere句が実行される。
3. 親のselect句がサブクエリを除いて実行される。
4. サブクエリが親の行数分実行される。

親のレコード3行分3回各レコードのt1_nameの値をパラメータとしてサブクエリが実行される！

```
select id
  from table2 s1
 where s1.name = "a" /*t1.name*/
 limit 1

select id
  from table2 s1
 where s1.name = "d" /*t1.name*/
 limit 1

select id
  from table2 s1
 where s1.name = "c" /*t1.name*/
 limit 1
```

d1

s1_id	t1_id	t1_name
10	1	a
null	2	d
30	3	c

s1

s1_id	s1_name
10	a
20	b
30	c

t1

t1_id	t1_name
1	a
2	d
3	c

業務ロジックをSQLで実装するメリット1

SQL言語にも制御フローがある

以上の説明からSQLにも制御フローが存在し、若干記法が異なるが、他のプログラム言語同様であることを説明しました。AP側での業務ロジックの実装に制御フローを用いるように、SQLの制御フローでも同様の実装が可能となります。RDBのスキーマ、テーブル、カラム名は、それぞれ、業務の言葉で定義されています。SQL言語でも、他のプログラム言語同様制御フローを学習することによって、業務知識があれば、求めるデータを求める形式(フォーマット)で取得できるようになります。これは、情報システム部門であったり、または、ユーザー部門内だけで簡単に業務タスクが完結できてしまう可能性を意味します。

業務システムで実際に開発実装する際の言語、フレームワーク、環境等は、目まぐるしく流行り廃りを繰り返しますが、ここ、数十年の歴史を振り返っても業務データをRDBで管理し、そのインターフェースとしてSQLを用いるのは、ほぼ一貫して不変です。過去を振り返ると、RDBを置き換えるような新しい技術は、所々で流布されてきましたが、未だにRDBかつSQLという組み合わせは揺らぐ気配がありません。

SQL言語とその他の言語

・可読性がいい。ロジックを業務の言葉(スキーマ、テーブル、カラム名で書ける)

・他の言語と大きく違う点が宣言的な言語であることがSQLの特徴になります。※ループ処理などを見ても、明示的にループ処理を書くようなことはないため、可読性が著しく向上します。かつて、プログラム実装においてバグの大きな原因の一つにgoto文の存在がありました。その後もバグの少ないプログラムは、人間にとって可読性のいい文法であることが求められてきました。例えば、ループ処理やその中で使うカウンタ値を用いた中で制御フローが複雑に入り組んでくるとバグになりやすい傾向にあります。現代風のプログラム言語では、そのための文法が用意されてきているように思います。※JSを例にすると、
foreach,filter,map,flat,,,etc

例えばfilterを例にとると、

■filter

```
const result = ['spray', 'elite', 'destruction', 'present'].filter(
  word => word.length > 6,
);
```

これは、SQLで言う所のselect文でwhere句内で条件絞り込みしてるようにみえませんか？

今までの手続き言語の文法がSQL言語ライクになってきてます。

業務ロジックをSQLで実装するメリット2

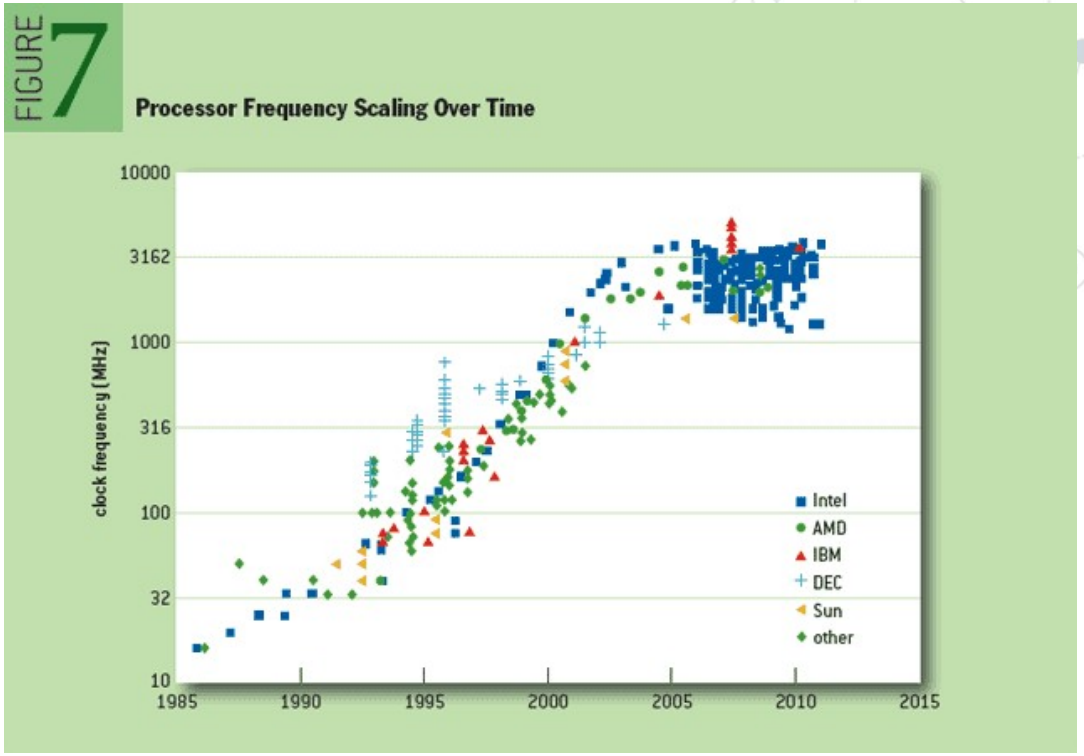
SQL言語とその他の言語2

・現代のCPUは、周波数の向上の伸びしろが少なくなり(*右図)、シングルでの高速化の進歩は、年率10%もいかないのではないでしょうか。そのため、およそ、ここ20年間のトレンドは、コア数の増加によるマルチコア化による性能向上が主だったもになっています。

・先にも一部触れましたが、イベントアクションからのAP側での処理は1スレッドで完結していることがほとんどだと思われます。CPUパッケージ全体としての性能向上に比べて、AP側の1イベントアクション(１スレッドでの処理)の性能向上は、CPUの性能向上に連動しずらくなっています。

・PostgreSQLでは１つのSQL処理をマルチコアで分散処理できるように進歩し続けています。これは、AP側の1イベント処理で例えるならば、１スレッド処理に対して、マルチスレッド処理の記述を一切行わなくても自動で最適なコア数で並列実行してくれることを意味します。(※VLIW)さらに、SQLに業務ロジックを埋め込むとは、AP側の１スレッドのイベント処理をマルチスレッド処理化するのと同様の効果を得られます。しかもそれは、マルチスレッド処理に伴う大きなデメリットである、スレッド管理を開発者は、一切意識する必要がありません。

SQLは勝手にマルチコアCPUを最大限活用して
並列処理してくれる。



https://queue.acm.org/detail.cfm?id=2181798

業務ロジックをSQLで実装するメリット3

SQL言語とその他の言語3

・SQLで業務ロジックを実装する上で最も重要なことは、制御フローがあることですが、そのほかにも重要な機能があります。

・CTE(共通テーブル式) with句。手続き言語のように上段から下段に向けて直行して処理の流れを記述できるようになります。

```
With c1 as ( select * from table1 … )
   , c2 as ( select * from c1 … )
   , c3 as ( select * from c2 … )
select * from c3 …;
```

with句がないと

```
select *
 from ( select *                                   ④
          from ( select *                          ③
                   from ( select *                 ②
                            from ( select * ①
                                     from table1
                                   ) as c1
                          ) as c2
                 ) as c3
)
```

①>②>③>④の順で実行される。

CTEがないと、著しく可読性が悪くなり、SQLで業務ロジックを実装するには難があったと思われます。

また、CTEの中で更新系処理Insert,Update,Delate なども行うことが可能です。

業務ロジックをSQLで実装するメリット4

SQL言語とその他の言語4

・select句内のサブクエリは1レコード1カラムの値しか返せません。

1レコード複数カラムを返す場合はカラム数分のサブクエリを書かねばなりません。

Lateral句を使うと1クエリで複数カラムを使うことが可能です。

```
select
 ( select cl1 from table2 limit 1)
,( select cl2 from table2 limit 1)
from table1
```

カラム数分サブクエリが必要

```
select
    t2.col1
  , t2.col2
from table1
left join lateral (select col1,col2 from table2 limit 1) as t2
    on true
```

サブクエリ１つで複数カラムを参照できる

※lateral句はmysqlでは8.0.14(2019年)からサポートされました。

ウィンドウ関数

・開発において、可読性、保守・改修性、統一性など意識して行いますが、SQLの開発も同様です。SQLで業務ロジックを実装すると、相関サブクエリだらけになります。特に１つのSQLに同一のテーブルの相関サブクエリが増える傾向があります。そのような場合は、冗長であることを避けるためにウィンドウ関数を用います。

```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

```
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

※depnameが同じ中でsalaryを昇順で順位付けするのがrankですが、ウィンドウ関数を使うことによって、empsalaryテーブルの出現はform句の1か所だけになっています。ウィンドウ関数を使わないとselect句内に再度empsalaryテーブルのサブクエリを実装する必要がでてきます。ほかにもウィンドウ関数は、多数存在します。

ウィンドウ関数の使用 ＝ サブクエリを減らす

業務ロジックをSQLで実装するメリット5

SQLをDBで一元管理

AP側でSQLを管理するよりもDB側で管理する方法としてストアドプロシージャがある。以下のメソッド内に静的、動的にSQLを記述できる。AP側からストアドプロシージャを呼び出すことでSQLを実行し結果取得が可能となる。

```
CREATE OR REPLACE PROCEDURE get_emp(IN pid INT, INOUT rc refcursor)
AS $$
BEGIN

    OPEN rc
     FOR SELECT *
           FROM emp
          WHERE id = pid;


END;
$$ LANGUAGE plpgsql;

BEGIN;
CALL get_emp(1,'rc');
FETCH all from rc;

COMMIT;
```

結果のデータセットを呼び出し側に返す場合は、基本的にカーソル参照を引数として渡す。

ストアドプロシージャ

SQLの外部で通常のプログラムのような制御フローを記述できる。

ストアド内でSQLの問合せ結果の加工処理を行う場合、カーソルを用いて結果セットを1レコードずつ処理を行う。

呼び出し元へ結果セットを返す場合、引数へカーソル(※ref cursor)の参照などを渡すことで呼び出し側へ返すことができる。

ストアドプロシージャを管理することを考える場合は、データベーススキーマ名単位でユニークな名称でなければならない。ストアドプロシージャを設計管理することを考えるとスキーマに階層構造を持たせることができないのがネックになる。

```
スキーマ1
     .proc1
     .proc2
スキーマ2
     .proc1
     .proc2( in param1 integer, in param2 varchar(10) )
     .proc2( in param1 varchar(10) , in param2 integer )
```

※オーバーロードはOK（※postgresql）

スキーマ名は、階層構造をもたせることができない。
大規模開発等を検討する場合は、命名規則を慎重に検討することが必要

ブラウザ内DB

DuckDB

・クライアントブラウザ上のメモリ内でwasmの軽量DBを使うことができる。サーバーDBのSQLで業務ロジックの実装のメリットを説明しましたが、クライアント側のブラウザ内でもDBを利用することによってさらに多くのメリットが得られます。

※クライント側のDBは画面遷移を行うと基本的に消滅するため、SPAであることが必須。

・サーバーDB側のクエリをシンプルにしてクラインDB側のSQLで代用することで、サーバーDB側のクエリをシンプルにできる。

・一般的には、サーバーDBからの結果セットをAP側セッションに保存し、クライント側でのページング処理をセッションから取得する。クライアントDBを使と、セッションに保存していたものをすべてクライアントDBへ保存することによってページング処理をクライアント内で完結することも可能である。データの流れだけを見ると、クライアントサーバー方式のインストールベースのシステムと同様になる。(※かつてのVBクライアントアプリケーションのような作りが可能になる。)

・具体的なアプリケーションとしてDuckDBがある。 高速かつ、求める機能が一通り揃っている。ベースはsqlite。

・sqliteもwasm化したものがあるがlateral句がない。

DuckDB
https://duckdb.org/

ブラウザ内でDBを使うメリット

・最大のメリットはやはり、本来サーバー側でしかできなかった処理がクライアントのSQLで代用することによってクライアント内JavaScriptで行っていた業務ロジックの実装を静的SQLで実装できることである。

・webアプリケーション全体を通して、よりRDB指向な作りになる。

・クライアント-サーバー通して業務ロジックがSQL実装になる。

リレーショナルデータベース指向のメリット

主なメリット

・業務データを取り扱うアプリケーションとしてRDBは圧倒的に主流でありそのI/FとしてのSQLも過去20年以上に渡り不動の地位にある。

・SQL言語はサブクエリ等が複雑にネストしたものでも、一般的なAP言語での実装に比べて、可読性がいい宣言型の言語である。

・ユーザー部門または、情シス部門内で開発を行いやすい。（※RDB内での業務データは、ユーザー部門または、その情報システム部門が知識として理解しているのが理想。そこをユーザー、情シス部門内で理解できていないと外部ベンダ頼みになってしまう。）

・SQL言語の制御フローを理解できれば、ほぼすべてのデータを欲しい形で取得できる。（※そしてそのSQL言語は、過去から今後にかけて他の技術などに置き換わる可能性が極めて少ない普遍的な言語である。）

・クライアントやAP側の実装言語やフレームワーク等は、その時々での流行り廃りに合わせて外部ベンダを使い倒し、基幹であるデータ定義、その操作するSQLが社内で完結できると、システム開発をより競争させることができる。(※ベンダ依存にならない。コスト削減に働く。)