

# データベース設計 SQL

講師：福田 剛志

[fukudat@acm.org](mailto:fukudat@acm.org)

<http://www.fukudat.com/>

## なぜ SQL か？

- SQL は「超」高級言語：  
プログラマは、(例えば)C++で必要とされるようなデータ操作の詳細のほとんどを、記述しなくても良い。
- SQL が使われている理由：  
実行がうまく「最適化」されていて、効率的に問合せが実行されるため。

## Select-From-Where 文

- 基本的な問合せ形式:

**SELECT** <属性のリスト>

**FROM** <一つ以上のテーブル>

**WHERE** <テーブルのタプルに関する条件>;

## 例に使うスキーマ

- この後の SQL 問合せは、以下のデータベーススキーマを用いることにする:

酒(酒名, 製造元)

呑み屋(呑み屋名, 住所, 定休日)

酒呑み(酒呑み名, 住所, 電話番号)

好き(酒呑み名, 酒名)

販売(呑み屋名, 酒名, 値段)

行きつけ(酒呑み名, 呑み屋名)

例えば...

- 酒(酒名, 製造元) を使って, 朝日酒造によって作られている酒の名前を求めるには?

```
SELECT 酒名  
FROM 酒  
WHERE 製造元='朝日酒造';
```

## その結果

酒名
‘久保田千寿’
‘久保田万寿’
‘久保田碧寿’

結果は属性「酒名」が一つあるリレーションで、  
各タプルは「朝日酒造」が作っている酒の名前

## 単一リレーションに対する問合せの意味

- FROM 節にあるリレーションから,
- WHERE 節にある条件を適用して,
- SELECT 節にある属性リストに射影する.

## 手続き的な意味

- FROM節に書かれたリレーシヨンのタプル一つ一つを値として取る変数を考える.
- その一つ一つの代入に対して, WHERE節を満たしているかどうか確かめる.
- もしそうなら, 注目しているタプルの値を用いて, SELECT節にある属性または式を計算する.

## SELECT 節中の \* (star)

- FROM節に一つしかリレーションがないとき、SELECT節の \* は「そのリレーションのすべての属性」を意味する.
- 例えば, 酒(酒名, 製造元)

```
SELECT *  
FROM 酒  
WHERE 製造元='朝日酒造';
```

## 問合せの結果

酒名	製造元
‘久保田千寿’	‘朝日酒造’
‘久保田万寿’	‘朝日酒造’
‘久保田碧寿’	‘朝日酒造’

今度は、「酒」のすべての属性を出力する。

## 属性の名前の付け替え

- 属性名を付け替えたい場合,  
“AS <新しい名前>” を用いる
- 例えば, 酒(酒名, 製造元) に対して:

```
SELECT 酒名 AS 日本酒, 製造元 AS 酒蔵  
FROM 酒  
WHERE 製造元 = '朝日酒造'
```

## SELECT節中の式

- 意味のある任意の式は SELECT節の要素として記述できる
- 例えば 販売(呑み屋名, 酒名, 値段) を使って

```
SELECT 呑み屋名, 酒名,  
       値段 / 105 AS ドル建て価格  
FROM 販売;
```

## その結果...

呑み屋名	酒名	ドル建て価格
つぼ八	一番絞り	4.57
清龍	サッポロドラフト	3.50
...	...	...

## 別の例: 定数式

- 好き(酒呑み名, 酒名) を使って,

```
SELECT 酒呑み名, '八海山好き' AS 特徴
FROM 好き
WHERE 酒名 = '八海山';
```

酒呑み名	特徴
'太郎'	'八海山好き'
'花子'	'八海山好き'
...	...

## WHERE節の複雑な条件

- 販売(呑み屋名, 酒名, 値段) を使って, つぼ八での一番絞りの値段を調べるには:

```
SELECT 値段
```

```
FROM 販売
```

```
WHERE
```

```
    呑み屋名 = 'つぼ八' AND
```

```
    酒名 = '一番絞り';
```

## 文字列パターン

- WHERE節には, 文字列に関するパターンを条件として使うことができる.
- 一般形:  
<属性> LIKE <パターン> or  
<属性> NOT LIKE <パターン>
- パターンは文字列で
  - % = 任意の文字列
  - \_ = 任意の文字を含むことができる.

## パターンの例

- 酒呑み(酒呑み名, 住所, 電話番号) を使って, 局番が 3333 の酒呑みの名前を列挙するには:

```
SELECT 酒呑み名  
FROM 酒呑み  
WHERE 電話番号 LIKE '%-3333-____';
```

# NULL値

- タプルの要素は NULL を値としてとる可能性がある
- その意味は、場合によって変わる. よく使われる意味は,
  - 値がわからない: 例えば, 呑み屋「つぼ八」には住所があることはわかっているが, 具体的な住所がわからない.
  - 値がない(値に意味がない): 例えば, 結婚していない人の配偶者

## NULL値に関する比較演算

- SQL の論理演算は 3値論理に基づく:  
真, 偽, 不明
- NULL値との比較の結果は不明となる
- 問合せの結果には, WHERE節の条件の真偽値が TRUE となったタプルだけが含まれる.
  - 偽となったもの, 不明となったものは含まれない

## 3値論理

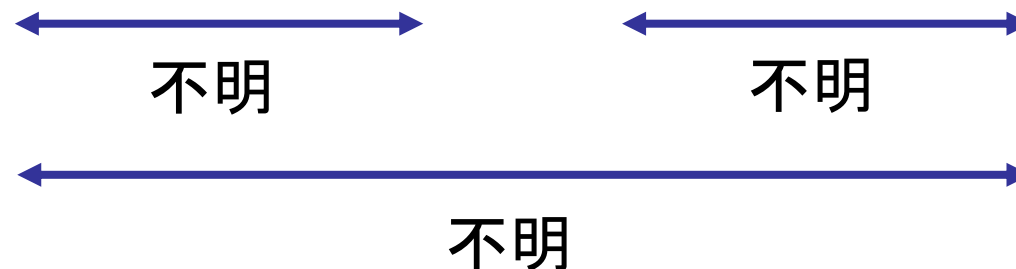
- 3値論理の意味を理解するには,
  - 真 = 1; 偽 = 0; 不明 =  $\frac{1}{2}$ .
  - AND = MIN; OR = MAX; NOT(x) = 1-x.として考えてみればよい.
- 例えば:
  - 真 AND (偽 OR NOT(不明)) =  
MIN(1, MAX(0, (1 -  $\frac{1}{2}$ ))) =  
MIN(1, MAX(0,  $\frac{1}{2}$ )) = MIN(1,  $\frac{1}{2}$ ) =  $\frac{1}{2}$  = 不明.

## 変な例

- 次の販売リレーションに対して

呑み屋名	酒名	値段
清龍	チュウハイ	NULL

```
SELECT 呑み屋名  
FROM 販売  
WHERE 値段 < 400 OR 値段 >= 400;
```



3値論理では、中排律が成り立たない

## 複数のリレーションに対する問合せ

- 役に立つ問合せは, しばしば複数のリレーションのデータを組み合わせる.
- FROM節にいくつもリレーションを並べればよい.
- 同名の属性がある場合は “<テーブル>.<属性>” として区別する.

## 例えば...

- 好き(酒呑み名, 酒名) と 行きつけ(酒呑み名, 呑み屋名) を使って, キリンシティーに行きつけている人が好きな酒を調べる:

```
SELECT 好き.酒名  
FROM 好き, 行きつけ  
WHERE 行きつけ.呑み屋名 = 'キリンシティー'  
AND 好き.酒呑み名 = 行きつけ.酒呑み名;
```

## 形式的な意味

- 単一リレーションに対する問合せとほとんど同じ:
  - FROM節のすべてのリレーションの積を取り,
  - WHERE節の条件を適用して(真のものだけを残し),
  - SELECT節にある属性リストに射影する

## 手続き的な意味

- FROM節にあるリレーションー一つずつに、タプル変数を想像する。
  - そのタプル変数の集合は、全部のタプルの組み合わせを値として取る。
- もしタプル変数の値がWHERE節の条件を満たせば、SELECT節に値を送る。

# 例

```
SELECT 酒  
FROM 好き, 行きつけ  
WHERE 呑み屋 = '麒麟シティー' AND  
好き.酒呑み = 行きつけ.酒呑み;
```

行きつけ

酒呑み名	呑み屋名
太郎	麒麟シティー

tv1



好き

酒呑み名	酒名
太郎	ブラウマイスター

tv2



麒麟シティーかどうか  
チェック

これらの値が  
等しいかどうかチェック

出力

## 明示的なタプル変数の使い方

- 一つの間合せの中で、同じリレーションのコピーを複数回参照したいときがある。
- FROM節のテーブル名の後にタプル変数をつけて、それぞれのコピーを区別する。

## 例えば...

- 酒(名前, 製造元) から同じ製造元が作る酒の組を求めたい
  - 単一製品の組 (e.g., (八海山, 八海山)) は要らない
  - 組の一方だけがほしい. (e.g., (千寿, 万寿)) は要るが, (万寿, 千寿) は要らない

```
SELECT 酒1.酒名, 酒2.酒名
FROM 酒 酒1, 酒 酒2
WHERE
    酒1.製造元 = 酒2.製造元 AND
    酒1.酒名 < 酒2.酒名;
```

## 副問合せ (sub-query)

- 括弧 () で囲まれた SELECT-FROM-WHERE 文 (sub-query) はいろいろなところで使うことができる。
- 例えば, FROM節のリレーション名を書く場所におくと, 問合せの結果に対して問合せをすることができる
  - 問合せの中間結果に名前をつけるため, タプル変数を使う

## 値が1つしかない副問合せ

- もし、副問合せの結果がタプル1つだけだということが保証されている場合、その副問合せは値として用いることができる。
  - 普通、そのタプルは要素が1つだけ
  - タプルが1つというのは、キーによって保障される
  - タプルが1つではないとき(0個のときまたは2個以上のとき)実行時エラーが発生する

## 例えば...

- 販売(呑み屋名, 酒名, 値段) から魚民が一番絞りに付けている値段と同じ値段で, モルツを売っている店を探したい
- 次の2つの問合せでうまくいくはず:
  - 魚民の一番絞りの値段を調べる
  - その値段でモルツを売っている店を探す

## 副問合せを使った解答

```
SELECT 呑み屋名  
FROM 販売  
WHERE
```

```
酒名 = 'モルツ' AND
```


```
値段 = (
```

```
SELECT 値段  
FROM 販売  
WHERE
```

```
呑み屋名 = '魚民' AND
```

```
酒名 = '一番絞り');
```

魚民での  
一番絞りの  
値段



## “IN” 演算子

- <タプル> IN <リレーション> が真となるのは、タプルがリレーションの要素であるとき、そのときのみ。
  - <タプル> NOT IN <リレーション> はその反対
- IN-式は WHERE節で用いられる
- <リレーション> は普通、副問合せ.

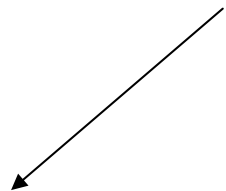
## 例えば...

- 酒(名前, 製造元) と 好き(酒呑み, 酒) から, 太郎が好きな酒の名前と製造元を求めよ

```
SELECT *  
FROM 酒  
WHERE
```

```
名前 IN (SELECT 酒名  
FROM 好き  
WHERE 酒呑み名 = '太郎');
```

太郎が好きな  
酒の名前の集合



## “EXISTS” 演算子

- EXISTS(<リレーション>) が真となるのは、リレーションが空でないとき、そのときのみ.
- EXISTS式は WHERE節 で用いられる

## 例えば...

- 酒(名前, 製造元) から, 製造元がそれだけしか作っていないような酒を求めよ

```
SELECT 酒名  
FROM 酒 酒1  
WHERE NOT EXISTS(  

```

酒1と同じ製造元で  
名前が異なる酒の集合

```
SELECT *  
FROM 酒  
WHERE 製造元 = 酒1.製造元 AND  
酒名 <> 酒1.酒名);
```

製造元, 名前のスコープは  
一番近い酒

SQL の不等号

## “ANY” 演算子

- $x = ANY( \langle \text{リレーション} \rangle )$  は、 $x$  がリレーション中の少なくとも一つと等しいとき、そのときに限り、真となる。
- 同様に、“=” はほかの比較演算子と置き換えることができる
  - 例えば、 $x \geq ANY( \langle \text{リレーション} \rangle )$  は、リレーションのいずれか一つでも  $x$  以下のとき真となる。
  - リレーションは複数の属性をもてない。

## “ALL” 演算子

- 同様に,  $x \langle \rangle \text{ALL}(\langle \text{リレーション} \rangle)$  は  $x$  がリレーションのどのタプルとも等しくないとき, そのときに限り, 真となる
  - つまり,  $x$  はリレーションのメンバーでないということ
- やはり, “ $\langle \rangle$ ” はほかの任意の比較演算子と取り替えることができる
  - 例えば,  $x \geq \text{ALL}(\langle \text{リレーション} \rangle)$  はリレーションのどのタプルも  $x$  以下のとき真となる.

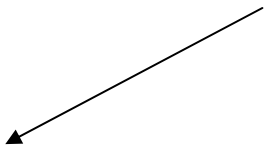
## 例えば...

- 販売(呑み屋名, 酒名, 値段) から最も高い値段で売られる酒を見つけたい場合

```
SELECT 酒名  
FROM 販売  
WHERE
```

```
値段 >= ALL(  
    SELECT 値段  
    FROM 販売);
```

外側の販売の値段が  
内側の販売のどの値段より  
大きいか等しい



和 (union), 交差 (intersection), 差 (difference)

- 和, 交差, 差は, 副問合せを用いて, 次のように表現される:
  - (副問合せ) UNION (副問合せ)
  - (副問合せ) INTERSECT (副問合せ)
  - (副問合せ) EXCEPT (副問合せ)

## 例えば...

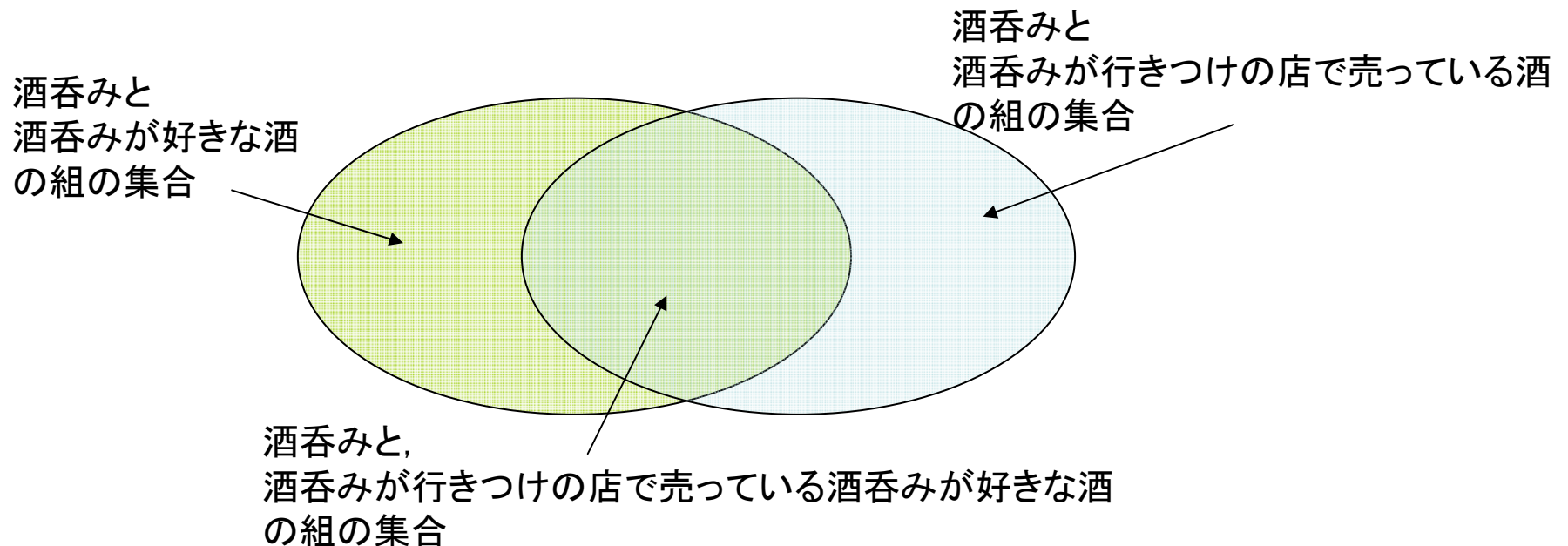
- 好き(酒呑み名, 酒名), 販売(呑み屋名, 酒名, 値段), 行きつけ(酒呑み名, 呑み屋名) から, 次のような酒呑みと酒を見つける:
  - 酒呑みはある酒がすきで,
  - その酒呑みが行きつけの店少なくとも一つで, その酒が売られている

# 解答

(SELECT 酒呑み名, 酒名 FROM 好き)  
INTERSECT

(SELECT 酒呑み名, 酒名  
FROM 販売, 行きつけ

WHERE 販売.呑み屋名 = 行きつけ.呑み屋名 );



## バッグ意味論

- SELECT-FROM-WHERE 文はバッグ意味論を用いているが、和・交差・差は、デフォルトでは集合意味論を用いる
  - すなわち、実行時に重複は除去される

## 動機: 効率

- 射影 (projection) を行うときには、重複を除去しないほうが簡単
  - 単に、タプルごとに処理すればよい
  - だからバッグ意味論を採用する。
- 交差 (intersection), 差 (difference) を実行する際には、まずソートしてしまうのが一番簡単
  - ソートした時点で、簡単に重複が除去できる
  - だから集合意味論を採用する

## 重複除去の制御方法

- 結果が集合（重複がない）ように強制するには、  
SELECT DISTINCT ...
- 結果がバッグ（出現回数を保存する）ように強制するには、“ALL”を用いて、  
UNION ALL ...

## 例: DISTINCT

- 販売(呑み屋, 酒, 値段) から売られている酒の値段を知るには,

```
SELECT DISTINCT 値段  
FROM 販売;
```

- DISTINCT を用いなければ, 同じ値段が, その呑み屋と酒の出現回数だけ表れる

## 例: ALL

- 行きつけ(酒呑み名, 呑み屋名) と 好き(酒呑み名, 酒名) を用いて, 行きつけの店が好きな酒の数より多い酒呑みを調べるには:

```
(SELECT 酒呑み名 FROM 行きつけ)  
EXCEPT ALL  
(SELECT 酒呑み名 FROM 好き);
```

とすると, そのような酒呑みが行きつけの店と好きな酒の数の差の回数, 結果に表れる

# JOIN 式

- SQL は関係代数の結合 (join) を表す, 数多くの形式を提供している
- 自然結合 (natural join):  
    R NATURAL JOIN S;
- 積 (product):  
    R CROSS JOIN S;
- 例えば...  
    好き NATURAL JOIN 販売;
- リレーション (上記では R, S) は副問合せ (括弧で囲われた SELECT 文) でも良い

## θ結合 (theta join)

- R JOIN S ON <条件> は, <条件>を選択条件とする θ結合を表す.
- 例えば... 酒呑み(酒呑み名, 住所, 電話番号)と行きつけ(酒呑み名, 呑み屋名)に対して

酒呑み JOIN 行きつけ ON  
酒呑み.酒呑み名 = 行きつけ.酒呑み名;

とすると, 酒呑みとその行きつけの店に関する5つ組 (酒呑み名, 住所, 電話番号, 酒呑み名, 呑み屋名) が得られる.

## 外部結合 (outer-join)

- R OUTER JOIN S が外部結合の基本形.
- 以下のようなバリエーションがある:
  - NATURAL を OUTER の前につける
  - ON <条件> 後ろにつける
  - LEFT, RIGHT, FULL を OUTER の前につける
    - LEFT = R のダングリングタプルだけを埋める
    - RIGHT = S のダングリングタプルだけを埋める
    - FULL = 両方を埋める. (デフォルト)

## 集約演算

- SUM, AVG, COUNT, MIN, MAX をSELECT節のカラムに適用することができる
- COUNT(\*) はタプル数を数える
- 例えば..., 販売(呑み屋名, 酒名, 値段) から八海山の平均価格を求めるには:

```
SELECT AVG(値段)  
FROM 販売  
WHERE 酒名 = '八海山';
```

## 集約の中の重複除去

- 集約関数の中に DISTINCT と書くと, 集約演算の前に重複の除去を行う
- 例えば..., 八海山の異なる値段の数を求めるには:

```
SELECT COUNT(DISTINCT 値段)  
FROM 販売  
WHERE 酒名 = '八海山';
```

## 集約における NULL の扱い

- NULL は合計, 平均, カウントなどには一切貢献せず, 無視される
- しかし, すべてが NULL だったとき, その集約結果も NULL となる

## 例: 集約における NULLの扱い

```
SELECT count(*)  
FROM 販売  
WHERE 酒名 = '八海山';
```

← 八海山を売っている  
店の数

```
SELECT count(値段)  
FROM 販売  
WHERE 酒名 = '八海山';
```

← 八海山を売っている  
値段がわかっている  
店の数

## グループ化 (grouping)

- SELECT-FROM-WHERE 文の後ろに, GROUP BY 節をつけることができる.
- SELECT-FROM-WHERE の結果が GROUP-BY 節で指定した属性の値でグループ化され, グループごとに集約演算が計算される.

## 例: グループ化

- 販売(呑み屋名, 酒名, 値段) から, 酒ごとの平均価格を求めるには,

```
SELECT 酒名, AVG(値段)  
FROM 販売  
GROUP BY 酒名;
```

## 例: グループ化

- 販売(呑み屋名, 酒名, 値段) と 行きつけ(酒呑み名, 呑み屋名) から, 各酒呑みごとの, 行きつけの店の八海山の平均価格を求めよ

```
SELECT 酒呑み名, AVG(値段)
FROM 販売, 行きつけ
WHERE 酒名 = '八海山' AND
      行きつけ.呑み屋名 = 販売.呑み屋名
GROUP BY 酒呑み名;
```

# 集約演算を伴う SELECT リストの制限

- 集約演算が使われている場合, SELECT リストに現れる属性は, 次のいずれかでなければならない:
  - 集約演算が施されるか,
  - GROUP-BY リストに現れる

- 正しくない問合せの例:

```
SELECT 呑み屋名, MIN(値段)  
FROM 販売  
WHERE 酒名 = '八海山';
```

八海山を最低の値段で販売している呑み屋を求める??? → 誤り

理由: SELECTリストの「呑み屋」は

- 集約演算が施されていないし,
- GROUP-BYリストにも現れない

## HAVING 節

- GROUP-BY節の後ろに HAVING <条件> をつけることができる.
- その場合, 各グループにその条件が適応され, 条件を満たさないグループは取り除かれる.

## HAVING節に書くことができる条件

- HAVING節の条件で参照される属性は、次のいずれかでなければならない：
  - GROUP-BY リストに現れている属性か、
  - 集約演算が施されている

## 例: HAVING節

- 販売(呑み屋名, 酒名, 値段)と酒(酒名, 製造元)から, 朝日酒造で作られている酒で, 最低3つ以上の呑み屋で売られている酒の, 平均価格を調べたい.

```
SELECT 酒名, AVG(値段)
FROM 販売, 酒
WHERE 販売.酒名 = 酒.酒名 AND
      酒.製造元 = '朝日酒造'
GROUP BY 酒名
HAVING COUNT(呑み屋名) >= 3
```

## データベースの更新

- 更新コマンドは問合せのように結果を返さないが、データベースに何らかの変更を加える。
- 3種類の更新がある：
  - 挿入 (insert): リレーションに新しいタプルを追加する
  - 削除 (delete): リレーションからタプルを取り除く
  - 更新 (update): 既存のタプルの値を変える

## 挿入 (insertion)

- 一つのタプルを挿入するには:

```
INSERT INTO <リレーション>  
VALUES ( <値のリスト = タプル> );
```

- 例えば...: 好き(酒呑み名, 酒名) に花子がバス  
ペールエールが好きだという事実を加えるには:

```
INSERT INTO 好き  
VALUES('花子', 'バスペールエール');
```

## 属性の指定

- リレーションに属性リストをつけることができる.
- そうする理由:
  - 属性の順番を覚えておかなくても済む.
  - 属性に明示的に与える値がなく, デフォルト値や NULL値で埋めたいとき.

## 例: 属性の指定

- 好き(酒呑み名, 酒名) に花子がバスペールエールが好きだという事実を加えるには:

```
INSERT INTO 好き(酒名, 酒呑み名)  
VALUES('バスペールエール', '花子');
```

## 複数のタプルの挿入

- 問合せの結果をまとめてリレーションに挿入することができる:

```
INSERT INTO <リレーション>  
( <副問合せ> );
```

## 例: 副問合せによる挿入

- 行きつけ(酒呑み, 呑み屋) から, 太郎の呑み友達候補, つまり太郎と同じ呑み屋に行きつけている酒呑み達を求め, リレーション「太郎の呑み仲間(名前)」に入れるには:

INSERT INTO 太郎の呑み仲間

(SELECT d2.酒呑み名

FROM 行きつけ d1, 行きつけ d2  
WHERE d1.酒呑み名 = '太郎' AND  
d2.酒呑み名 <> '太郎' AND  
d1.呑み屋名 = d2.呑み屋名

);

太郎じゃないほうの  
酒呑み

太郎と, 太郎以外の  
誰かで, 同じ呑み屋に  
行きつけている人の  
ペアを作る

## 削除 (deletion)

- 条件を満たすタプルを削除するには:

```
DELETE FROM <リレーション>  
WHERE <条件>;
```

- 例えば...: 好き(酒呑み名, 酒名) から花子がスーパー  
ドライが好きだという事実を削除するには:

```
DELETE FROM 好き  
WHERE 酒呑み名 = '花子' AND  
      酒名 = 'スーパードライ';
```

## 例：全タプルの削除

- リレーション「好き」を空にするには:

```
DELETE FROM 好き;
```

- WHERE節は不要

## あいまいな例: 複数タプルの削除

- 酒(酒名, 製造元) から, 製造元が同じ酒があるような酒 (= 複数の酒を造っている製造元が造る酒) を全て削除する.

```
DELETE FROM 酒 s1  
WHERE EXISTS (
```

```
SELECT 酒名 FROM 酒 s2  
WHERE s1.製造元 = s2.製造元 AND  
s1.酒名 <> s2.酒名);
```

↑  
タプル変数 s1 で表される酒と  
同じ製造元を持ち, 名前が異  
なる酒の名前のリレーション

## 削除の意味論 (1)

- 朝日酒造が久保田千寿と久保田万寿しか作っていないとする。
- 久保田千寿に先にたどり着いたとする。
- 朝日酒造は久保田万寿も作っているなので、この副問合せは空でない。したがって、久保田千寿は削除される。
- では、久保田万寿の番になったとき、削除すべきだろうか？

## 削除の意味論 (2)

- 答え： 久保田万寿も削除する.
- 理由： 削除は次の2フェーズに分けて実行することになっているから：
  - 元のリレーションの中で, WHERE節の条件を満たすもの全てに印をつける.
  - 印のついたタプルを削除する.

## 更新 (update)

- あるタブルのある属性の値を変更するには:

```
UPDATE <リレーション>  
SET <属性に対する代入のリスト>  
WHERE <条件>;
```

- 例えば...: 酒呑み「太郎」の電話番号を変更するには,

```
UPDATE 酒呑み  
SET 電話番号 = '03-1234-5678'  
WHERE 酒呑み名 = '太郎';
```

## 例: 複数タプルの更新

- 酒の値段の上限を 600円 にする.

```
UPDATE 販売  
SET 値段 = 600  
WHERE 値段 > 600;
```

## データベーススキーマの定義方法

- データベーススキーマ (database schema) は、データベース中のリレーション(テーブル)の宣言 (declaration) の集まり
- あとで紹介するビュー (view), インデックス (index), トリガ (trigger) などもデータベーススキーマの一部

## リレーションの宣言

- 最も簡単な形式:

```
CREATE TABLE <名前> (  
    <テーブル要素のリスト>  
);
```

- リレーションをデータベーススキーマから削除するには:

```
DROP TABLE <name>;
```

## テーブル要素の宣言

- テーブル要素の基本形は, 属性名と型のペア
- よく使われる型は:
  - INT or INTEGER
  - REAL or FLOAT
  - CHAR(n) = n 文字の固定長文字列
  - VARCHAR(n) = 最大 n 文字までの可変長文字列

## 例: 販売テーブルの作成

```
CREATE TABLE 販売 (  
    呑み屋名 CHAR(20),  
    酒名      VARCHAR(20),  
    値段     REAL  
);
```

## 日付 (date) と 時間 (time)

- DATE と TIME は SQL の型
- DATE型の値の形 = DATE 'yyyy-mm-dd'
  - 例えば, DATE '2004-12-14' は 2004年12月14日
- TIME型の値の形 = TIME 'hh:mm:ss[.xxx]'
  - ただし, 括弧内 [.xxx] はなくても良い
  - 例えば, TIME '14:30:03.5' は午後2時半の3.5秒後

## キーの宣言

- 属性(一つまたは複数)は PRIMARY KEY または UNIQUE と宣言できる.
- どちらも, その属性(一つまたは複数)に, 他の全ての属性が, 関数従属であることを意味する.
- 若干の違いがある(後で述べる)

## 一つの属性がキーのとき

- 型の後ろに PRIMARY KEY または UNIQUE とつける.

- 例:

```
CREATE TABLE 酒 (  
    酒名          CHAR(20) PRIMARY KEY,  
    製造元       CHAR(20)  
);
```

## 複数属性がキーのとき

- キーの宣言は、属性の宣言とは別に、テーブル要素として記述することもできる
  - この形式は、キーが複数属性のときに必要となる
  - キーが単一属性の場合に使ってもよい
- 例えば...: {呑み屋名, 酒名}がキーの販売テーブルの宣言

```
CREATE TABLE 販売 (  
    呑み屋名 CHAR(20),  
    酒名     VARCHAR(20),  
    値段     REAL,  
    PRIMARY KEY (呑み屋名, 酒名)  
);
```

## PRIMARY KEY 対 UNIQUE

- SQLの標準は、次のような違いを規定している：
  - 一つのリレーションに対して、高々一つしか PRIMARY KEY があってはならない。UNIQUE はいくつあっても良い。
  - PRIMARY KEY 属性の値は NULL であってはならないが、UNIQUE 属性は NULL を許す。
- SQLの標準は、DBMSの実装に対して、独自の違いを許している
  - 例えば、あるDBMSはPRIMARY KEYに対して自動的にインデックスを作成するが、UNIQUEに対しては作成しないなど。

## 属性に対するその他の宣言

- NOT NULL: その属性値が NULL になってはならないことを宣言する.
- DEFAULT <value>: その属性に対して値が指定されなかったとき, <value> をデフォルト値として用いることを宣言する.
- 例えば....:

```
CREATE TABLE 酒呑み (  
    酒呑み名 CHAR(30) PRIMARY KEY,  
    住所 CHAR(50) DEFAULT '千代田区1-1',  
    電話 CHAR(16)  
);
```

## DEFAULT の効果 (1)

- リレーション「酒呑み」に花子を追加することにするが、住所も電話番号もわからなかったとする。
- INSERT文に一部の属性リスト(この場合、酒呑み名)だけを指定すれば、それが可能:

```
INSERT INTO 酒呑み(酒呑み名)  
VALUES('花子');
```

## DEFAULT の効果 (2)

- このとき、挿入されたタプルはどうなるのか？

名前	住所	電話
‘花子’	‘千代田区1-1’	NULL

- もし、電話が NOT NULL と宣言されていたら、この挿入はエラーとなって拒否される。

## 属性の追加

- リレーションスキーマに新しい属性を追加することができる:

```
ALTER TABLE <名前> ADD  
    <属性定義>;
```

- 例えば....:

```
ALTER TABLE 呑み屋 ADD  
    電話 CHAR(16) DEFAULT '不明';
```

## 属性の削除

- リレーションスキーマから不要な属性を削除することができる:

```
ALTER TABLE <名前>  
DROP <属性>;
```

- 例えば....:

```
ALTER TABLE 呑み屋  
DROP 電話;
```

## ビュー (view)

- ビュー (view) は, 他のテーブルの内容を使って定義される「仮想テーブル」.
- 定義方法:  
CREATE VIEW <名前> AS <問合せ>;
- これに対して, データベースに直接, 値が格納されるテーブルのことを「ベーステーブル (base table)」と呼ぶ.

## 例: ビュー一定義

- ビュー「飲める(酒呑み名, 酒名)」は酒呑みが行きつけの店のいずれかで売っている酒とその酒呑みのペアを「保持する」テーブルとすると:

```
CREATE VIEW 飲める AS
  SELECT 酒呑み名, 酒名
  FROM 行きつけ, 販売
  WHERE 行きつけ.呑み屋名 = 販売.呑み屋名;
```

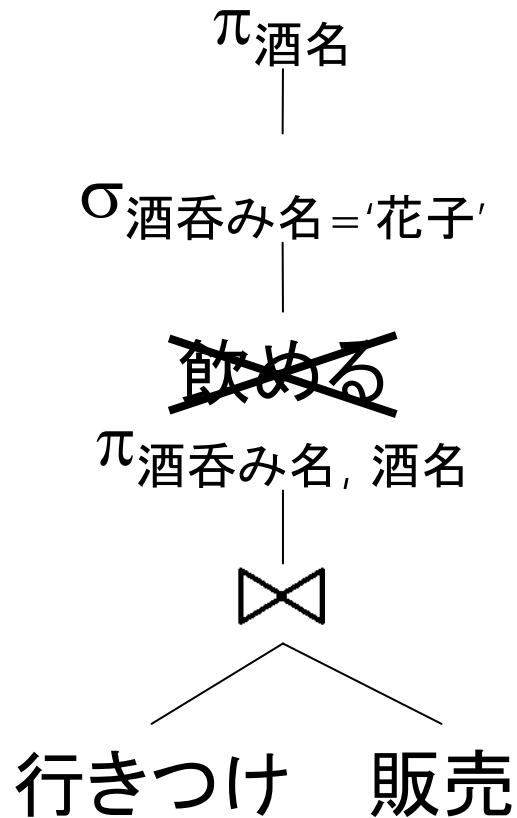
## 例: ビューの使い方

- ビューはテーブルと同様に使える
  - ただし, 変更は, 元になるベーステーブルに対する変更として意味のあるものだけに, 限定される.
- 例えば....:

```
SELECT 酒名 FROM 飲める  
WHERE 酒呑み名 = '花子';
```

# ビューが問合せで使われるとどうなるか？

```
SELECT 酒名  
FROM 飲める  
WHERE 酒呑み名 = '花子';
```



```
CREATE VIEW 飲める AS  
SELECT 酒呑み名, 酒名  
FROM 行きつけ, 販売  
WHERE 行きつけ.呑み屋名 = 販売.呑み屋名;
```

## 問合せ最適化 (query optimization)

- この後, ほとんどの DBMS は, 代数式を同じ意味を持つがより高速に実行できるものに変換する「最適化」を行う.
- 最適化のキーポイント:
  - 選択 (selection) を木の下のほうへ押し込む
  - 不要な射影 (projection) を削除する

# 例:最適化

