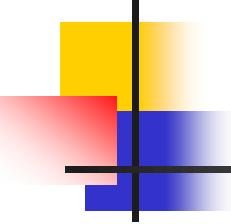


# データベースとは何か

- データベースとは
  - データベースシステムとDBMS
- データモデル
  - 概念モデルと論理モデル
  - 実体－関連モデル
- リレーショナルデータベースの意義

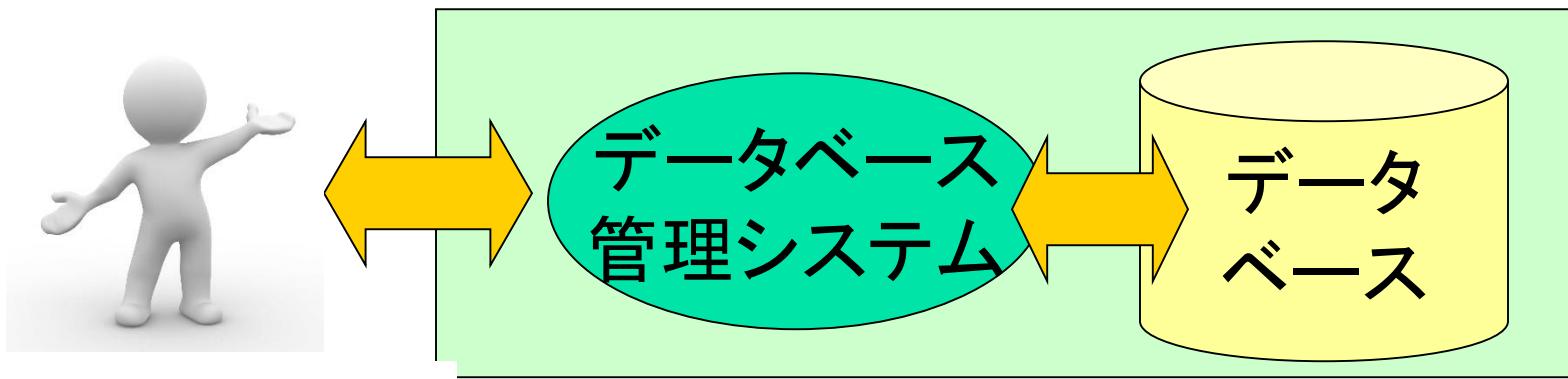


# データベースとは

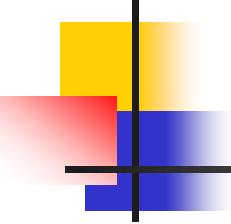
- データベースとは？
  - 複数の応用目的での共有を意図して組織的かつ永続的に格納されたデータ群
  - 大量データの検索、更新処理など機能を提供
- データベースの適用例
  - 企業の顧客・在庫・売上などのビジネスデータの管理
  - 文書データ、地理データベース、DNAデータ、動画像データなどマルチメディアデータにも適用

# データベースシステムとDBMS

- データベースシステム
  - データベース
    - データを組織的・永続的に管理
  - データベース管理システム(DBMS)
    - データベースを管理するためのソフトウェア



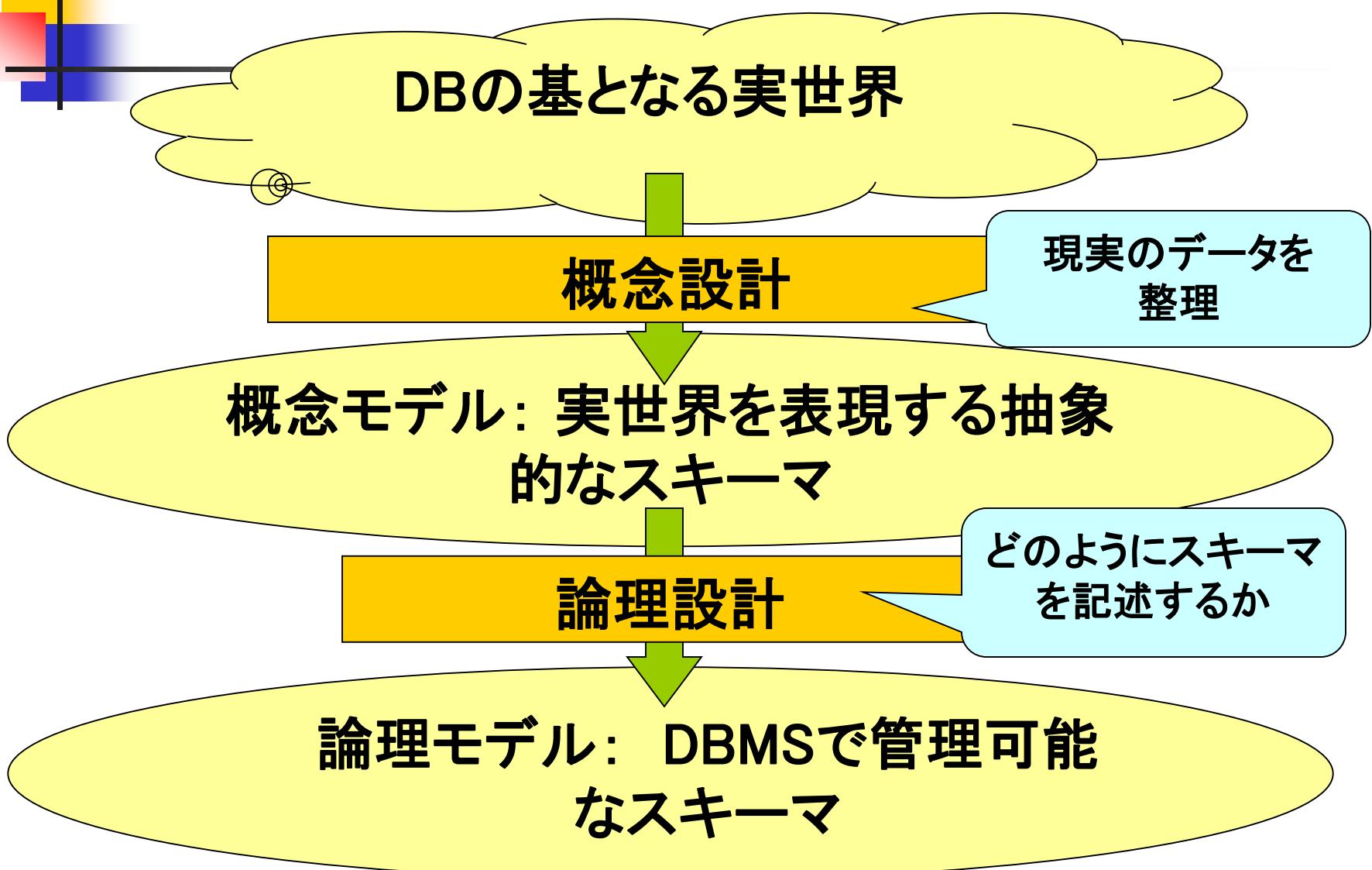
プログラムまたは人

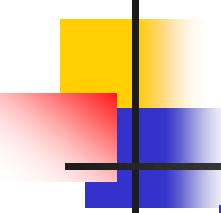


# DBMSの役割

- ペタバイト・エクサバイト単位のデータも取り扱える
  - 格納・管理・検索
- 数百万ユーザーからの同時アクセスを(矛盾なく)処理可能
- システム障害からデータを守る
  - データは決して失われない
- 24時間365日稼動が求められる

# データモデル





# 概念モデルと論理モデル

## ■ 概念設計の狙い

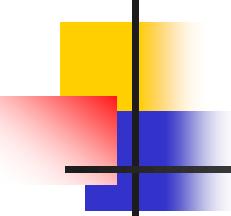
- DBMS のスキーマとは独立に実世界のデータを整理

## ■ 概念モデル

- 人間が見て理解できる図式
- 1976年にP. P. Chenが、実体-関連モデル(Entity–Relationship model)を提案

## ■ 論理モデル

- DBMS で管理可能なスキーマ
- 例えば、リレーションナルデータモデル



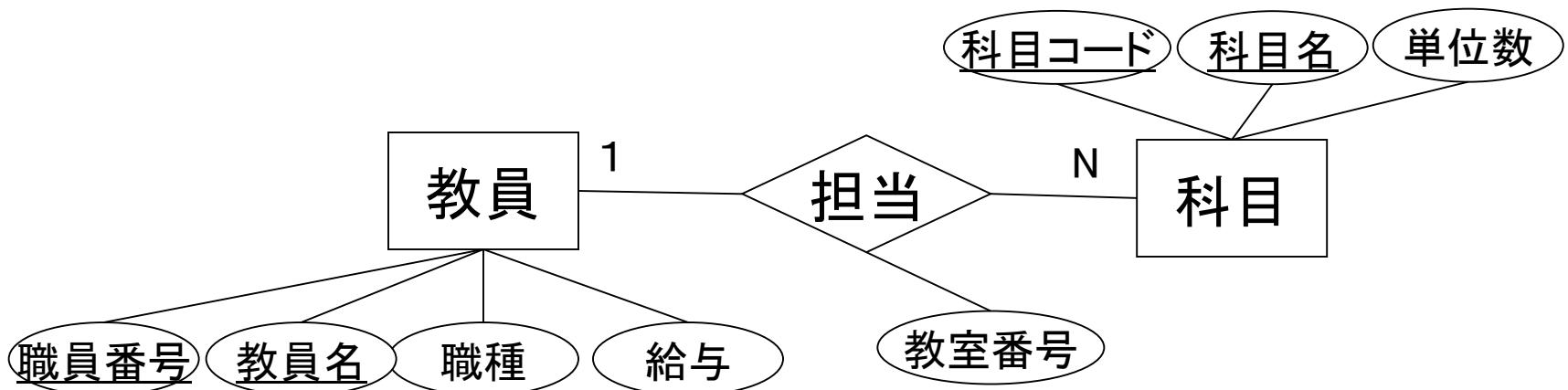
# 実体一関連モデル

- 実体一関連モデルでの実世界の見え方
  - 実体 (entity) : 実世界に存在する事物
    - 名詞に相当
    - 実体型 (entity type) を構成
  - 関連 (relationship) : 実体間の相互関係を表現
    - 動詞に相当
    - 関連型 (relationship type) を構成
  - 属性 (attribute) : 実体、関連に付随する性質
    - 形容詞に相当

# 実体-関連図の構成要素

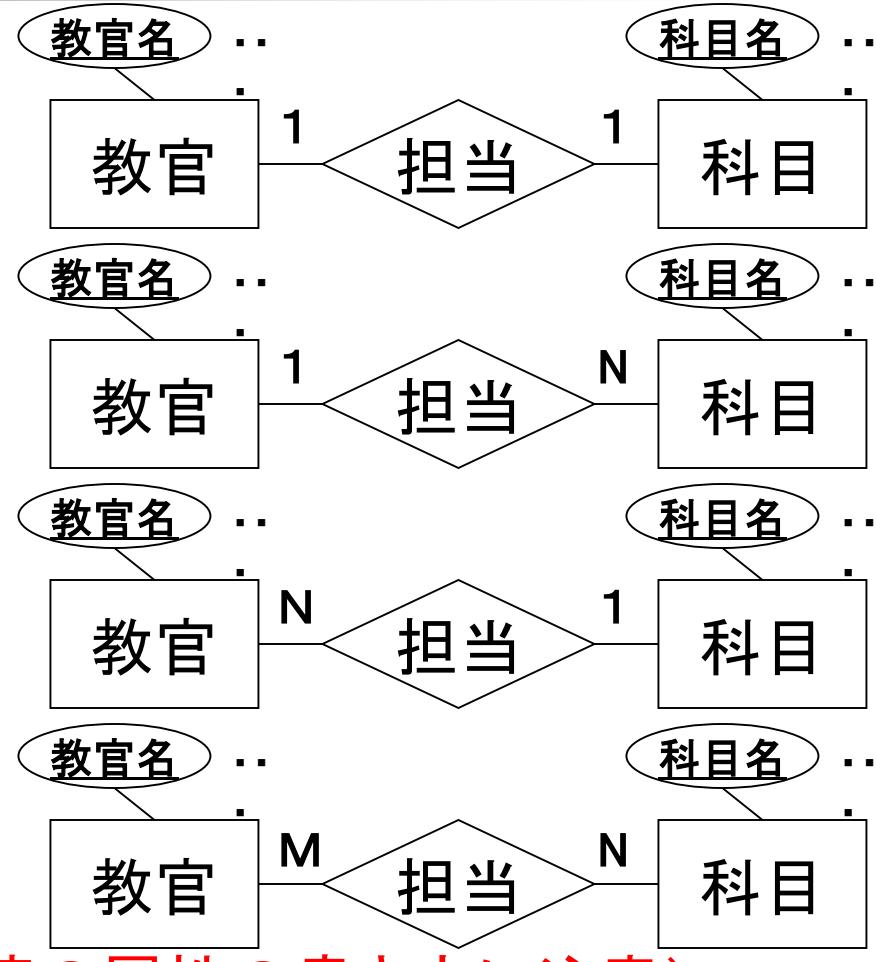
## ■ 実体-関連図 (E-R図)

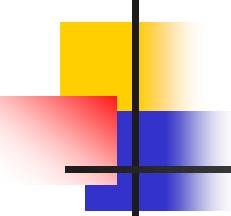
- 実体と関連を図式で示したもの
- 矩形: 実体の集合
- 菱形: 関連の集合
- 楕円: 属性の集合
- 線: 実体間の対応関係



# 実体間の対応関係

- 対応関係の分類
  - 1対1関連型
  - 1対多関連型
  - 多対1関連型
  - 多対多関連型
- 主キー(primary key)
  - 実体を識別する属性集合
  - 下線を引き、他と区別(関連の属性の書き方に注意)





# 次の文に含まれるデータ構造を E-R図で表現してみよう

- 各学生は複数の科目を履修できる
- 科目の情報として科目名と単位数がある
- 学生の情報として学籍番号、氏名、住所がある
- 科目を履修したら、科目名・学籍番号の組に対して(テストの)得点が付加される

# 【参考】拡張E-R図(鳥の足法)



実は、いまはあまり使われていない。なぜなら、エンティティもリレーションも同じ「表」の形をしているのに、役割によって異なる記号をつけられてしまう。実際は、以下の鳥の足記法が使われることが多い。

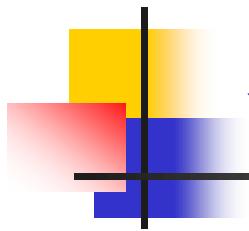


1:多

1:1

# 論理データモデルの種類と歴史

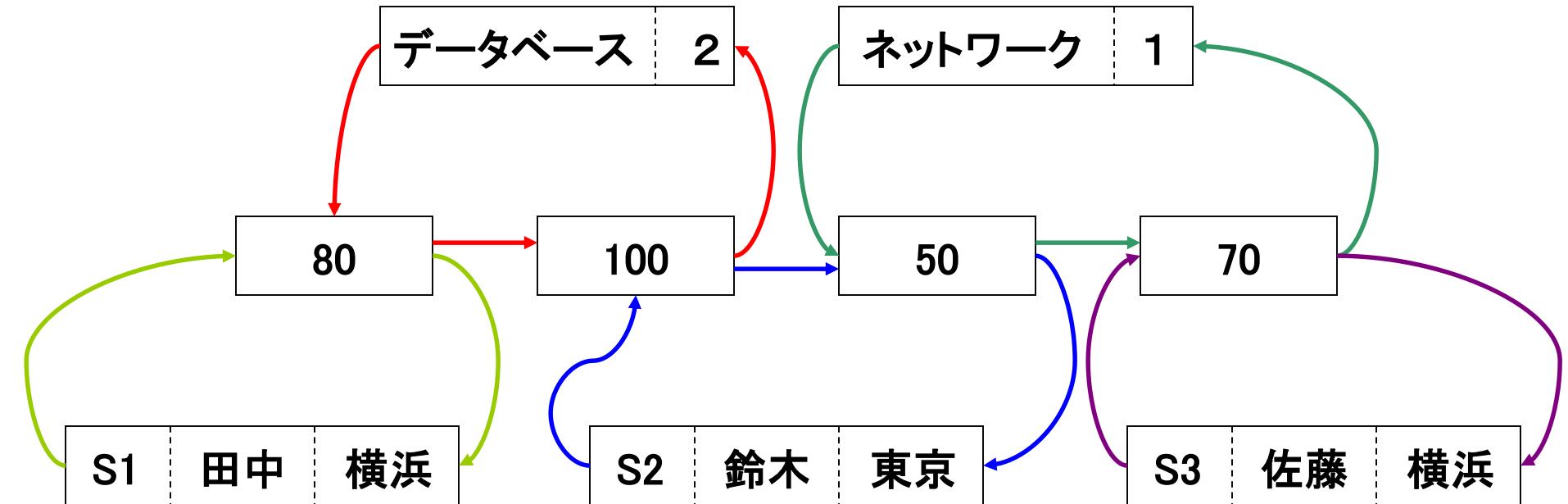
- 手続き型プログラミング言語(COBOLなど)を使ってファイルを直接操作
- 効率的にデータ処理を行うシステムを開発
  - ネットワークデータモデル(網構造)
  - ハイアラキカルデータモデル(階層構造)
- データモデルと物理構造の分離(データ独立)が進展
  - リレーションナルデータモデル
  - オブジェクト指向データモデル(参照構造)
- 現在はリレーションナルデータモデルが主流



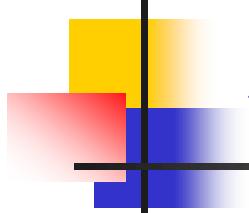
# ネットワークデータモデル

- ネットワークデータモデル
  - 1971年にCOBOLのデータベース機能として提案される
  - インスタンスはレコードオカレンスで表現
  - 親からすべての子を、一連のポインタをたどることで表現
  - ポインタを順にたどってデータ処理を実行

# ネットワークデータモデル例



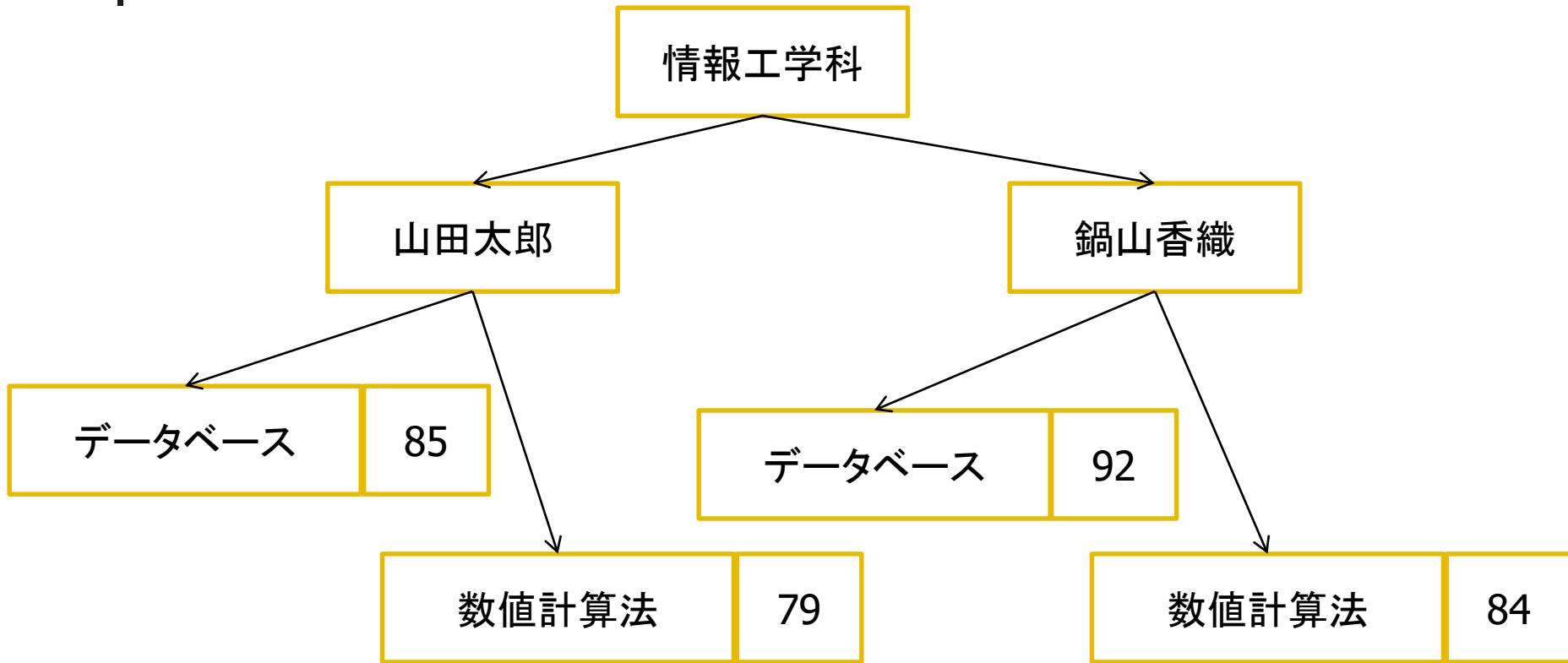
レコードオカレンス:  →  
ポインタ:



# ハイアラカルデータモデル

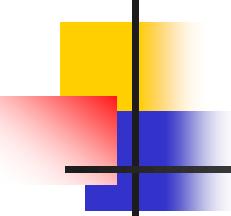
- ハイアラカルデータモデル
  - 1968年にIMS(IBM)によって開発
  - レコードの階層構造によってデータを表現
  - 多対多関係の表現が課題
  - 階層をたどってデータ処理を実行

# ハイアラキカルデータモデル例



レコード:  
ポインタ:

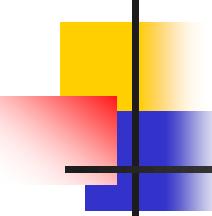
→



# リレーションナルデータモデル

## ■ リレーションナルデータモデル

- 1970年にCodd(IBM)によって関係理論をベースに提案
- 他のデータモデルと比べて、形式的でデータ独立性が高い
  - 複数の属性の組み合わせによってリレーション(関係、あるいは2次元の表)を定義
  - リレーション同士の関係演算により関係完備(Relational Complete)な処理を実現
- データ構造が複雑なデータ格納に課題



# リレーションナルデータモデル例

## 科目

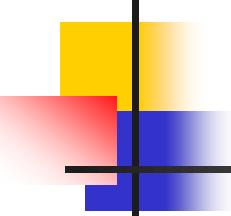
科目名	単位数
データベース	2
ネットワーク	1

## 学生

学籍番号	学生名	住所
S1	田中	横浜
S2	鈴木	東京
S3	佐藤	横浜

## 履修

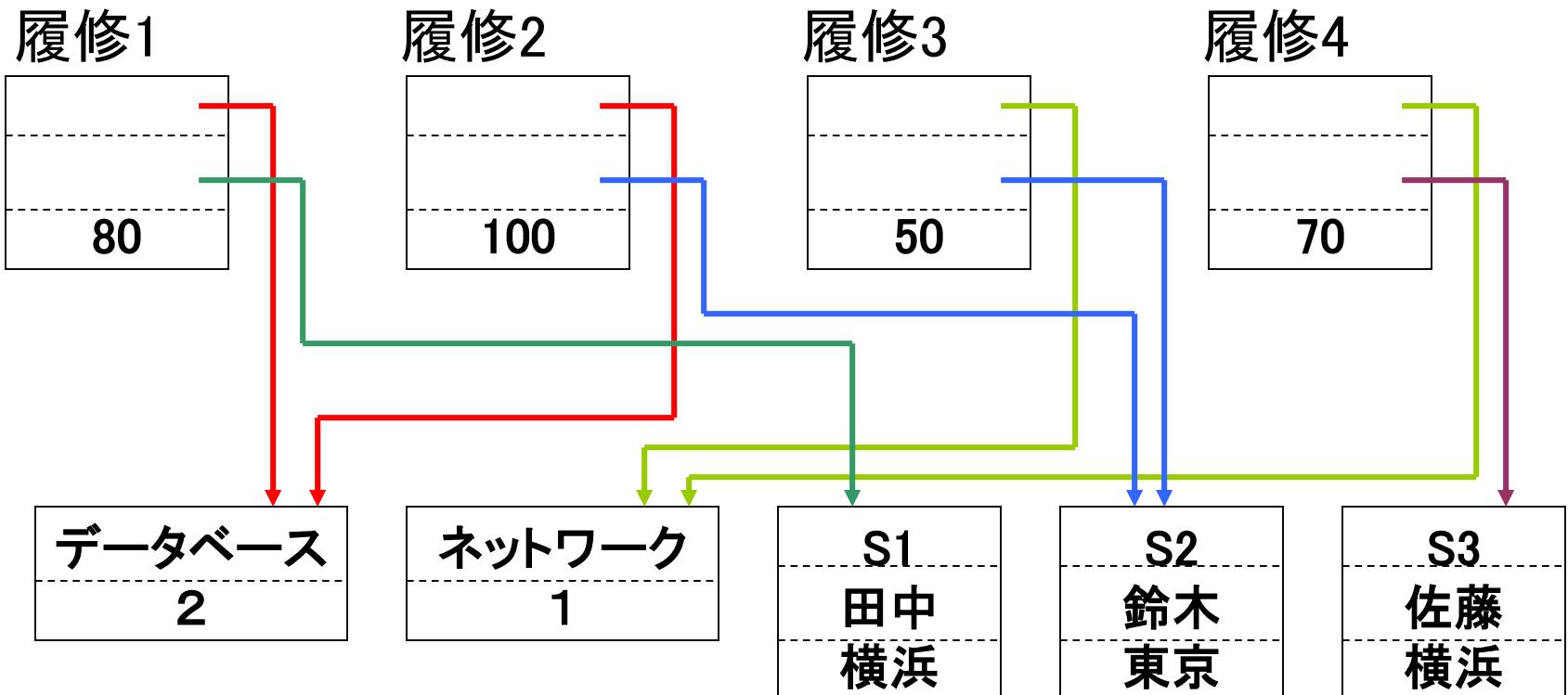
科目名	学籍番号	得点
データベース	S1	80
ネットワーク	S2	100
データベース	S2	50
ネットワーク	S3	70



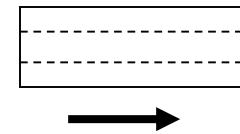
# オブジェクト指向データモデル

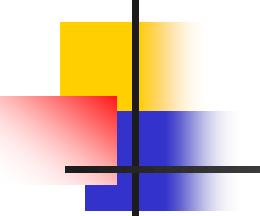
- オブジェクト指向データモデル
  - オブジェクト指向によりデータをモデリング
  - スキーマとしてクラスを定義
    - メソッドの定義
    - 繙承(inheritance)・ポリモルフィズム(polymorphism)などのオブジェクト指向の概念
  - Java や C++ などの複合オブジェクト(composite object)を永続化
  - 手続き型プログラミング言語からシームレスに操作可能

# オブジェクト指向データモデル例



インスタンス(オブジェクト):  
参照識別子:



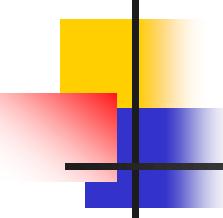


# ファイルシステムとの比較

## ■ ファイルシステムの課題

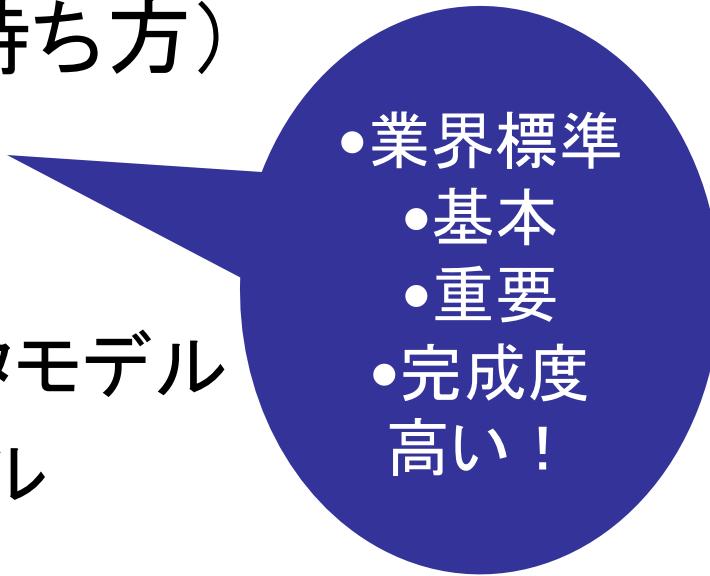
- データとプログラムの相互依存関係
  - データを作成したプログラムだけがデータの取り出し、変更が可能で、共用できない
- データの整合性を一元的に管理できない
  - データの一貫性(integrity)制御に課題
  - アクセス権限(privilege)の管理が不可
  - 機密保護(security)、障害回復(recovery)などの支援ができない

## ■ リレーショナルデータベースは上記の課題を解消

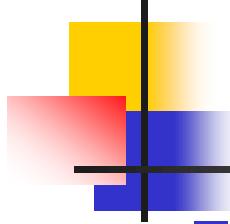


# データモデリング

- データモデルとは、情報をデータベースに保持する方法。(データの持ち方)
  - リレーションナルデータモデル
  - ネットワークデータモデル
  - ハイアラキカル(階層)データモデル
  - オブジェクト指向データモデル



- 業界標準
- 基本
- 重要
- 完成度  
高い！



# リレーション(1/4)

## ■ ドメイン

- 集合のこと
  - 人名の集合 ( $D_1 = \{x \mid x\text{は人名}\}$ )
  - 金額の集合 ( $D_2 = \{x \mid x\text{は金額}\}$ )
  - 整数(INTEGER)の集合 ( $D_3 = \{x \mid x=0, \pm 1, \pm 2, \dots\}$ )

など

## ■ ドメインの直積

- ドメインを単純に並べたもの

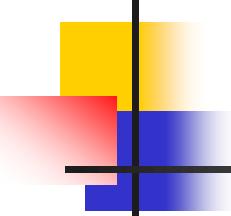
- $D = D_1 \times D_2 = \{(x,y) \mid x\text{は人名}, y\text{は金額}\}$

# リレーション(2/4)

## ■ ドメインの直積の例

- $D_1 = \{\text{“木村”, “坂本”, “桂”}\}$
- $D_2 = \{\text{“20万円”, “30万円”}\}$
- $D_1 \times D_2$   
 $= \{(\text{“木村”, “20万円”}), (\text{“木村”, “30万円”}),$   
 $\quad (\text{“坂本”, “20万円”}), (\text{“坂本”, “30万円”}),$   
 $\quad (\text{“桂”, “20万円”}), (\text{“桂”, “30万円”})\}$

要素の一つ一つを  
タップル(tuple)と呼ぶ



# リレーション(3/4)

それぞれのドメインに属したものを並べたものの集合  
のことをリレーションという

## ■ リレーションの定義

- $D_1, D_2, D_3, \dots, D_n$  をドメインとする。

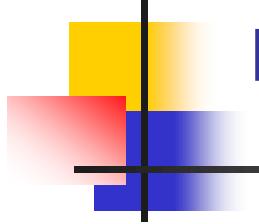
$D_1, D_2, D_3, \dots, D_n$ 上のリレーション

=直積  $D_1 \times D_2 \times D_3 \times \dots \times D_n$ の部分集合

$R = \{("木村", "20万円"), ("坂本", "30万円"), ("桂", "30万円")\}$

並べている個数 (=ドメインの数) = 次数

次数1=单項、次数2=2項、次数3=3項 …



## リレーション(4/4)

リレーションは次数が決まっているから表にすると便利

木村	20万円
坂本	30万円
桂	30万円

D<sub>1</sub>の要素

D<sub>2</sub>の要素

# 属性名とリレーション名(1/2)

表の各列やリレーションそのものに、何を表しているか  
名前をつけてやるとわかりやすい。

リレーション名(=テーブル名)

社員給与

属性名  
(=カラム名)

A<sub>1</sub>

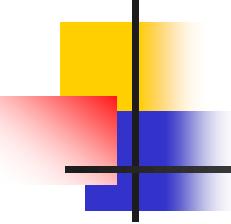
属性名  
(=カラム名)

A<sub>2</sub>

社員名	俸給月額
木村	20万円
坂本	30万円
桂	30万円

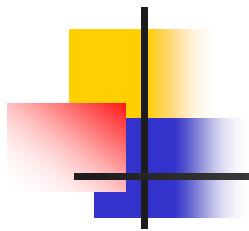
D<sub>1</sub>

D<sub>2</sub>



# 属性名とリレーション名(2/2)

- ドメイン関数
  - 属性名とドメインを結びつけるもの
  - $\text{Dom} : A_i \rightarrow D_i \quad (i=1,2,3,\dots,n)$
  - $D_i = \text{Dom}(A_i)$ 
    - $D_1 = \{x | x \text{は人名}\} = \text{Dom}(\text{社員名})$
- 属性名を使ったリレーションの表し方
  - $R(A_1, A_2, \dots, A_n) \subseteq \text{Dom}(A_1) \times \text{Dom}(A_2) \times \dots \times \text{Dom}(A_n)$
- タップルの表し方
  - $t = (a_1, a_2, \dots, a_n) \iff a_i = t[A_i]$



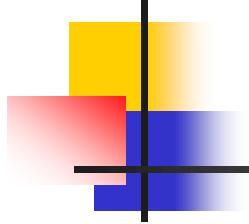
# リレーションスキーマ

スキーマはデータによらない。これを設計する！

社員給与

社員名	俸給月額
木村	20万円
坂本	30万円
桂	30万円

実際に利用するときには、データを入れておく

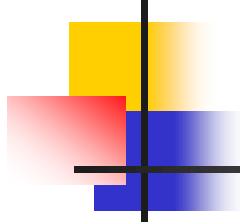


## リレーションスキーマ(2/2)

社員給与(社員名, 債給月額)

### 社員給与

社員名	俸給月額
木村	20万円
坂本	30万円
桂	30万円

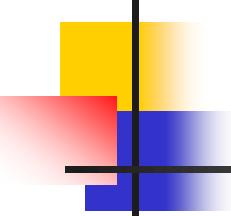


# 主キー(1/4)

- これまでのリレーションは、ただタップルを並べていただけ。

「あるデータを探し出したい！」

- リレーションのタップルを特定するための仕組みとして**主キー**がある。



# 主キー (2/4)

主キーがひとつの属性値で構成されている場合

社員マスタ(社員番号,社員名,所属部門)

社員番号	社員名	所属部門
L001	桂 小五郎	SE部
L002	坂本 龍馬	経理部
L003	桂 小五郎	経理部
L004	近藤 勇	営業部

↑  
社員番号だけで  
タップルが特定できる！

# 主キー (3/4)

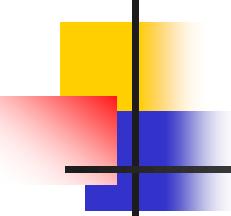
主キーが複数の属性値で構成されている場合

営業成績(製品名,担当者,売上)

製品名だけでは  
タップルは特定できない

製品名	担当者	売上
携帯電話	木村	20万円
パソコン	木村	100万円
コンボ	木村	50万円
携帯電話	坂本	10万円
パソコン	坂本	150万円
掃除機	坂本	10万円

製品名と担当者を合わせて初めて  
タップルが特定できる！

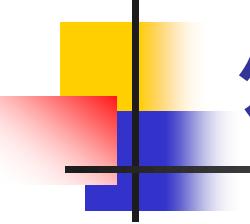


# 主キー (4/4)

- タップルを特定するため、主キーは以下の条件を備える必要がある。
  - タップルの唯一識別能力を備えていること
  - 主キーを構成する属性の値は空値(null value)ではないこと
    - 下の例では、桂さんを一意に特定できない。(空値は「値がない」ので識別能力なし)

社員番号を  
主キーとする

社員番号	社員名	所属部門
L001	桂 小五郎	SE部
L002	坂本 龍馬	経理部
-	桂 小五郎	経理部
L004	近藤 勇	営業部



# 外部キー (1/2)

## ■ さつきのリレーション(表)

社員番号	社員名	所属部門
L001	桂 小五郎	SE部
L002	坂本 龍馬	経理部
L003	桂 小五郎	経理部
L004	近藤 勇	営業部

で、各社員の所属部門が「存在しないもの」  
だったら困る。

# 外部キー (2/2)

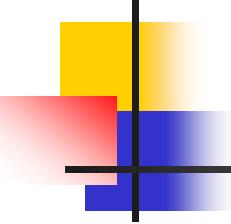
社員マスタ

社員番号	社員名	所属部門
L001	桂 小五郎	SE部
L002	坂本 龍馬	経理部
L003	桂 小五郎	経理部
L004	近藤 勇	営業部

社員マスタの所属部門の属性値が  
部門マスタの部門名に  
含まれていないとエラー  
⇒社員マスタの所属部門は  
部門マスタの外部キー

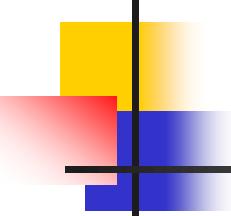
部門マスタ

部門名	所属長
SE部	L008
経理部	L019
営業部	L101



# 権限

- ユーザーに応じて、検索(SELECT)、更新(INSERT、UPDATE)、削除(DELETE)などの権限を付加したり剥奪したりすることができる。
- 例)社員リレーション(T\_SHAIN)への挿入権限をユーザーU001に付加/剥奪する。
  - GRANT INSERT ON T\_SHAIN TO U001; (付加)
  - REVOKE INSERT ON T\_SHAIN TO U001; (剥奪)



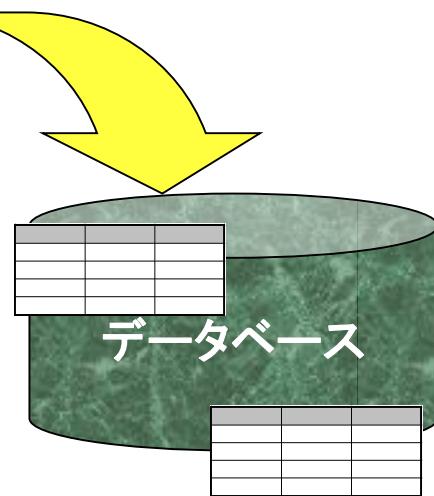
# データベーススキーマ

- データベースの構造・枠組みを定義する体系
  - データベーススキーマ名(枠組みの名前)
  - ドメイン定義(必要があればドメインに名前をつける)
  - リレーションスキーマ定義
  - ビュー定義(SELECT文の結果をリレーションに見立てたもの)
  - 表明定義
  - トリガー定義
  - 権限定義
- つまり、今日やったことをワンセットにしたもの

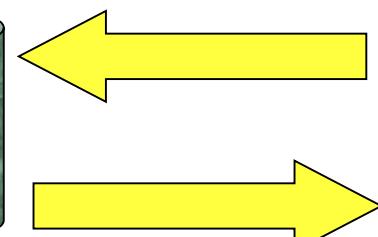
# データベースへの操作(1/3)

- 本来、データベースはデータを貯めたり自分が知りたいデータを調べるもの。

データを貯める・修正する



データを検索する



結果を返す

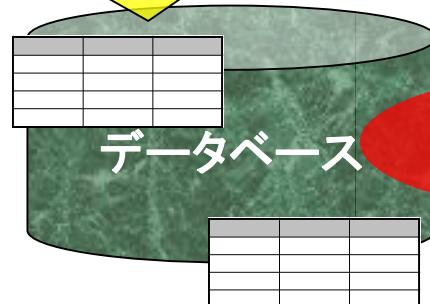


# データベースへの操作(2/3)

- 問い合わせ(Query) = タップルの検索
- 更新 = タップルの追加・変更・削除

データを貯める・修正する

更新



データを検索する

問い合わせ

結果を返す



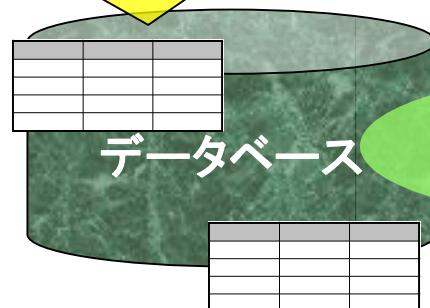
# データベースへの操作(3/3)

- 問い合わせ(Query)
- 更新

リレーショナル  
データ操作言語  
(DML)

データを貯める・修正する

Update文など



データベース

データを検索する

SELECT文

結果を返す



# 「問い合わせ」とは(1/3)

リレーション「社員」から  
「給与が50万円以上の社員名と所属を求めよ」

社員

社員番号	社員名	給与	所属
E0101	井上陽水	61万円	研究開発部
E3015	宇多田ヒカル	22万円	営業部
E0201	桑田 佳祐	57万円	営業部
E4119	島谷ひとみ	18万円	人事部
E0304	子門正人	51万円	総務部

給与が50万円以上のタップル

# 「問い合わせ」とは(2/3)

リレーション「社員」から  
「給与が50万円以上の社員名と所属を求めよ」

社員

社員番号	社員名	給与	所属
E0101	井上陽水	61万円	研究開発部
E3015	宇多田ヒカル	22万円	営業部
E0201	桑田 佳祐	57万円	営業部
E4119	島谷ひとみ	18万円	人事部
E0304	子門正人	51万円	総務部

給与が50万円以上のタップル

# 「問い合わせ」とは(3/3)

リレーション「社員」から  
「給与が50万円以上の社員名と所属を求めよ」

社員

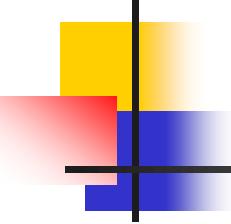
社員番号	社員名	給与	所属
E0101	井上陽水	61万円	研究開発部
E3015	宇多田ヒカル	22万円	営業部
E0201	桑田 佳祐	57万円	営業部
E4119	島谷ひとみ	18万円	人事部
E0304	子門正人	51万円	総務部

結果もリレーションの形  
をしている！



結果リレーション

社員名	所属
井上陽水	研究開発部
桑田 佳祐	営業部
子門正人	総務部



# リレーショナル代数

- 「問い合わせ」の方法を記述する「言語」
- 集合演算がベース
  - 和集合演算
  - 差集合演算
  - 共通集合演算
  - 直積集合演算
- リレーショナル代数固有の演算
  - 射影演算
  - 選択演算
  - 結合演算
  - 商演算

詳細は次週!



# データベース(第3回)

情報工学科 木村昌臣

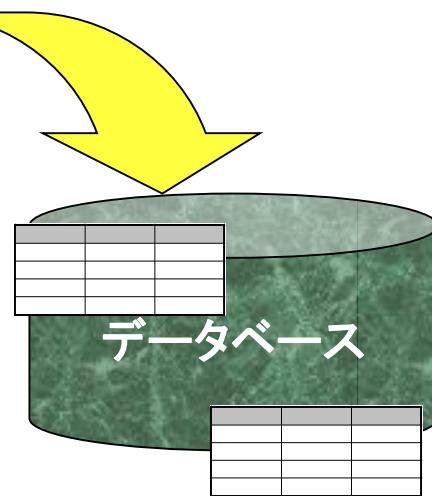


# リレーションナル代数(その1)

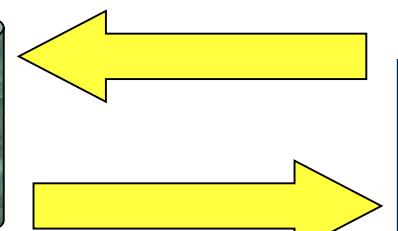
# データベースへの操作(1/3)

- 本来、データベースはデータを貯めたり自分が知りたいデータを調べるもの。

データを貯める・修正する



データを検索する



結果を返す

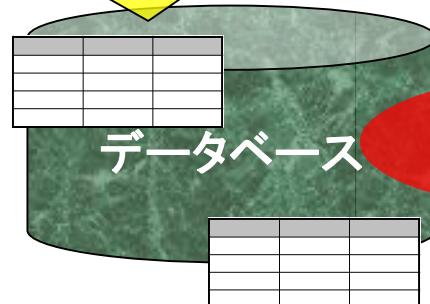


# データベースへの操作(2/3)

- 問い合わせ(Query) = タップルの検索
- 更新 = タップルの追加・変更・削除

データを貯める・修正する

更新



データを検索する

問い合わせ

結果を返す



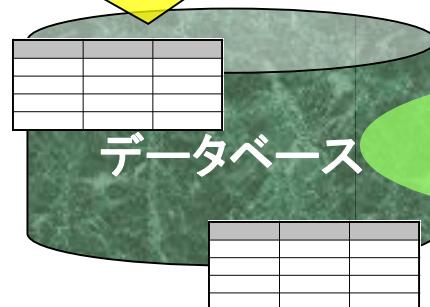
# データベースへの操作(3/3)

- 問い合わせ(Query)
- 更新

リレーショナル  
データ操作言語  
(DML)

データを貯める・修正する

Update文など



データを検索する

SELECT文

結果を返す



# 「問い合わせ」とは(1/3)

リレーション「社員」から  
「給与が50万円以上の社員名と所属を求めよ」

社員

社員番号	社員名	給与	所属
E0101	井上陽水	61万円	研究開発部
E3015	宇多田ヒカル	22万円	営業部
E0201	桑田 佳祐	57万円	営業部
E4119	島谷ひとみ	18万円	人事部
E0304	子門正人	51万円	総務部

給与が50万円以上のタップル

# 「問い合わせ」とは(2/3)

リレーション「社員」から  
「給与が50万円以上の社員名と所属を求めよ」

社員

社員番号	社員名	給与	所属
E0101	井上陽水	61万円	研究開発部
E3015	宇多田ヒカル	22万円	営業部
E0201	桑田 佳祐	57万円	営業部
E4119	島谷ひとみ	18万円	人事部
E0304	子門正人	51万円	総務部

給与が50万円以上のタップル

# 「問い合わせ」とは(3/3)

リレーション「社員」から  
「給与が50万円以上の社員名と所属を求めよ」

社員

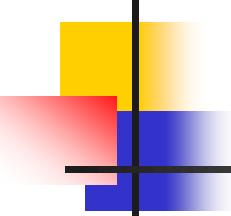
社員番号	社員名	給与	所属
E0101	井上陽水	61万円	研究開発部
E3015	宇多田ヒカル	22万円	営業部
E0201	桑田 佳祐	57万円	営業部
E4119	島谷ひとみ	18万円	人事部
E0304	子門正人	51万円	総務部

結果もリレーションの形  
をしている！



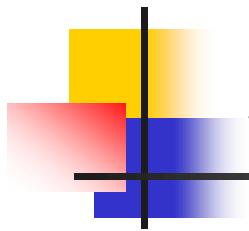
結果リレーション

社員名	所属
井上陽水	研究開発部
桑田 佳祐	営業部
子門正人	総務部



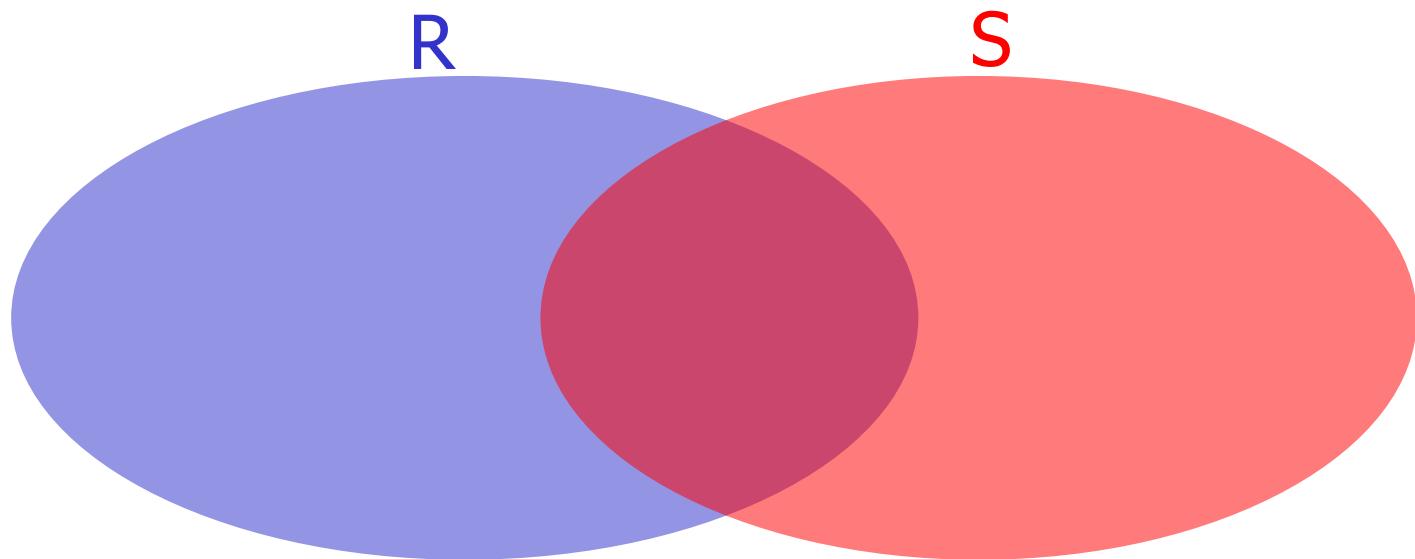
# リレーショナル代数

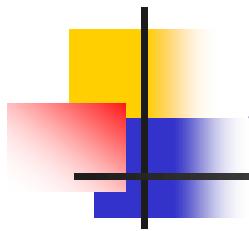
- 「問い合わせ」の方法を記述する「言語」
- 集合演算がベース
  - 和集合演算
  - 差集合演算
  - 共通集合演算
  - 直積集合演算
- リレーショナル代数固有の演算
  - 射影演算
  - 選択演算
  - 結合演算
  - 商演算



# 集合演算の復習 (1/5)

集合Rと集合Sがあるとする





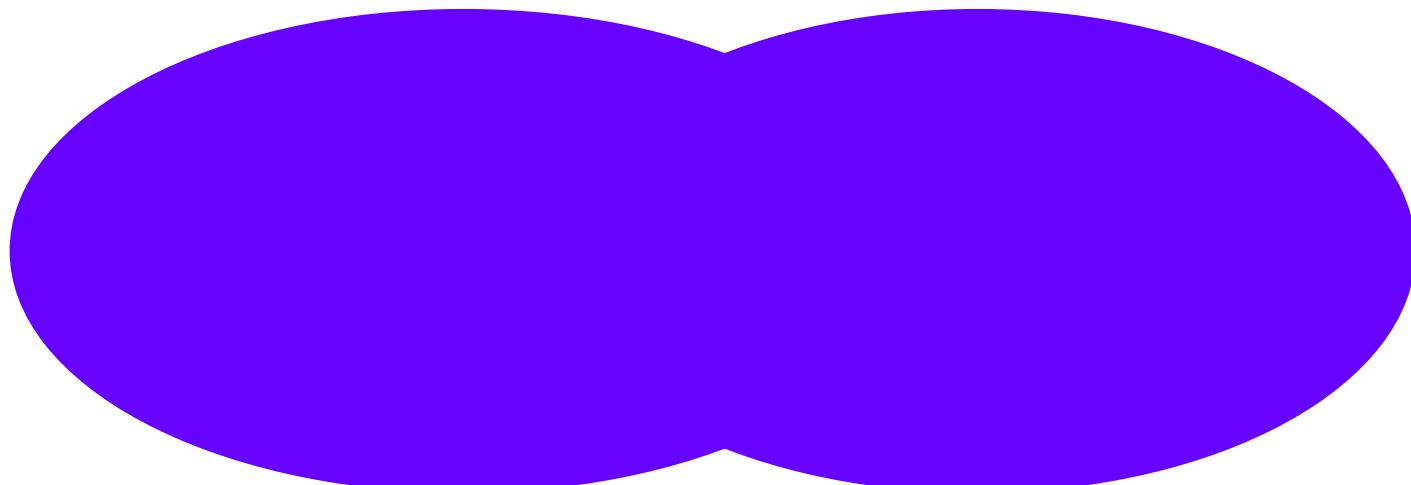
# 集合演算の復習 (2/5)

RとSの和集合R $\cup$ S

$$x \in R \cup S \Leftrightarrow x \in R \vee x \in S$$

R

S

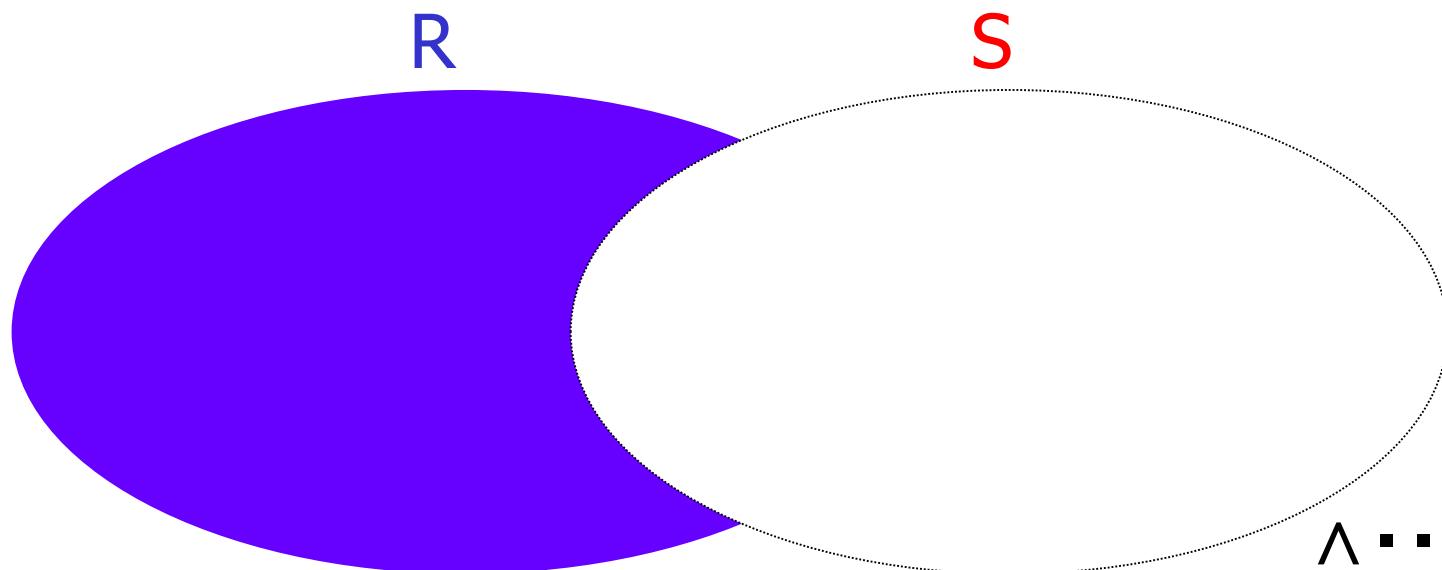


$\vee \cdots$ または

# 集合演算の復習 (3/5)

RとSの差集合  $R - S$

$$x \in R - S \Leftrightarrow x \in R \wedge \neg(x \in S)$$

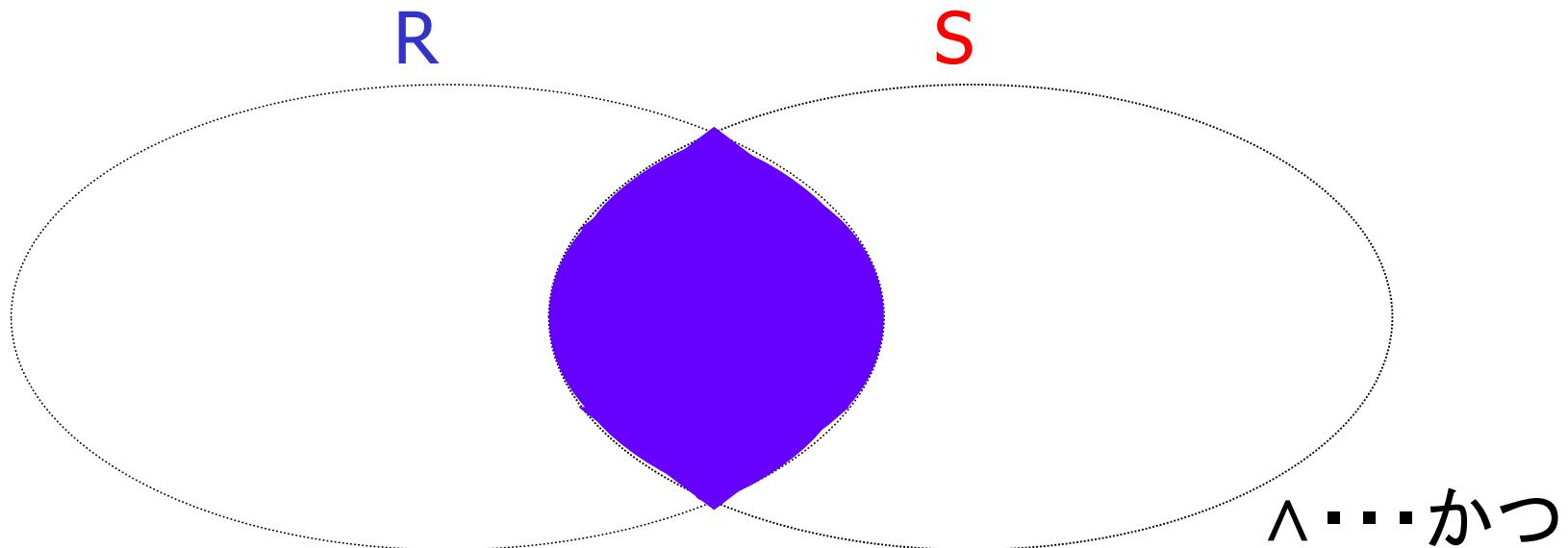


$\wedge \cdots$ かつ  
 $\neg \cdots$ 否定

# 集合演算の復習 (4/5)

RとSの共通集合  $R \cap S$

$$x \in R \cap S \Leftrightarrow x \in R \wedge x \in S$$

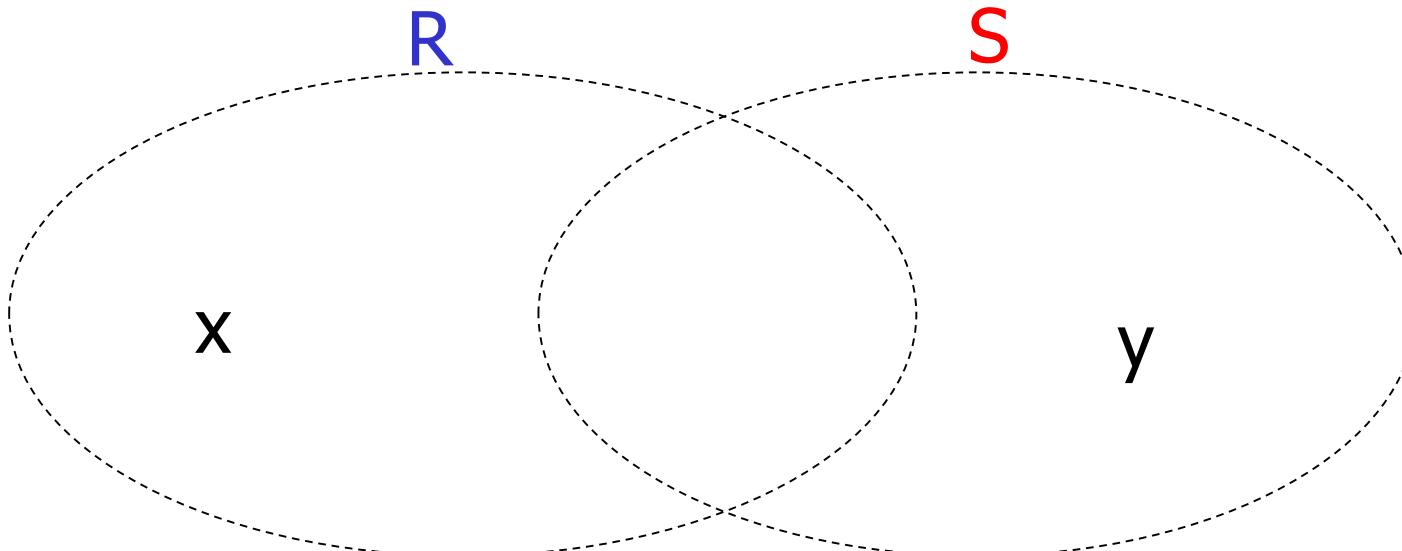


# 集合演算の復習 (5/5)

RとSの直積集合  $R \times S$

xとyの組

$$(x, y) \in R \times S \Leftrightarrow x \in R \wedge y \in S$$



# リレーションナル代数 和演算

リレーションR


リレーションS


リレーション $R \cup S$


$$R \cup S = \{t \mid t \in R \vee t \in S\}$$

ただし、  
列の数(次数)が同じこと  
RとSの各列が同じドメインにいること  
が必要

# リレーションナル代数

## 差演算

リレーションR


リレーションS


リレーションR-S


$$R-S = \{t \mid t \in R \wedge \neg(t \in S)\}$$

ただし、  
列の数(次数)が同じこと  
RとSの各列が同じドメインにいること  
が必要

# リレーションナル代数

## 共通集合演算

リレーションR



リレーションS



リレーション $R \cap S$



$$R \cap S = \{t \mid t \in R \wedge t \in S\}$$

ただし、  
列の数(次数)が同じこと  
RとSの各列が同じドメインにいること  
が必要

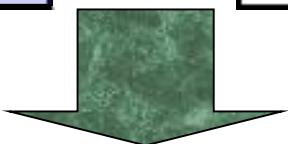
# リレーションナル代数

## 直積演算

リレーションR


リレーションS


リレーションR × S



R	S
---	---

$$R \times S = \{(t, s) \mid t \in R \wedge s \in S\}$$

ただし、タップルの数はRのタップルの数とSのタップルの数の積になる

# リレーションナル代数 直積演算（例）

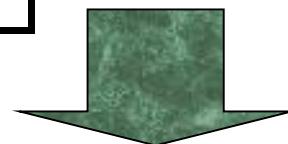
リレーションR

学部	学科
工学部	情報
工学部	建築
システム工学部	電子情報

リレーションS

キャンパス
芝浦
大宮

リレーションR × S



学部	学科	キャンパス
工学部	情報	芝浦
工学部	情報	大宮
工学部	建築	芝浦
工学部	建築	大宮
システム工学部	電子情報	芝浦
システム工学部	電子情報	大宮

$3 \times 2 = 6$  タップル

# リレーションナル代数 射影演算

射影演算=列の切り出し(縦)

社員

社員番号	社員名	給与	所属
E0101	井上陽水	61万円	研究開発部
E3015	宇多田ヒカル	22万円	営業部
E0201	桑田 佳祐	57万円	営業部
E4119	島谷ひとみ	18万円	人事部
E0304	子門正人	51万円	総務部

$$\begin{aligned} R[\text{社員名}, \text{所属}] = & \{ u \mid u \in \text{dom}(\text{社員名}) \times \text{dom}(\text{所属}) \wedge (\exists t \in \text{社員}) \\ & \wedge t[\text{社員名}] = u[\text{社員名}] \wedge t[\text{所属}] = u[\text{所属}] \} \end{aligned}$$

# リレーションナル代数 選択演算(1/5)

ある条件をつけて、タップルを選択する

社員

社員番号	社員名	給与	所属
E0101	井上陽水	61万円	研究開発部
E3015	宇多田ヒカル	22万円	営業部
E0201	桑田 佳祐	57万円	営業部
E4119	島谷ひとみ	18万円	人事部
E0304	子門正人	51万円	総務部

給与が50万円以上のタップル

# 選択演算を行うための条件 (2/5)

- θ-比較可能であること
  - $R(A_1, A_2, \dots, A_n)$ をリレーションとするとき $A_i, A_j$ が次の条件を満たしているときθ-比較可能であるという
    - $\text{dom}(A_i) = \text{dom}(A_j)$
    - 任意のタップル $t$ に対して、 $t[A_i] \theta t[A_j]$ の真偽が常に定まること
      - θの例:  $= < > \neq \leq \geq$

# θ-選択演算(3/5)

- $R(A_1, A_2, \dots, A_n)$ をリレーションとする
- $R$ の属性 $A_i$ と $A_j$  上の $\theta$ -選択を $R[A_i \theta A_j]$ と書く:

$$R[A_i \theta A_j] = \{t \mid t \in R \wedge (t[A_i] \theta t[A_j])\}$$

商品

商品番号	商品名	原価	売価
G110	刺身	400	300
G120	豆腐	90	75
G130	卵	95	100
G140	コーヒー	500	700
G150	砂糖	200	300

$R$ =商品  
 $A_i$ =原価  
 $A_j$ =売価  
 $\theta = >$

商品[原価>売価]

商品番号	商品名	原価	売価
G110	刺身	400	300
G120	豆腐	90	75

# θ-選択演算(定数と比較)(5/5)

商品[原価>300 ]

= $(\text{商品} \times \text{CONST}) [\text{商品.原価} > \text{CONST.C}] [\text{商品.商品番号}, \text{商品.商品名}, \text{商品.原価}, \text{商品.原価}]$

商品 × CONST

元のリレーション「商品」の列への射影

商品番号	商品名	原価	売価	CONST.C
G110	刺身	400	300	300
G120	豆腐	90	75	300
G130	卵	95	100	300
G140	コーヒー	500	700	300
G150	砂糖	200	300	300

商品.原価>CONST.C

# リレーションナル代数

## 結合演算(1/4)

- リレーションナルデータベースでは、リレーションの間の関係は「陰」に定義されている。
  - ネットワークデータモデルでは、データ間の関係はポインタを使って「陽」に定義されている。
  - リレーションナルデータモデルでは、二つ以上のリレーションを共通の属性(属性値)で結びつけることでそれらの関係を定義している。

# リレーションナル代数

## 結合演算(2/4)

リレーションR

		R.A			

リレーションS

S.B					

リレーションRの列AとリレーションSの列Bが $\theta$ -比較可能とする。

$$R[A\theta B]S = \{ (t,u) \mid t \in R \wedge u \in S \wedge \underline{t[A]\theta u[B]} \}$$

この部分は直積演算

$R[A\theta B]S$

		R.A	S.B		

$t[A]\theta u[B]$   
を満たす

# リレーションナル代数 結合演算(例) (3/4)

社員

社員番号	社員名	給与	所属コード
E3059	城島 茂	22万円	100
E3015	宇多田 ヒカル	22万円	200
E0201	桑田 圭祐	57万円	200
E4119	島谷 ひとみ	18万円	100
E0304	子門 正人	51万円	300

部門

部門コード	部門名
100	人事部
200	営業部
300	経理部

社員.所属と部門.部門コード  
はθ-比較可能

社員[社員.所属コード=部門.部門コード]部門

社員番号	社員名	給与	所属コード	部門コード	部門名
E3059	城島 茂	22万円	100	100	人事部
E3015	宇多田 ヒカル	22万円	200	200	営業部
E0201	桑田 圭祐	57万円	200	200	営業部
E4119	島谷 ひとみ	18万円	100	100	人事部
E0304	子門 正人	51万円	300	300	経理部

# リレーションナル代数 結合演算－自然結合演算 (4/4)

社員

社員番号	社員名	給与	所属コード
E3059	城島 茂	22万円	100
E3015	宇多田 ヒカル	22万円	200
E0201	桑田 圭祐	57万円	200
E4119	島谷 ひとみ	18万円	100
E0304	子門 正人	51万円	300

部門

部門コード	部門名
100	人事部
200	営業部
300	経理部

$\theta = “=”$ の場合

社員\*部門

社員番号	社員名	給与	所属コード	部門名
E3059	城島 茂	22万円	100	人事部
E3015	宇多田 ヒカル	22万円	200	営業部
E0201	桑田 圭祐	57万円	200	営業部
E4119	島谷 ひとみ	18万円	100	人事部
E0304	子門 正人	51万円	300	経理部

社員と部門  
の自然結合

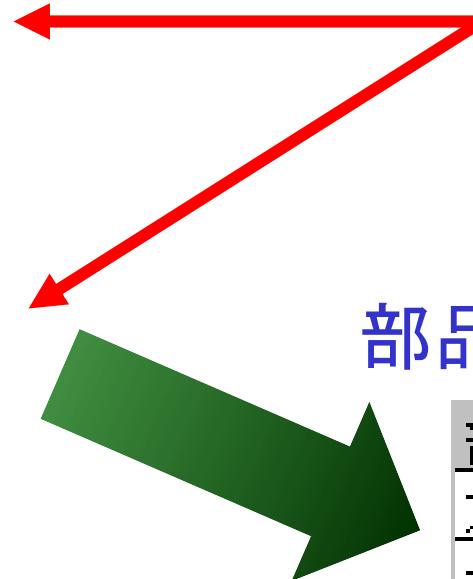
共通部分は  
片方だけ残す

# リレーションナル代数 商演算

部品供給

部品工場	部品
大宮	ナット
大宮	ボルト
芝浦	ボルト
芝浦	釘
豊洲	ナット
豊洲	ボルト
豊洲	釘

必要部品が揃う  
工場



必要部品

必要部品
ナット
ボルト

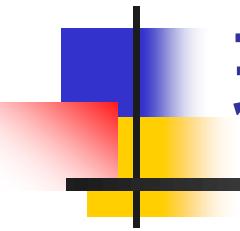
部品供給 ÷ 必要部品

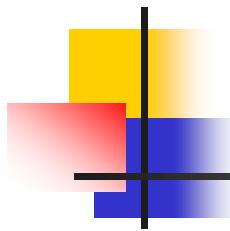
部品が揃う工場
大宮
豊洲

部品供給 ÷ 必要部品

$$= \{ t \mid t \in \text{部品供給}[\text{部品工場}] \wedge (\forall u \in \text{必要部品})((t, u) \in \text{部品供給}) \}$$

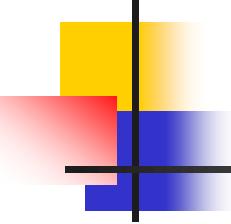
# リレーションナルデータベース設計 理論





# 第一正規形(1/2)

- リレーションのあるべき姿
- タップルの属性値(レコードのフィールド値)  
に繰り返しがあってはいけない。
- 繰り返すなら、いっそ新しいタップル(レコード)を作るべし！



# 第一正規形(2/2)

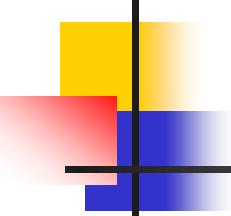
社員番号	社員名	趣味
L001	木村	ドライブ、カラオケ
L002	坂本	テニス、音楽、山登り

フィールド内で繰り返すのではなく、  
レコードとして繰り返す！

社員番号	社員名	趣味
L001	木村	ドライブ
L001	木村	カラオケ
L002	坂本	テニス
L002	坂本	音楽
L002	坂本	山登り



正規化



# 更新時異常(1/4)

- 第一正規化だけでは、実際にリレーショナルデータベースを設計・運用するときに不都合がでる場合がある。
- どんなときか？
  - タップルの挿入時
  - タップルの削除時
  - タップルの修正時

# タップル挿入時異常(2/4)

注文	主キー	顧客名	商品名	数量	単価
		A商店	テレビ	3	198,000
		Bマート	テレビ	10	198,000
		Bマート	パソコン	5	59,800
		C社	ゲーム機	1	29,800

リレーション注文は、商品名と単価の属性をもつので  
新製品である電子レンジ(単価74,800円)の単価情報を持たせたい



`t=(null, 電子レンジ, null, 74,800, null)` を挿入することになるが、  
顧客名は主キーなのでこれは挿入できない！！！

# タップル削除時異常(3/4)

注文

主キー

顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

リレーション注文から、C社のゲーム機注文記録を削除したい。



C社の注文記録を削除すると、ゲーム機の単価情報もなくなってしまう！！

# タップル修正時異常(4/4)

注文	主キー	顧客名	商品名	数量	単価
		A商店	テレビ	3	198,000
		Bマート	テレビ	10	198,000
		Bマート	パソコン	5	59,800
		C社	ゲーム機	1	29,800

テレビの単価が間違っていた。修正しなければ。



テレビの単価が変わったという单一の事象が起こっただけなのに、  
テレビ注文関連のタップルのすべて(A商店、Bマート)を修正しなければならない

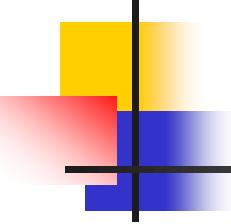
# 更新時異常が起こる原因(1/3)

注文		主キー	
顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

商品がいくらするか

顧客がどの商品をどれだけ注文したか

ひとつのリレーションに複数の事象  
が含まれている！！



# 更新時異常を解決するには(2/3)


ひとつのリレーションには  
ひとつの事象しか格納しないようにするべし

つまり……？

# 更新時異常を解決するには(3/3)

……事象ごとにリレーションを分ける

顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

顧客がどの商品を  
どれだけ注文したか

商品がいくらするか

顧客名	商品名	数量
A商店	テレビ	3
Bマート	テレビ	10
Bマート	パソコン	5
C社	ゲーム機	1

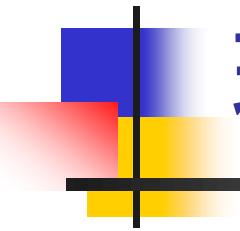
商品名	単価
テレビ	198,000
パソコン	59,800
ゲーム機	29,800

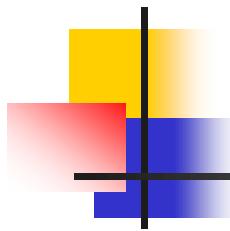


# データベース(第4回)

情報工学科 木村昌臣

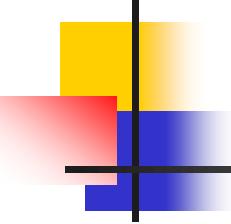
# リレーションナルデータベース設計 理論





# 第一正規形(1/2)

- リレーションのあるべき姿
- タップルの属性値(レコードのフィールド値)  
に繰り返しがあってはいけない。
- 繰り返すなら、いっそ新しいタップル(レコード)を作るべし！



# 第一正規形(2/2)

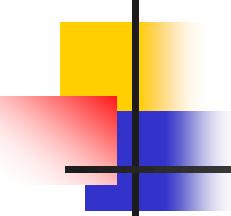
社員番号	社員名	趣味
L001	木村	ドライブ、カラオケ
L002	坂本	テニス、音楽、山登り

フィールド内で繰り返すのではなく、  
レコードとして繰り返す！

社員番号	社員名	趣味
L001	木村	ドライブ
L001	木村	カラオケ
L002	坂本	テニス
L002	坂本	音楽
L002	坂本	山登り



正規化



# 更新時異常(1/4)

- 第一正規化だけでは、実際にリレーショナルデータベースを設計・運用するときに不都合がでる場合がある。
- どんなときか？
  - タップルの挿入時
  - タップルの削除時
  - タップルの修正時

# タップル挿入時異常(2/4)

注文	主キー	顧客名	商品名	数量	単価
		A商店	テレビ	3	198,000
		Bマート	テレビ	10	198,000
		Bマート	パソコン	5	59,800
		C社	ゲーム機	1	29,800

リレーション注文は、商品名と単価の属性をもつので  
新製品である電子レンジ(単価74,800円)の単価情報を持たせたい



`t=(null, 電子レンジ, null, 74,800, null)` を挿入することになるが、  
顧客名は主キーなのでこれは挿入できない！！！

# タップル削除時異常(3/4)

注文

主キー

顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

リレーション注文から、C社のゲーム機注文記録を削除したい。



C社の注文記録を削除すると、ゲーム機の単価情報もなくなってしまう！！

# タップル修正時異常(4/4)

注文	主キー	顧客名	商品名	数量	単価
		A商店	テレビ	3	198,000
		Bマート	テレビ	10	198,000
		Bマート	パソコン	5	59,800
		C社	ゲーム機	1	29,800

テレビの単価が間違っていた。修正しなければ。



テレビの単価が変わったという单一の事象が起こっただけなのに、  
テレビ注文関連のタップルのすべて(A商店、Bマート)を修正しなければならない

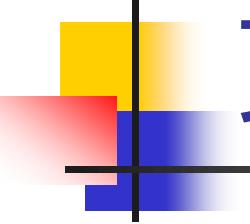
# 更新時異常が起こる原因(1/3)

注文		主キー	
顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

商品がいくらするか

顧客がどの商品をどれだけ注文したか

ひとつのリレーションに複数の事象  
が含まれている！！



# 更新時異常を解決するには(2/3)


ひとつのリレーションには  
ひとつの事象しか格納しないようにするべし

つまり……？

# 更新時異常を解決するには(3/3)

……事象ごとにリレーションを分ける

顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

顧客がどの商品を  
どれだけ注文したか

商品がいくらするか

顧客名	商品名	数量
A商店	テレビ	3
Bマート	テレビ	10
Bマート	パソコン	5
C社	ゲーム機	1

商品名	単価
テレビ	198,000
パソコン	59,800
ゲーム機	29,800

# 情報無損失分解 (1/4)

事象ごとにリレーションを分けても  
もとのリレーションを再現できる

注文

顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

自然結合

注文[顧客名,商品名,数量]

顧客名	商品名	数量
A商店	テレビ	3
Bマート	テレビ	10
Bマート	パソコン	5
C社	ゲーム機	1

注文[商品名,単価]

商品名	単価
テレビ	198,000
パソコン	59,800
ゲーム機	29,800

商品がいくらするか

顧客がどの商品を  
どれだけ注文したか

# 自然結合演算 (2/4)

社員

社員番号	社員名	給与	所属コード
E3059	城島 茂	22万円	100
E3015	宇多田 ヒカル	22万円	200
E0201	桑田 圭祐	57万円	200
E4119	島谷 ひとみ	18万円	100
E0304	子門 正人	51万円	300

部門

部門コード	部門名
100	人事部
200	営業部
300	経理部

$\theta = “=”$ の場合

社員\*部門

社員番号	社員名	給与	所属コード	部門名
E3059	城島 茂	22万円	100	人事部
E3015	宇多田 ヒカル	22万円	200	営業部
E0201	桑田 圭祐	57万円	200	営業部
E4119	島谷 ひとみ	18万円	100	人事部
E0304	子門 正人	51万円	300	経理部

社員と部門  
の自然結合

共通部分は  
片方だけ残す

# 情報無損失分解 (3/4)

リレーションを分解することにより、  
更新時異常が解決

注文[商品名,単価]

商品名	単価
テレビ	198,000
パソコン	59,800
ゲーム機	29,800

商品がいくらするか

注文[顧客名,商品名,数量]

顧客名	商品名	数量
A商店	テレビ	3
Bマート	テレビ	10
Bマート	パソコン	5
C社	ゲーム機	1

顧客がどの商品をどれだけ注文したか

1. タップル挿入時異常→**単価情報を持たせたい**→左側のリレーションを更新 解決!
2. タップル削除時異常→**注文記録を削除すると単価情報が消える**  
→右側のリレーションのタップルのみ削除 解決!
3. タップル修正時異常→**单一の事象が起こっただけなのに関連タップルを全部修正しなければならない**  
→左側のリレーションを修正 解決!

# 情報無損失分解(一般論)(4/4)

勝手に分解するととのリレーションが再現できない

R	顧客名	商品名	数量	単価
A商店	テレビ	3	198,000	
Bマート	テレビ	10	198,000	
Bマート	パソコン	5	59,800	
C社	ゲーム機	1	29,800	

R1

顧客名	商品名
A商店	テレビ
Bマート	テレビ
Bマート	パソコン
C社	ゲーム機

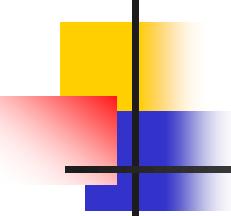
$$R1 * R2 \neq R$$

商品名	数量	単価
テレビ	3	198,000
テレビ	10	198,000
パソコン	5	59,800
ゲーム機	1	29,800

R2

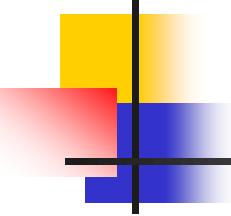
顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
A商店	テレビ	10	198,000
Bマート	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

Rにない  
タップルができている  
(結合の罠)



# 多値従属性 (1/4)

- 今までの情報無損失分解は、タップルレベルでの議論
- 本来、情報無損失分解は、リレーションスキーマレベル（要するにテーブル設計）で議論されるべき

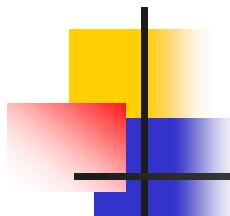


## 多値従属性 (2/4)

リレーションスキーマ  $R(A_1, A_2, \dots, A_l, B_1, B_2, \dots, B_m, C_1, C_2, \dots, C_n)$  について  
 $R$  に多値従属性(MVD)が存在するとは

**B<sub>1</sub>,B<sub>2</sub>,...,B<sub>m</sub>の情報(の集合)が  
C<sub>1</sub>,C<sub>2</sub>,..,C<sub>n</sub>に属する情報に依存せず  
A<sub>1</sub>,A<sub>2</sub>,...,A<sub>l</sub>の情報のみで決定すること**

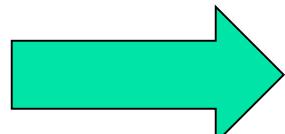
$A_1, A_2, \dots, A_l \rightarrow\rightarrow B_1, B_2, \dots, B_m$



# 多値従属性(3/4) 具体例-1

社員	子供	本人スキル
高田健二	タロウ	ネットワークエンジニア
高田健二	ユキコ	ネットワークエンジニア
吉川裕子	フウタ	データベースエンジニア

社員が決まれば、その社員の子供の集合が求まる



子供には社員に対する多値従属性がある  
 $\text{社員} \rightarrow\!\!\!-\!\!\!-\!\!\! \rightarrow \text{子供}$

# 多値従属性(4/4) 具体例-2

商品がいくら  
するか

↓  
単価は商品名に  
多値従属

↓  
だれがいくつ  
買ったかに  
依存しない

注文

顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

商品

商品名	単価
テレビ	198,000
パソコン	59,800
ゲーム機	29,800

顧客がどの商品を  
何個いくら分注文したか

↓  
商品名と数量は  
顧客名に多値従属

↓  
商品の単価に  
依存しない

発注

顧客名	商品名	数量
A商店	テレビ	3
Bマート	テレビ	10
Bマート	パソコン	5
C社	ゲーム機	1

# 関数従属性 (1/4)

## 注文

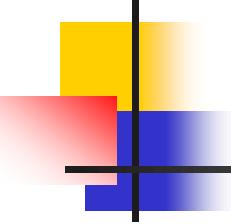
顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

商品名がひとつ決まると単価が  
ひとつ決まる!  
(多値従属性の特別な場合)



単価は、商品名にしか依存せず、  
商品名が決まると単価がただひとつ決まる

商品名	単価
テレビ	198,000
パソコン	59,800
ゲーム機	29,800



# 関数従属性(定義) (2/4)

リレーションナルスキーマ  $R(A_1, \dots, A_l, B_1, \dots, B_m, C_1, \dots, C_n)$  に  
関数従属性

$$A_1 A_2 \dots A_l \rightarrow B_1 B_2 \dots B_m$$

が存在するとは、次の条件が成立することをいう。

$R$  を  $R$  のインスタンスとするとき、

$$(\forall t, t' \in R)(t[A_1 A_2 \dots A_l] = t'[A_1 A_2 \dots A_l] \Rightarrow t[B_1 B_2 \dots B_m] = t'[B_1 B_2 \dots B_m])$$

# 候補キー

- リレーションスキーマ  $R(A_1, A_2, \dots, A_n)$  の属性集合  $K$  が候補キーであるとは、以下を満たすことをいう。
  - $R$  を  $R$  の任意のインスタンスとする
    - ①  $(\forall t, t' \in R)(t[K] = t'[K] \Rightarrow t = t')$
    - ②  $K$  のどのような真部分集合  $H$  についても ① は成り立たない

## 発注

顧客名	商品名	数量	金額
A商店	テレビ	3	594,000
Bマート	テレビ	10	1,980,000
Bマート	パソコン	5	299,000
C社	ゲーム機	1	29,800

$K = \{\text{顧客名}, \text{商品名}\}$

属性集合: 属性名のまとめ・集合

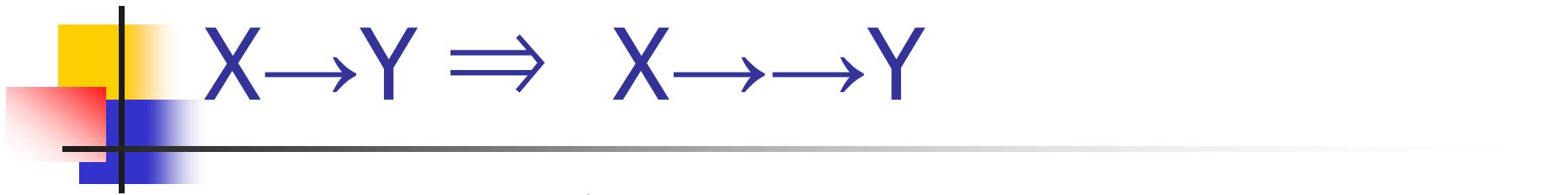
# 完全関数従属性

- 関数従属性  $X \rightarrow Y$  で、 $X'$ を $X$ の真部分集合とすると、 $X' \rightarrow Y$  が成立しない場合、YはXに完全関数従属しているという。

## 発注

顧客名	商品名	数量	金額
A商店	テレビ	3	594,000
Bマート	テレビ	10	1,980,000
Bマート	パソコン	5	299,000
C社	ゲーム機	1	29,800

{顧客名, 商品名} → {数量} は成り立つが、  
{顧客名} → {数量} {商品名} → {数量} は成り立たない。

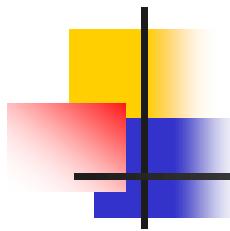

$$X \rightarrow Y \Rightarrow X \rightarrow \rightarrow Y$$

- リレーションスキーマ  $R$  で、  
 $X \rightarrow Y$  なら  $X \rightarrow \rightarrow Y$  であるが、逆は成り立たない。
  - なぜなら、「 $X \rightarrow Y$ 」は  $X$  がきまると  $Y$  が一意にきまるという特殊な「多値従属性」であるから。
  - 逆に、一般に「多値従属性」があっても、 $X$  がきまったくいつって  $Y$  はひとつには決まらない。
- よって、関数従属性は多値従属性の十分条件
  - よって、 $A_1A_2\dots A_l \rightarrow B_1B_2\dots B_m$  はリレーションスキーマ  $R(\{A\}, \{B\}, \{C\})$  を情報無損失分解するための十分条件である。



# データベース(第5回)

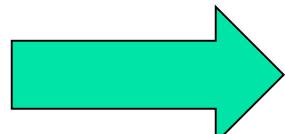
情報工学科 木村昌臣



## 【復習】多値従属性

社員	子供	本人スキル
高田健二	タロウ	ネットワークエンジニア
高田健二	ユキコ	ネットワークエンジニア
吉川裕子	フウタ	データベースエンジニア

社員が決まれば、その社員の子供の集合が求まる



子供には社員に対する多値従属性がある  
 $\text{社員} \rightarrow\!\!\! \rightarrow \text{子供}$

# 【復習】関数従属性

## 注文

顧客名	商品名	数量	単価
A商店	テレビ	3	198,000
Bマート	テレビ	10	198,000
Bマート	パソコン	5	59,800
C社	ゲーム機	1	29,800

商品名がひとつ決まると単価が  
ひとつ決まる!  
(多値従属性の特別な場合)



単価は、商品名にしか依存せず、  
商品名が決まると単価がただひとつ決まる

商品名	単価
テレビ	198,000
パソコン	59,800
ゲーム機	29,800

# 【復習】完全関数従属性

- 関数従属性  $X \rightarrow Y$  で、 $X'$ を $X$ の真部分集合とすると、 $X' \rightarrow Y$  が成立しない場合、 $Y$ は $X$ に完全関数従属しているという。

## 発注

顧客名	商品名	数量	金額
A商店	テレビ	3	594,000
Bマート	テレビ	10	1,980,000
Bマート	パソコン	5	299,000
C社	ゲーム機	1	29,800

{顧客名, 商品名} → {数量} は成り立つが、  
{顧客名} → {数量} {商品名} → {数量} は成り立たない。

# 【復習】候補キー

- リレーションスキーマ  $R(A_1, A_2, \dots, A_n)$  の属性集合  $K$  が候補キーであるとは、以下を満たすことをいう。
  - $R$  を  $R$  の任意のインスタンスとする
    - ①  $(\forall t, t' \in R)(t[K] = t'[K] \Rightarrow t = t')$
    - ②  $K$  のどのような真部分集合  $H$  についても ① は成り立たない

## 発注

顧客名	商品名	数量	金額
A商店	テレビ	3	594,000
Bマート	テレビ	10	1,980,000
Bマート	パソコン	5	299,000
C社	ゲーム機	1	29,800

$K = \{\text{顧客名}, \text{商品名}\}$



# 第二正規形

# 第2正規形

- 更新操作を考えたとき、第1正規形では更新時異常の発生が防げない

高次の正規化を行う必要あり

第2正規形

リレーションスキーマ **R** が第2正規形であるとは以下の二つの条件を満たすときを言う

1. **R** は第1正規形である
2. **R** のすべての非キー属性は **R** の各候補キーに完全関数従属している

非キー属性: 候補キーに含まれていない属性

# 第2正規形

## 注文

顧客名	商品名	数量	単価	金額
A商店	テレビ	3	198,000	594,000
Bマート	テレビ	10	198,000	1,980,000
Bマート	パソコン	5	59,800	299,000
C社	ゲーム機	1	29,800	29,800

{顧客名,商品名} → 数量

{顧客名,商品名} → 金額

単価に関する完全関数従属は

{商品名} → 単価

は完全関数従属しているが、

定義の(2)に反する！  
⇒リレーション注文は  
第2正規形ではない！！

# 第2正規形

注文

顧客名	商品名	数量	単価	金額
A商店	テレビ	3	198,000	594,000
Bマート	テレビ	10	198,000	1,980,000
Bマート	パソコン	5	59,800	299,000
C社	ゲーム機	1	29,800	29,800

{顧客名,商品名} → 数量  
{顧客名,商品名} → 金額  
は完全関数従属している

第2正規形

単価に関する  
完全関数従属は  
{商品名} → 単価

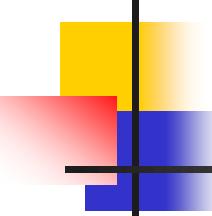
顧客名	商品名	数量	金額
A商店	テレビ	3	594,000
Bマート	テレビ	10	1,980,000
Bマート	パソコン	5	299,000
C社	ゲーム機	1	29,800

商品名	単価
テレビ	198,000
パソコン	59,800
ゲーム機	29,800

情報無損失分解される！！



# 第三正規形



# 関数従属性の推移律

## 住所録

氏名	住所	郵便番号	市外局番
浜崎あゆみ	愛知県 名古屋市千種区	464-0852	052
松浦亜弥	和歌山県 和歌山市	640-8322	073
後藤真希	神奈川県 横浜市鶴見区	230-0024	045
藤原紀香	兵庫県 神戸市東灘区	658-0083	078
伊藤美咲	福岡県 久留米市	839-0843	0942

1. 氏名 → 住所
2. 住所 → 郵便番号 という関数従属性がある。

このことから、

3. 氏名 → 郵便番号 という関数従属性が導き出される。

# 関数従属性の推移律

## 住所録

氏名	住所	郵便番号	市外局番
浜崎あゆみ	愛知県 名古屋市千種区	464-0852	052
松浦亜弥	和歌山県 和歌山市	640-8322	073
後藤真希	神奈川県 横浜市鶴見区	230-0024	045
藤原紀香	兵庫県 神戸市東灘区	658-0083	078
伊藤美咲	福岡県 久留米市	839-0843	0942

一般的には、

$A \rightarrow B$ かつ  $B \rightarrow C$ ならば  $A \rightarrow C$   
(推移律)

関数従属性を二つ組み合わせて、  
新たな関数従属性が求められた！

# 第3正規形

- 実は、第2正規形ではすべての更新時異常を解決できない

主キー

社員番号	社員名	給与	所属	勤務地
0650	内山理名	30	人事部	幕張
1508	反町隆史	40	営業部	丸の内
0231	竹内結子	35	営業部	丸の内
2034	稻川淳二	50	人事部	幕張
2100	優香	25	経理部	日本橋

- 新しい所属と勤務地の情報 (null, null, null, 中部営業部, 静岡) が挿入できない
- 優香さんが退職したら、経理部が日本橋にあるという情報が消えてしまう
- 営業部が丸の内から品川へ移ったとすると反町さん、竹内さんの二つのタップルを修正しなければならない(営業部全員の勤務地を修正しなければならない)

# 第3正規形

- 実は、第2正規形ではすべての更新時異常を解決できない

主キー

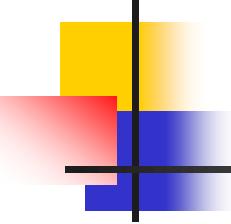
社員番号	社員名	給与	所属	勤務地
0650	内山理名	30	人事部	幕張
1508	反町隆史	40	営業部	丸の内
0231	竹内結子	35	営業部	丸の内
2034	稻川淳二	50	人事部	幕張
2100	優香	25	経理部	日本橋

理由: {社員番号} → {勤務地} は、

{社員番号} → {所属} かつ {所属} → {勤務地} という関数従属性の推移律から得られた関数従属性であるから

対策

推移的に関数従属性している部分は別リレーションに分ける



# 第3正規形

## 第3正規形

リレーションスキーマ **R** が第3正規形であるとは  
以下の二つの条件を満たすときを言う

1. **R** は第2正規形である
2. **R** のすべての非キー属性は  
**R** のどの候補キーにも推移的に関数従属していない

非キー属性: 候補キーに含まれていない属性

# 第3正規形

社員番号	社員名	給与	所属	勤務地
0650	内山理名	30	人事部	幕張
1508	反町隆史	40	営業部	丸の内
0231	竹内結子	35	営業部	丸の内
2034	稻川淳二	50	人事部	幕張
2100	優香	25	経理部	日本橋

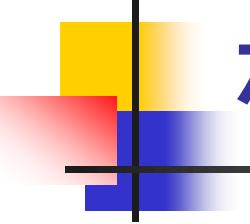
第3正規形

{社員番号}→{所属}

{所属}→{勤務地}

社員番号	社員名	給与	所属
0650	内山理名	30	人事部
1508	反町隆史	40	営業部
0231	竹内結子	35	営業部
2034	稻川淳二	50	人事部
2100	優香	25	経理部

所属	勤務地
人事部	幕張
営業部	丸の内
経理部	日本橋



# ボイス-コッド正規形

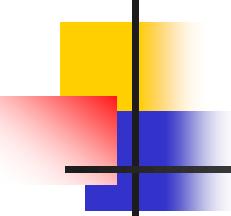
## ボイス-コッド正規形

リレーションスキーマ **R** がボイス-コッド正規形であるとは  
以下の条件を満たすときを言う。  $X \rightarrow Y$  を **R** の関数従属性とする

1.  $X \rightarrow Y$  は自明な関数従属性であるか、または
2.  $X$  は **R** のスーパーキーである

自明な関数従属性  
スーパーキー

: 候補キーとそれ以外の属性集合の間の関数従属性  
: 候補キーを含む属性集合



# 第4正規形

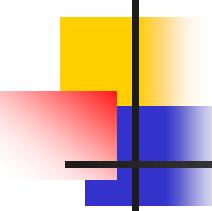
- 第3正規形までは関数従属性での分解
- 多値従属性での分解が第4正規形

## 第4正規形

リレーションスキーマ  $R$  が第4正規形であるとは  
以下の条件を満たすときを言う。

$X \rightarrow\rightarrow Y$  を  $R$  の多値従属性とすると

1.  $X \rightarrow\rightarrow Y$  は自明な多値従属性であるか、または
2.  $X$  は  $R$  のスーパーキーである



# 第5正規形

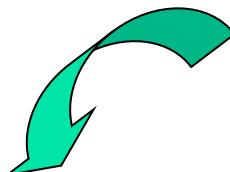
- 第4正規形でも実は更新時異常が起こりうる
- 下のリレーションのようにすべての属性が主キーになる場合
  - 大宮工場が釘を供給開始。でも、タップル(大宮工場,釘, null)は挿入できない
  - タップル(大宮工場,ボルト, × × 工業)を削除すると、大宮工場がボルトを作っているという情報が残らない

供給元	部品	供給先
芝浦工場	ネジ	○○電機
芝浦工場	ネジ	× × 工業
芝浦工場	ボルト	× × 工業
大宮工場	ボルト	× × 工業

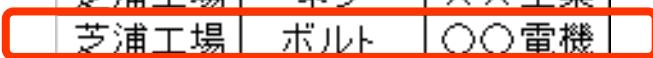
# 第5正規形

- しかし、多値従属性がないため、今までどおりの方法では情報無損失分解はできない。

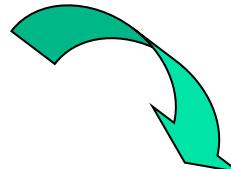
供給元	部品	供給先
芝浦工場	ネジ	○○電機
芝浦工場	ネジ	××工業
芝浦工場	ボルト	××工業
大宮工場	ボルト	××工業



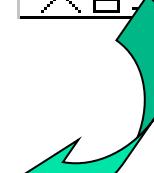
供給元	供給先
芝浦工場	○○電機
芝浦工場	××工業
大宮工場	××工業



供給元	部品	供給先
芝浦工場	ネジ	○○電機
芝浦工場	ネジ	××工業
芝浦工場	ボルト	○○電機
芝浦工場	ボルト	××工業
大宮工場	ボルト	××工業



供給元	部品
芝浦工場	ネジ
芝浦工場	ボルト
大宮工場	ボルト



供給元	部品
芝浦工場	ネジ
芝浦工場	ボルト
大宮工場	ボルト

よけいなタップルができるてしまう



# SQL概説

# リレーションナルデータベース言語の標準化

IBM サンノゼ研 SystemR

SEQUEL

UCB INGRES

QUEL

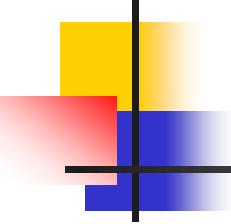
IBM Watson研

QBE

★Query by Example

ANSI/ISO/JISにて標準化

SQL

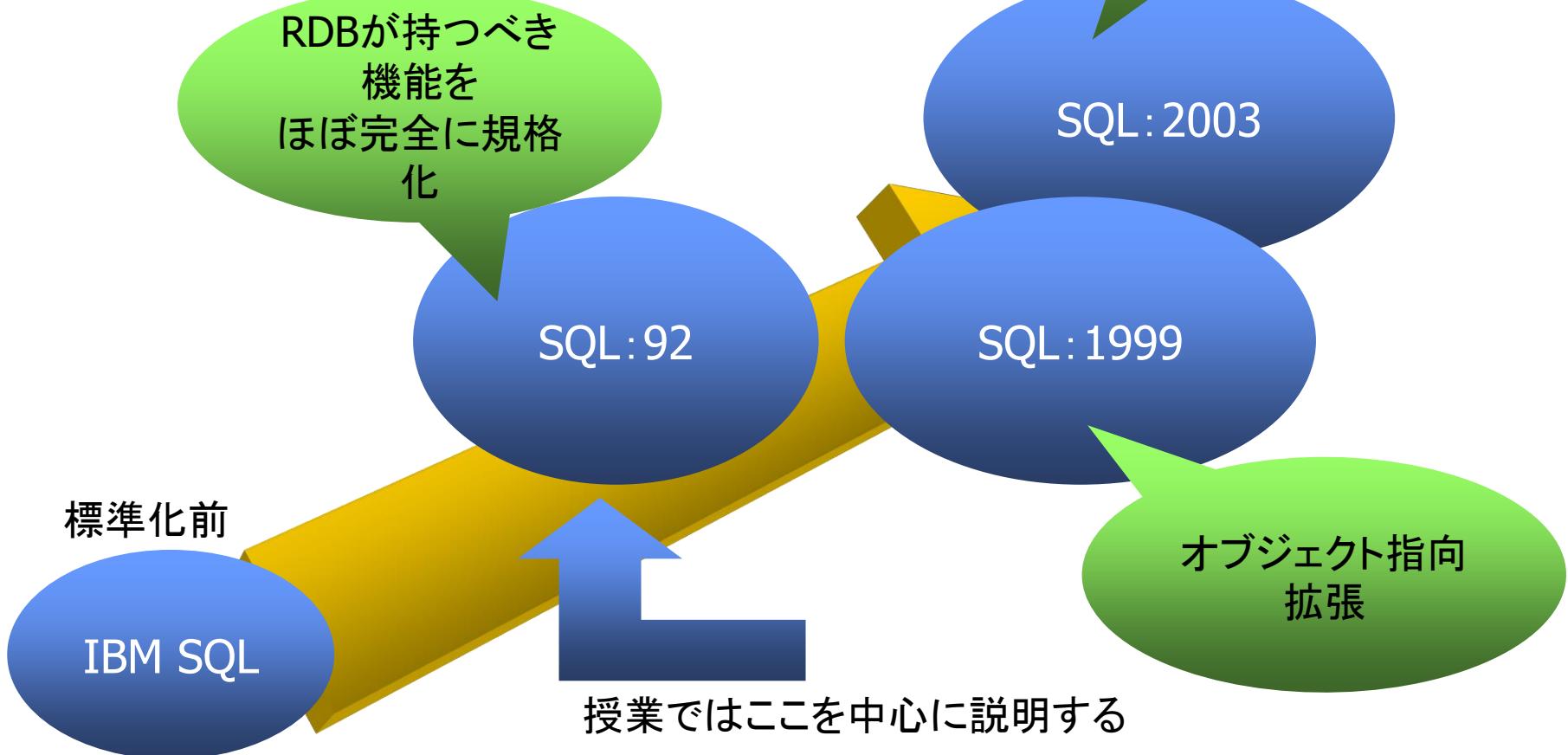


# 標準化されていないと…

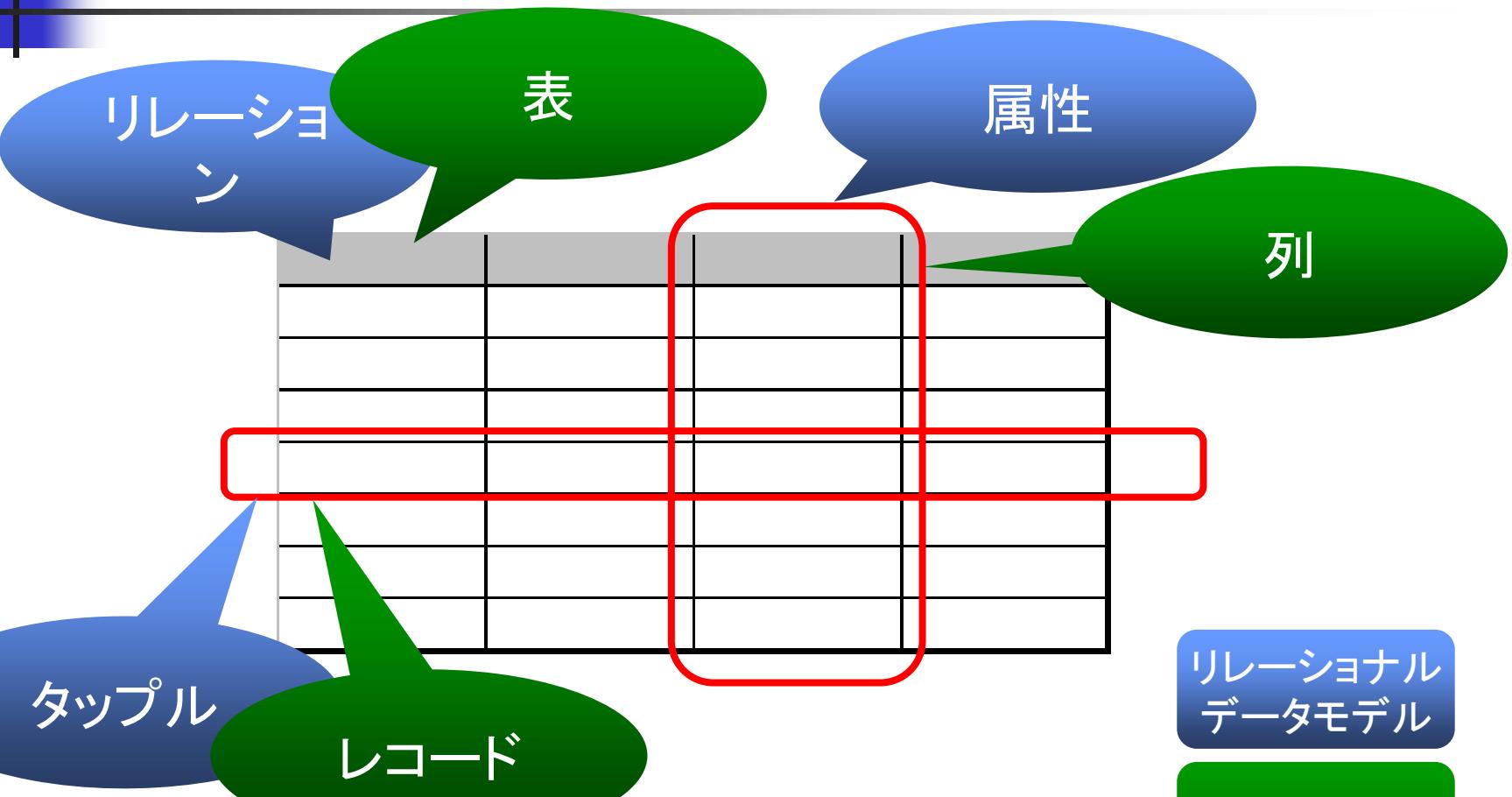
- あるデータベースについて習熟しても、別のデータベースを扱うときにまた勉強しなおさなければならぬ
- DBMSを変えたら、データベースを扱うソフトウェアも書き換えなければならない

など、困ることが多い

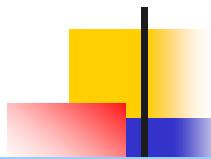
# SQLの進化



# RDBとリレーショナルデータモデルとの比較



タップル=集合の元=重複は許されない  
レコード=表の一行分=重複はあってもよい



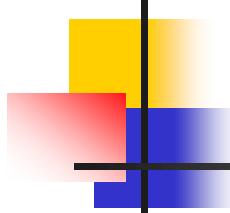
# SQLの例

テーブルT\_EMPから部署コードがD001な社員名を取り出す

```
SELECT 社員名  
FROM   T_EMP  
WHERE  部署コード='D001'
```

T\_EMP

社員番号	社員名	部署コード
E00001	志村けん	D001
E00002	加藤茶	D001
E00003	劇団ひとり	D003
E00004	高木ブー	D004
E00005	仲本工事	D003
E00006	いかりや長介	D006
E00007	妻夫木聰	D003
E00008	矢田亜希子	D001
E00009	加藤ローサ	D002



# SQLはプログラミング言語

- C言語やFORTRANは手続き型言語
  - 手続き・制御構造などを駆使してプログラムを作る
    - 「どういう方法で処理するか」を記述
- SQLは非手続き型言語
  - 制御構造は基本的に記述せず、レコードを指定する条件などを記述する
    - 「何を抽出するか」を記述する



# データベース(第5回)

情報工学科

木村昌臣

# リレーションナルデータベース言語の標準化

IBM サンノゼ研 SystemR

SEQUEL

UCB INGRES

QUEL

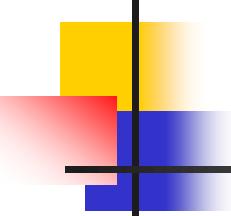
IBM Watson研

QBE

★Query by Example

ANSI/ISO/JISにて標準化

SQL

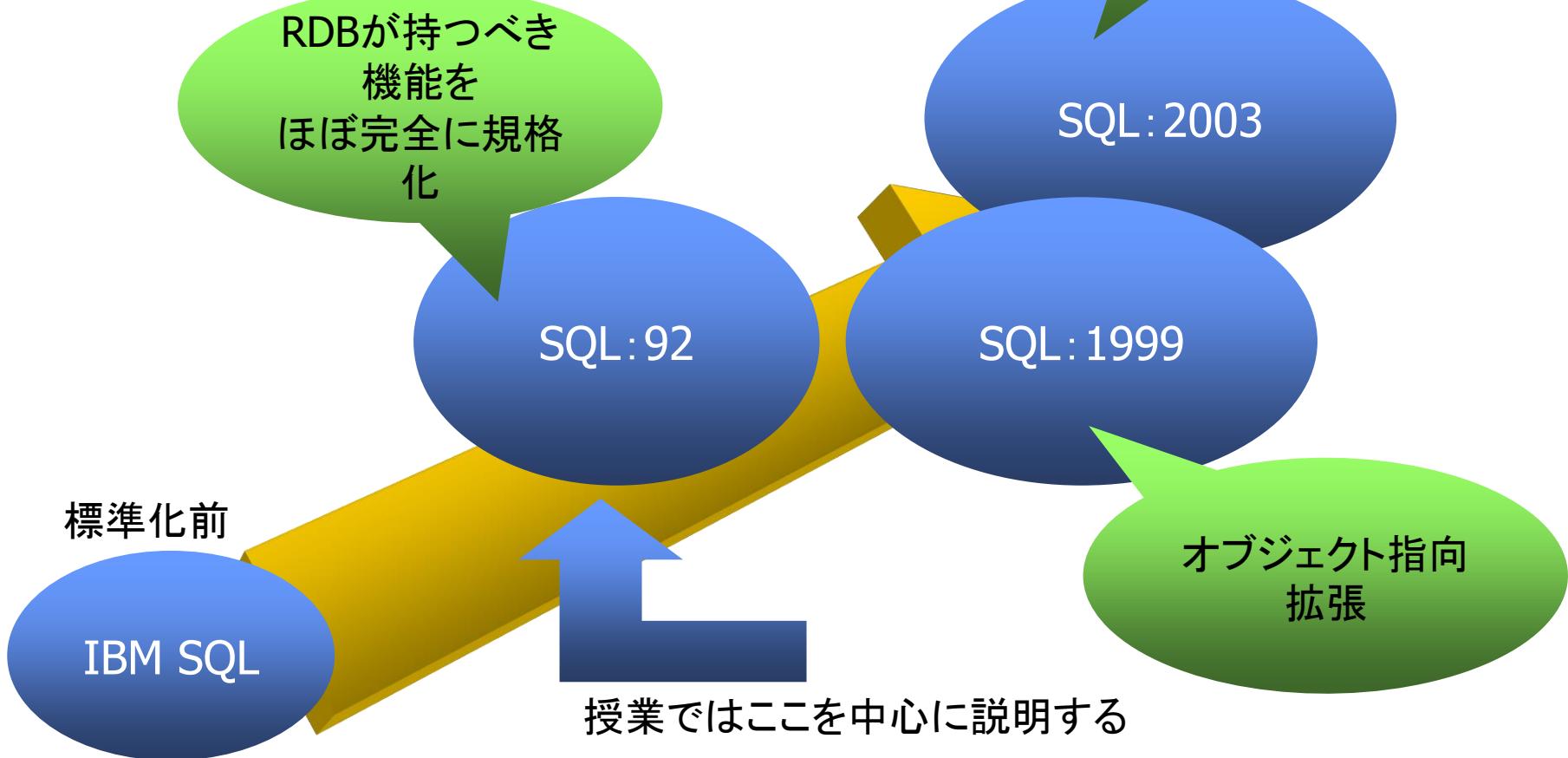


# 標準化されていないと…

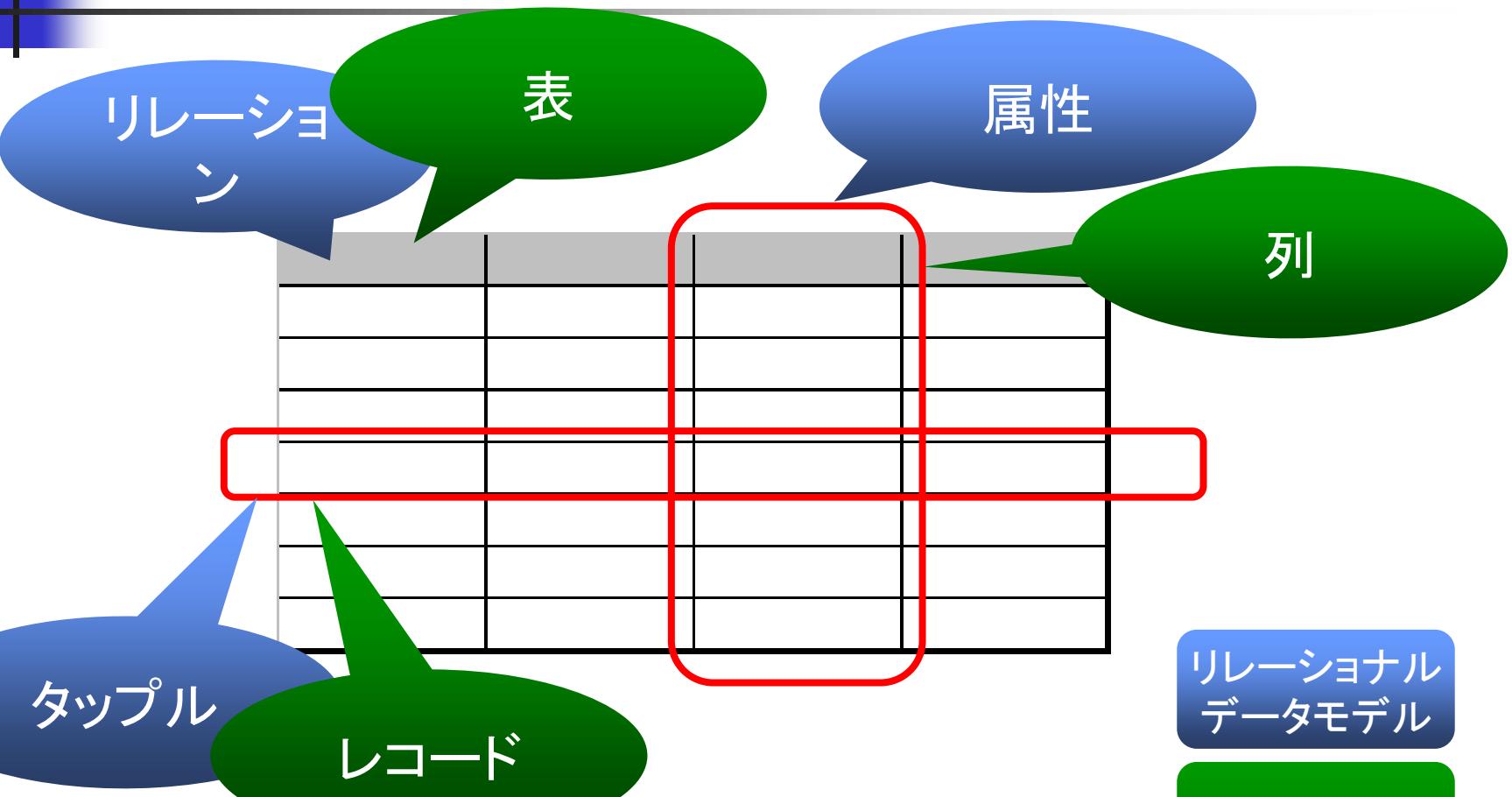
- あるデータベースについて習熟しても、別のデータベースを扱うときにまた勉強しなおさなければならぬ
- DBMSを変えたら、データベースを扱うソフトウェアも書き換えなければならない

など、困ることが多い

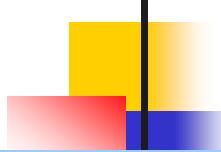
# SQLの進化



# RDBとリレーショナルデータモデルとの比較



タップル=集合の元=重複は許されない  
レコード=表の一行分=重複はあってもよい



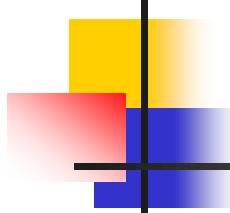
# SQLの例

テーブルT\_EMPから部署コードがD001な社員名を取り出す

```
SELECT 社員名  
FROM   T_EMP  
WHERE  部署コード='D001'
```

T\_EMP

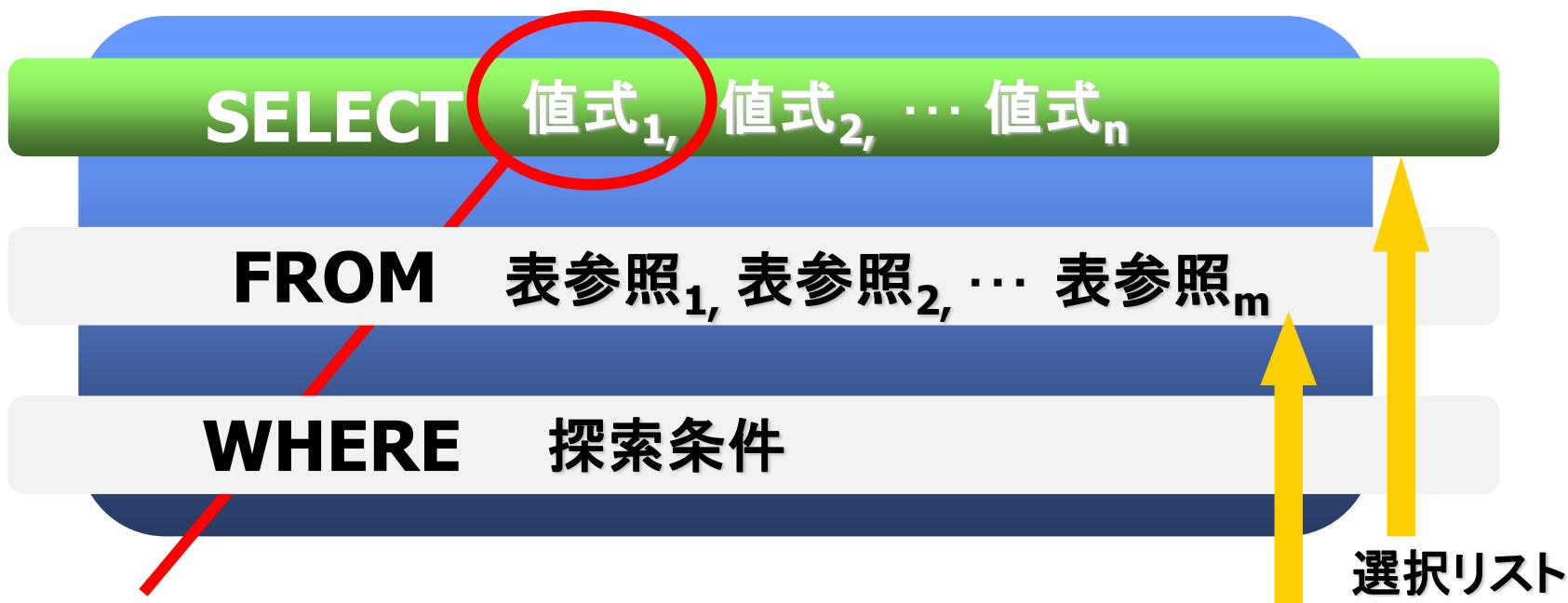
社員番号	社員名	部署コード
E00001	志村けん	D001
E00002	加藤茶	D001
E00003	劇団ひとり	D003
E00004	高木ブー	D004
E00005	仲本工事	D003
E00006	いかりや長介	D006
E00007	妻夫木聰	D003
E00008	矢田亜希子	D001
E00009	加藤ローサ	D002



# SQLはプログラミング言語

- C言語やFORTRANは手続き型言語
  - 手続き・制御構造などを駆使してプログラムを作る
    - 「どういう方法で処理するか」を記述
- SQLは非手続き型言語
  - 制御構造は基本的に記述せず、レコードを指定する条件などを記述する
    - 「何を抽出するか」を記述する

# SQLによる問い合わせ (SELECT文)



- ①列名でもよし、  
列関数 (MAX,MIN,AVEなど) でもよし、  
列名に対して演算したものでもよし
- ②INTEGER, CHAR, VARCHARなどの型がある

# SQLによる問い合わせ (SELECT文)

**SELECT** 値式<sub>1</sub>, 値式<sub>2</sub>, … 値式<sub>n</sub>

**FROM** 表参照<sub>1</sub>, 表参照<sub>2</sub>, … 表参照<sub>m</sub>

**WHERE** 探索条件

列名に対し、比較演算子 >, >=, <, <=, <> だけでなく、  
BETWEEN、IN、LIKE、IS (NOT) NULL、EXISTSが使える

# 単純質問(2)

```
SELECT 県名, 人口*1000  
FROM T_PREF
```

列名だけではなく、  
式を含めることができる

T\_PREF

県名	人口
宮城県	2371
埼玉県	7047
山形県	1223
神奈川県	8732
長野県	2211



結果表

県名	人口*1000
宮城県	2371000
埼玉県	7047000
山形県	1223000
神奈川県	8732000
長野県	2211000

# 条件(BETWEEN)

```
SELECT 県名,人口  
FROM T_PREF  
WHERE 人口 BETWEEN 2000 AND 5000
```

T\_PREF

県名	人口
宮城県	2371
埼玉県	7047
山形県	1223
神奈川県	8732
長野県	2211



結果表

県名	人口
宮城県	2371
長野県	2211

# 条件(Like)

```
SELECT *
FROM T_EMP
WHERE 社員名 like '加藤%'
```

T\_EMP

社員番号	社員名	部署コード
E00001	志村けん	D001
E00002	加藤茶	D001
E00003	劇団ひとり	D003
E00004	高木ブー	D004
E00005	仲本工事	D003
E00006	いかりや長介	D006
E00007	妻夫木聰	D003
E00008	矢田亜希子	D001
E00009	加藤ローサ	D002

結果表

社員番号	社員名	部署コード
E00002	加藤茶	D001
E00009	加藤ローサ	D002

# 条件(IN)

```
SELECT *
FROM T_EMP
WHERE 部署コード IN ('D001','D002')
```

T\_EMP

社員番号	社員名	部署コード
E00001	志村けん	D001
E00002	加藤茶	D001
E00003	劇団ひとり	D003
E00004	高木ブー	D004
E00005	仲本工事	D003
E00006	いかりや長介	D006
E00007	妻夫木聰	D003
E00008	矢田亜希子	D001
E00009	加藤ローサ	D002

結果表

社員番号	社員名	部署コード
E00001	志村けん	D001
E00002	加藤茶	D001
E00008	矢田亜希子	D001
E00009	加藤ローサ	D002

# ORDER BY

```
SELECT 県名,人口  
FROM T_PREF  
WHERE 人口 >2000  
ORDER BY 人口
```

T\_PREF

県名	人口
宮城県	2371
埼玉県	7047
山形県	1223
神奈川県	8732
長野県	2211



結果表

県名	人口
長野県	2211
宮城県	2371
埼玉県	7047
神奈川県	8732

# GROUP BY

```
SELECT 部署コード, count(社員名)  
FROM T_EMP  
GROUP BY 部署コード
```

T\_EMP

社員番号	社員名	部署コード
E00001	志村けん	D001
E00002	加藤茶	D001
E00003	劇団ひとり	D003
E00004	高木ブー	D004
E00005	仲本工事	D003
E00006	いかりや長介	D006
E00007	妻夫木聰	D003
E00008	矢田亜希子	D001
E00009	加藤ローサ	D002



結果表

部署コード	COUNT(社員名)
D001	3
D002	1
D003	3
D004	1
D006	1

# HAVING

```
SELECT 部署コード, count(社員名)
      FROM T_EMP
    GROUP BY 部署コード
      HAVING count(社員名)>2
```

T\_EMP

社員番号	社員名	部署コード
E00001	志村けん	D001
E00002	加藤茶	D001
E00003	劇団ひとり	D003
E00004	高木ブー	D004
E00005	仲本工事	D003
E00006	いかりや長介	D006
E00007	妻夫木聰	D003
E00008	矢田ア希子	D001
E00009	加藤ローサ	D002

結果表



部署コード	count(社員名)
D001	3
D003	3

# 結合質問

商品マスターテーブル

商品コード	商品名	単価
10001	携帯電話	20000
10002	パソコン	200000
23333	プリンタ	35000
43990	ハードディスク	10000

納品テーブル

商品コード	納品先	納品数量	納品日
10001	A百貨店	250	7月22日
10001	B電気店	100	7月22日
10002	B電気店	100	7月22日
23333	A百貨店	50	7月21日
23333	B電気店	100	7月22日
23333	Cショップ	50	7月22日
43990	A百貨店	40	7月21日

二つのテーブルを  
商品コードによって  
つなげて見たい  
＝結合演算

SELECT A.\* , B.\*  
FROM 商品マスタ A, 納品 B  
WHERE A.商品コード=B.商品コード

A,Bは相関名という

# 結合質問

商品コードを重ならないよう  
にSELECTリストを選ぶと自然  
結合になる

二つのテーブルを  
商品コードによって  
つなげて見たい  
＝結合演算

**SELECT A.\* , B.\*  
FROM 商品マスタ A, 納品 B  
WHERE A.商品コード=B.商品コード**

商品コード	商品名	単価	商品コード	納品先	納品数量	納品日時
10001	携帯電話	20000	10001	A百貨店	250	7月21日
10001	携帯電話	20000	10001	B電気店	100	7月22日
10002	パソコン	200000	10002	B電気店	100	7月22日
23333	プリンタ	35000	23333	A百貨店	50	7月21日
23333	プリンタ	35000	23333	B電気店	100	7月22日
23333	プリンタ	35000	23333	Cショップ	50	7月22日
43990	コードディス	10000	43990	A百貨店	40	7月21日

商品マスタテーブル部分

納品テーブル部分

# 結合質問

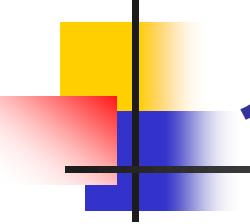
結合質問(JOIN)は、三つ以上のテーブルに対しても可能

```
SELECT A.商品コード, A.商品名, B.納品先, C.納品先住所  
FROM 商品マスタ A, 納品 B, 住所マスタ C  
WHERE A.商品コード=B.商品コード AND  
B.納品先名=C.納品先名
```

結合質問(JOIN)は、ひとつのテーブルだけでも可能

```
SELECT A.商品名, A.単価, B.商品名, B.単価  
FROM 商品マスタ A, 商品マスタ B  
WHERE A.単価>B.単価
```

たとえば、単価の高いものと安いものを並べる場合

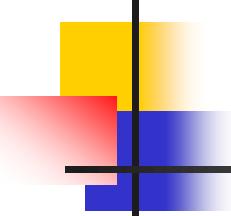


# 入れ子型質問(副照会)

- SELECT文の条件部分(WHERE句)にSELECT文が入れ子で入っているもの。
- どっちかというと、普通は副照会という。

```
SELECT 顧客番号,顧客名  
FROM 顧客マスタ  
WHERE 顧客番号 IN (SELECT 顧客番号  
                      FROM 納品  
                      WHERE 商品番号='10001')
```

納品テーブルの中にある商品'10001'を購入した顧客を調べる(納品テーブルでひつかかった顧客番号をもつ顧客マスタのレコードを調べ、顧客番号と顧客名を表示する)



# データの挿入(INSERT文)

商品マスタテーブル

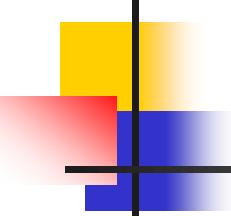
商品コード	商品名	単価
10001	携帯電話	20000
10002	パソコン	200000
23333	プリンタ	35000
43990	ハードディスク	10000

新製品 MP3プレーヤー(単価25000円)を登録したい

INSERT INTO 商品マスタ VALUES  
(10003, 'MP3プレーヤー', 25000)

単価はまだ決まっていないがMP3プレーヤーを登録したい

INSERT INTO 商品マスタ(商品コード,商品名) VALUES  
(10003, 'MP3プレーヤー')



# データの挿入(INSERT文)

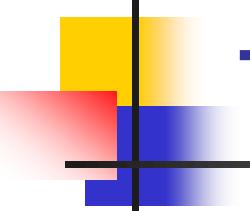
商品マスタテーブル

商品コード	商品名	単価
10001	携帯電話	20000
10002	パソコン	200000
23333	プリンタ	35000
43990	ハードディスク	10000

P社製製品テーブルから、データを商品マスタテーブルにとどめたい

```
INSERT INTO 商品マスタ  
(SELECT 製品コード, 製品名, 希望小売価格  
FROM P社製品)
```

INSERT文にSELECT文を含めることができる  
VALUESがないところに注意



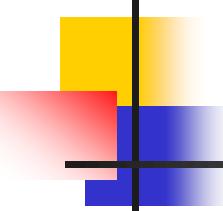
# データの変更(UPDATE文)

商品コード	商品名	単価
10001	携帯電話	20000
10002	パソコン	200000
23333	プリンタ	35000
43990	ハードディスク	10000

商品コード'10001'の商品の単価を15000円に修正したい

```
UPDATE 商品マスタ  
SET 単価=15000  
WHERE 商品コード='10001'
```

同じレコードの二つ以上のフィールドを変更したい場合は  
SET 列名1='XXXX', 列名2='YYYY'  
のように列記する。



# データの削除(DELETE文)

商品コード	商品名	単価
10001	携帯電話	20000
10002	パソコン	200000
23333	プリンタ	35000
43990	ハードディスク	10000

商品コード'10001'の商品はWITHDRAWになったので  
マスタから削除したい

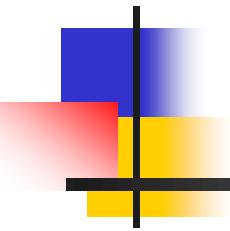
```
DELETE FROM 商品マスタ  
WHERE 商品コード='10001'
```

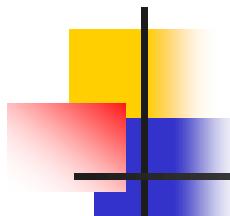


# データベース(第7回)

情報工学科 木村昌臣

# ANSI/X3/SPARC の3層スキーム構造





# DBMSはどうあるべき？

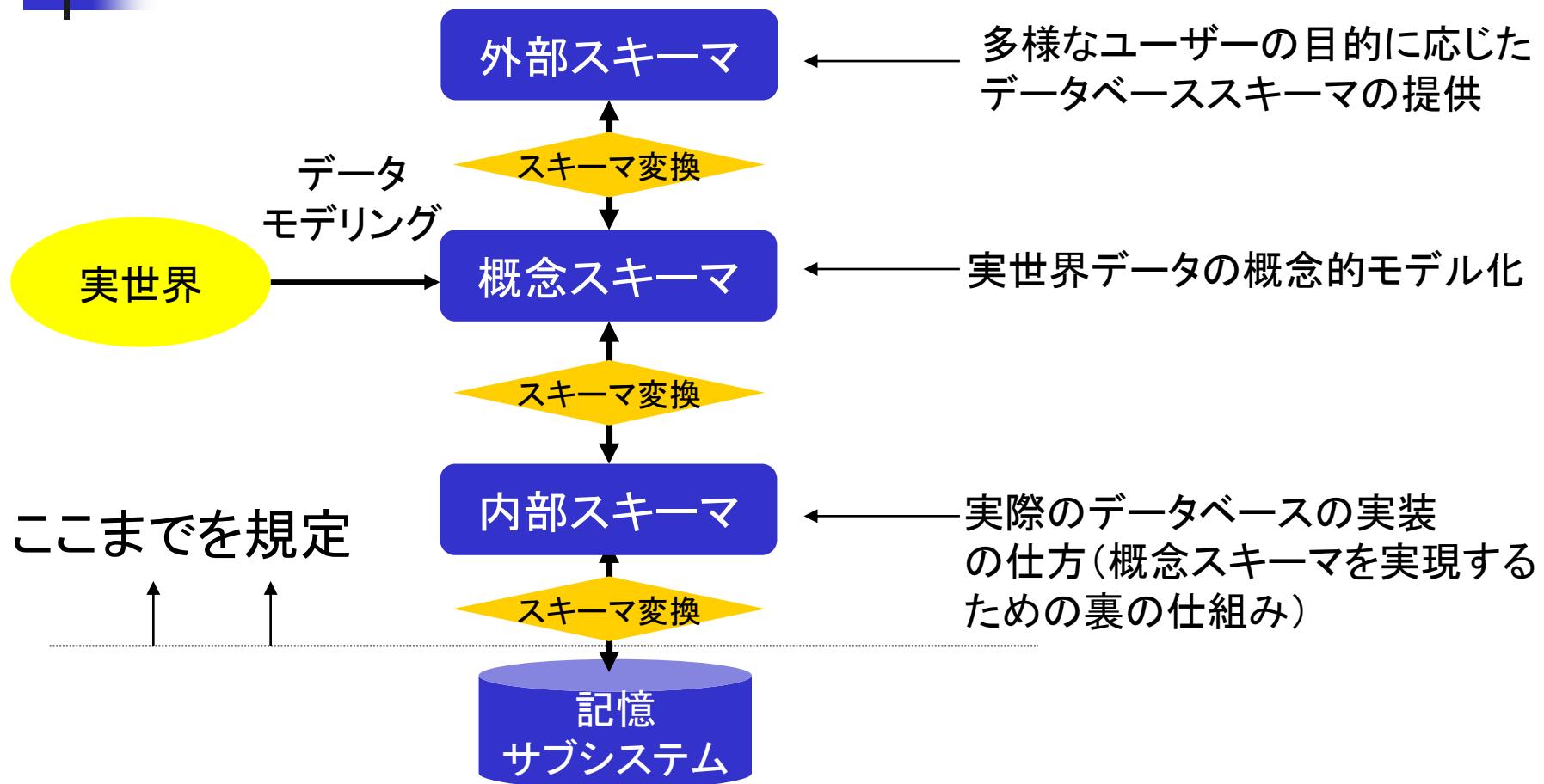
- DBを構築する場合、どのような構成にすべきか



## ANSI/X3/SPARC の 3層スキーマ構造

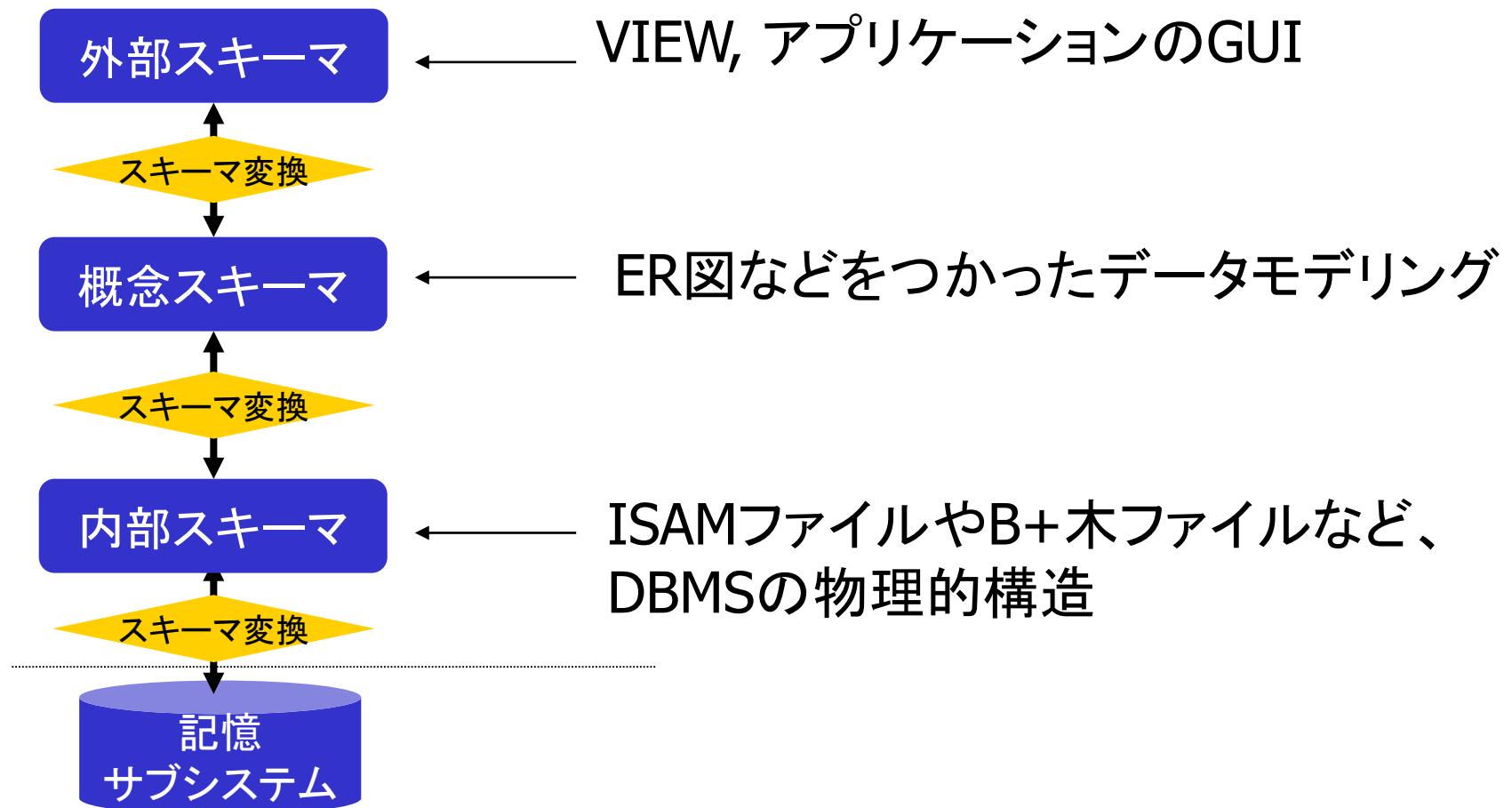
- RDBMSだけでなくDBMSであれば満たすべき構造
- 1973 審議開始
- 1978 報告書にまとめる

# ANSI/X3/SPARC 3層スキーマ構造

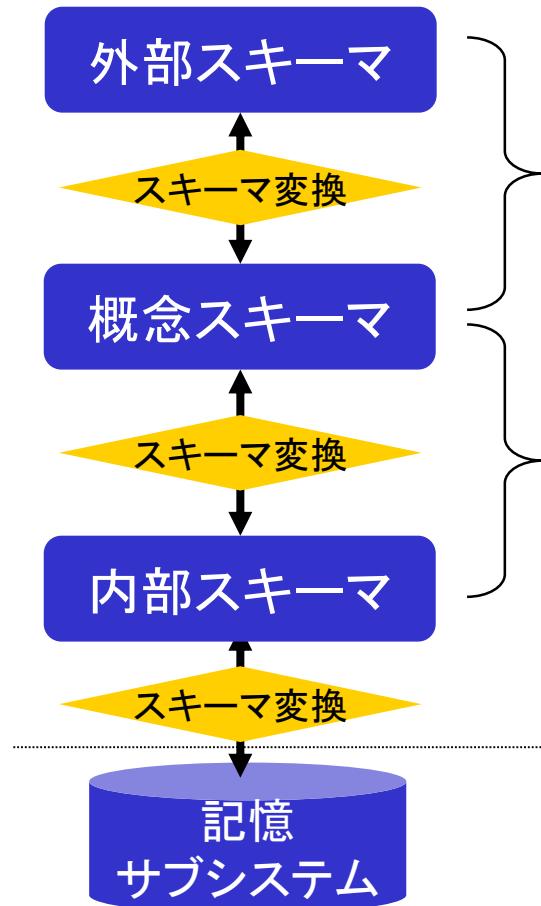


# ANSI/X3/SPARC

## 3層スキーマ構造(例)



# 3層スキーマの意義



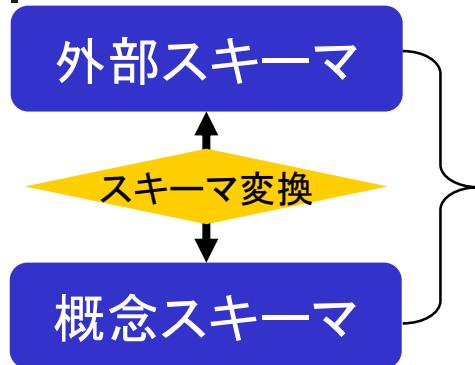
## 論理的データ独立性

実世界が変化しても、アプリケーションプログラムを普遍に保つことができる

## 物理的データ独立性

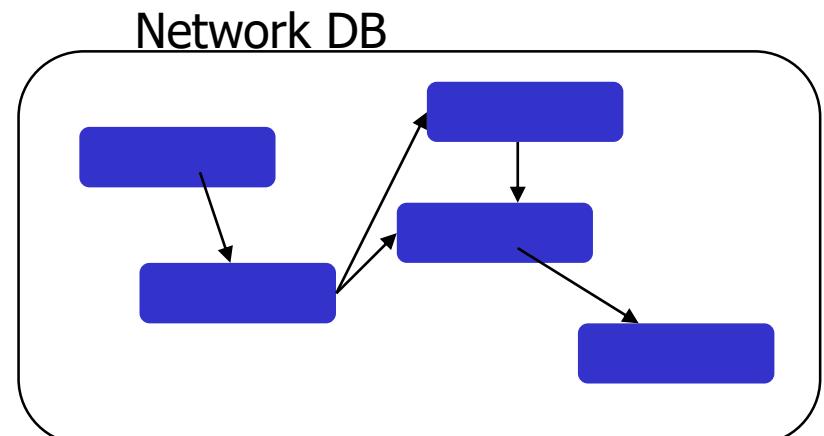
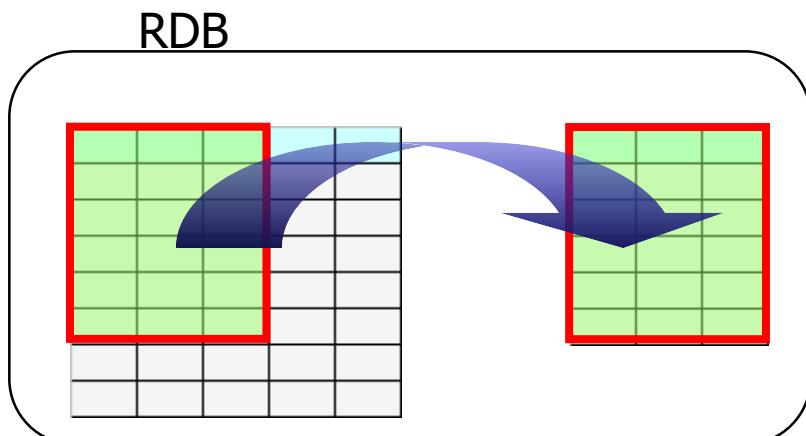
より効率のよい検索ロジックやファイルアクセス法に実装が変わっても、DBのユーザーに影響を与えない

# 3層スキーマの意義



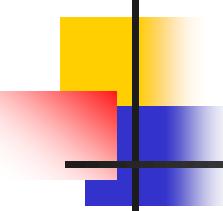
## 論理的データ独立性

- ✓ リレーションナルデータベースでは、問い合わせ結果と元の表の構造がおなじため、問い合わせ結果として得られる表を外部スキーマとして利用することが可能
- ✓ ネットワークデータベースや階層データベースは概念スキーマと内部スキーマの区別がはっきりしないため、外部スキーマのサポートは困難





VIEW



# ビュー(View)

- 実際にデータを持つリレーションを実リレーションという。
- 問い合わせの結果できるリレーションのことを結果リレーションという。
- ビューとは、結果リレーションを仮想的なリレーションとして定義したもの。
  - ビューの定義は、結果リレーションを得るための定義（実リレーションへの問い合わせ）のみ。
  - ビューには、実データが存在しない。（実データがあるのはあくまで実リレーション）

# VIEW

View作成のSQL文

```
CREATE VIEW View名  
AS  
SELECT文
```

SELECT文の結果をViewという仮想的な表としてユーザーに提供する

例)

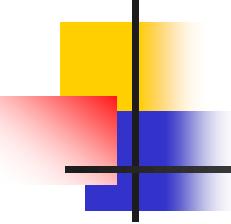
## 選択ビュー

```
CREATE VIEW 中堅社員  
AS  
SELECT *  
FROM 社員  
WHERE 年齢 BETWEEN 30 AND 40
```

社員テーブルから該当レコードを選択して表示

## 和ビュー

```
CREATE VIEW 全社社員  
AS  
SELECT *  
FROM 東京本社社員  
UNION  
SELECT *  
FROM 大阪支社社員
```



# VIEWの種類

- 選択ビュー
  - 元のテーブルから条件に合うレコードを選択してビューとして定義
- 和ビュー
  - 複数テーブルのレコードの和集合をビューとして定義
- 結合ビュー
  - 複数のテーブルを結合(JOIN)したものをビューとして定義
- 値式を使って定義されるビュー
  - SELECTリストの列名の代わりに値式をつかってできた「表」をビューとして定義

# VIEWの更新可能性

```
CREATE VIEW 取引  
AS  
SELECT X.仕入れ先, Y.納入先  
FROM 供給 X, 需要 Y  
WHERE X.部品 = Y.部品
```

(XX工業,YY商店)を  
削除したい  
どうしたらいい?

1. 供給テーブルから  
(XX工業, 部品ZZ)  
を削除

2. 需要テーブルから  
(部品ZZ, YY商店)  
を削除

3. 1.2.の両方を削除

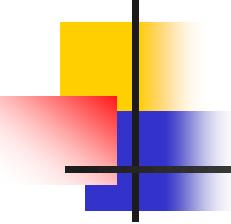
1.~3.は、意味としては全く異なるが  
これらのどれにしたらいいか、SQL  
からは判別できない。

# SQL規格におけるVIEWの更新可能性

- DISTINCTを使わない
- 値式はすべて列参照
- FROM句には単一のテーブル(実表もしくは更新可能なVIEW)が指定されている
- 副照会をしているときは、相関がないこと
- GROUP BYやHAVINGを使用していないこと

つまり

一枚のテーブルに対する射影や選択演算で  
切り出された結果表である場合のみ更新可能



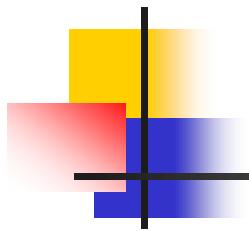
# 体現VIEW

- 通常のVIEWに問い合わせや更新を行う場合には、ビュー定義がそれらに組み込まれて実行される。
- ただし、それではパフォーマンスがよくないことがある
- あたかも実テーブルのようにデータベースに格納した（実データを伴う）VIEWのことを**体現VIEW**という。



# データベース(第8回)

情報工学科 木村昌臣



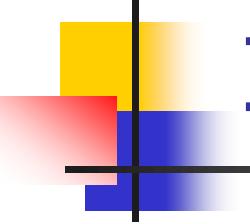
# RDBMSの三大機能

1. メタデータ管理
2. 質問処理
3. トランザクション管理

## 1. メタデータ管理

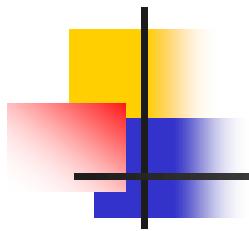
## ■ メタデータとは

- 「データのデータ」の意味
  - RDBMSの場合、以下を指す
    - テーブルに関するデータ
    - 各テーブルの列名・属性
    - キーについてのデータ
    - ユーザー名、権限(ロール)
    - スキーマ名、スキーマ所有者、カタログ名
    - ビューに関するデータ(名前、定義、更新可能性についてのデータ)
    - アクセス法(インデックスの定義)
    - シノニム(別名)
    - 統計データ



# 1. メタデータ管理

- これらのメタデータをRDBMSはユーザーに提供できなければならない
  - 質問処理(SQL)を実行するときに必要
  - 初めてそのデータベースを使うユーザーに「どんなテーブルがあるか」などの情報を提供するときに必要
- 「システムカタログ」というテーブル・ビュー群にて管理

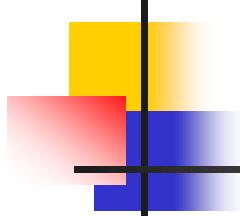


# 1. メタデータ管理(情報スキーマ)

- SQL-92規格では、RDBMSはメタデータを以下の形式で提供するよう定められている
  - USERS,SCHEMA,TABLES,COLUMNS…などの24個の**実表(DEFINITION\_SCHEMA)**
  - これらが保持するメタデータをエンドユーザー やアプリケーションに提供するため、これらをSELECT文で検索するために定義された**ビュー(INFORMATION\_SCHEMA)**

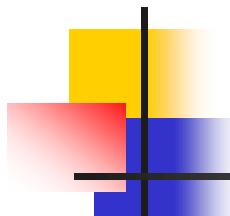


基本的には更新不可



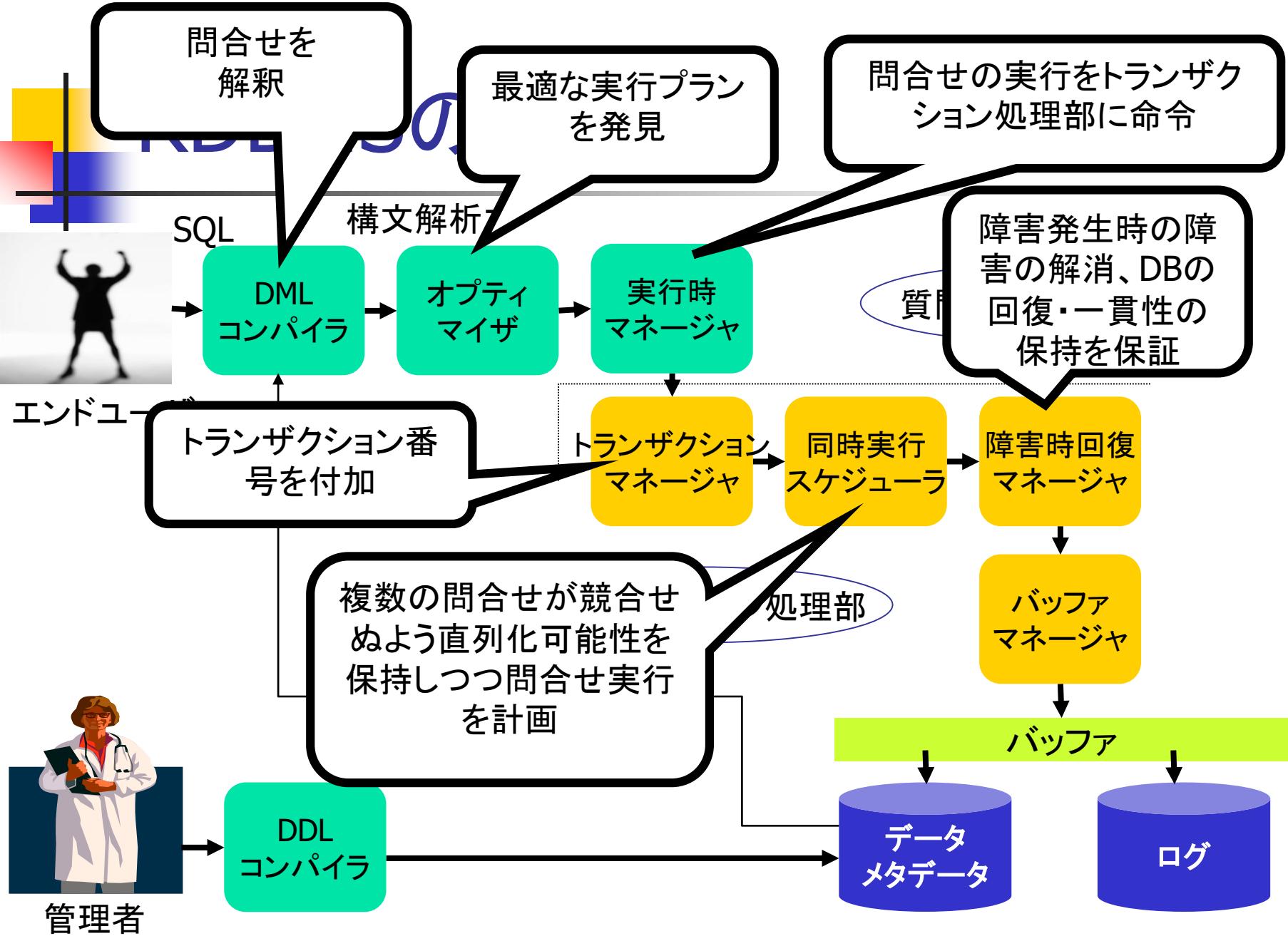
## 2. 質問処理

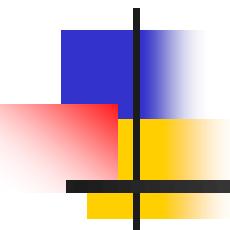
- SQLのSELECT文などの解釈・コンパイル
  - 所望のデータを検索するための内部スキーマレベルのアクセスコードを生成
- 質問処理の最適化
  - SQLは非手続き的なため、RDBMSが責任もつてコードの最適化を行う必要あり



### 3. トランザクション管理

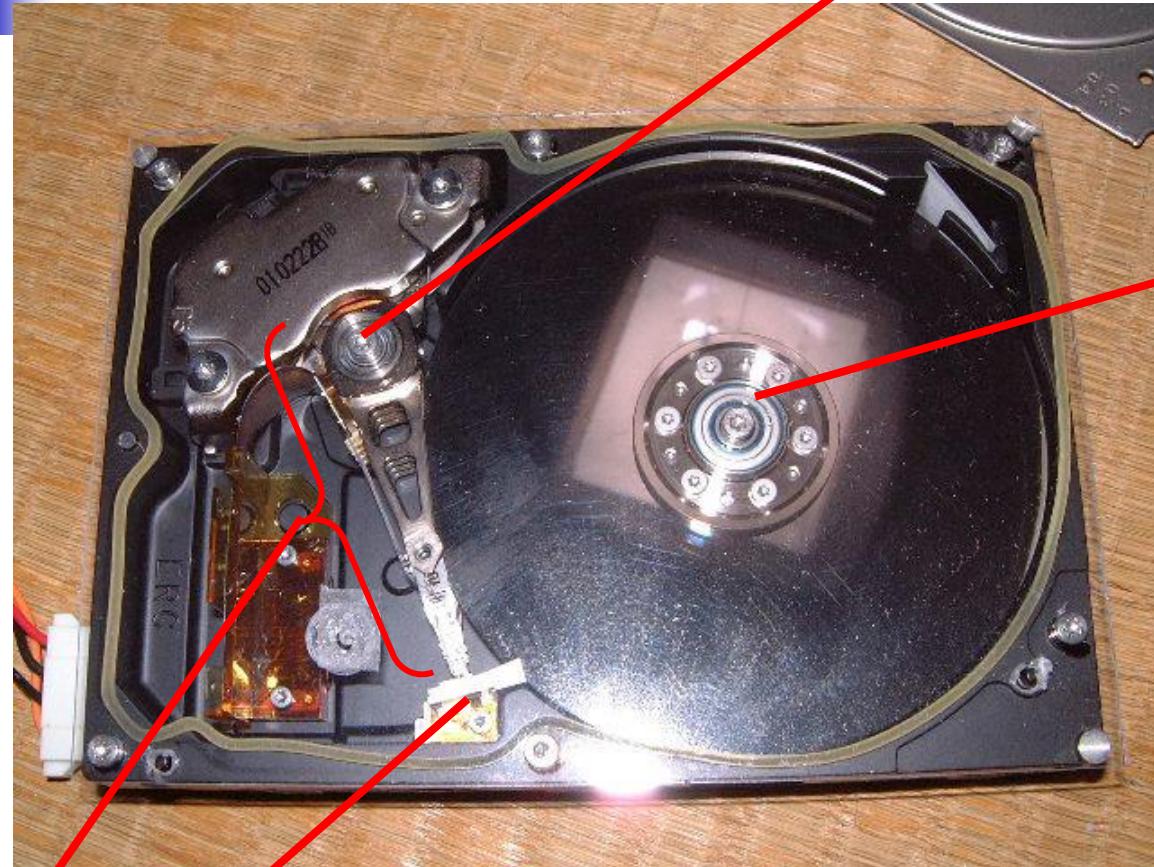
- トランザクション
  - データベースのデータの読み書きの単位
- これを管理することにより、データベースの一貫性を保証
  - 障害時回復
  - 同時実行制御
  - 例)書き込みに失敗したら、そのトランザクションの直前の状態まで戻す(ロールバック)





# ファイルのアクセス方式と インデックス

# HDDの構造

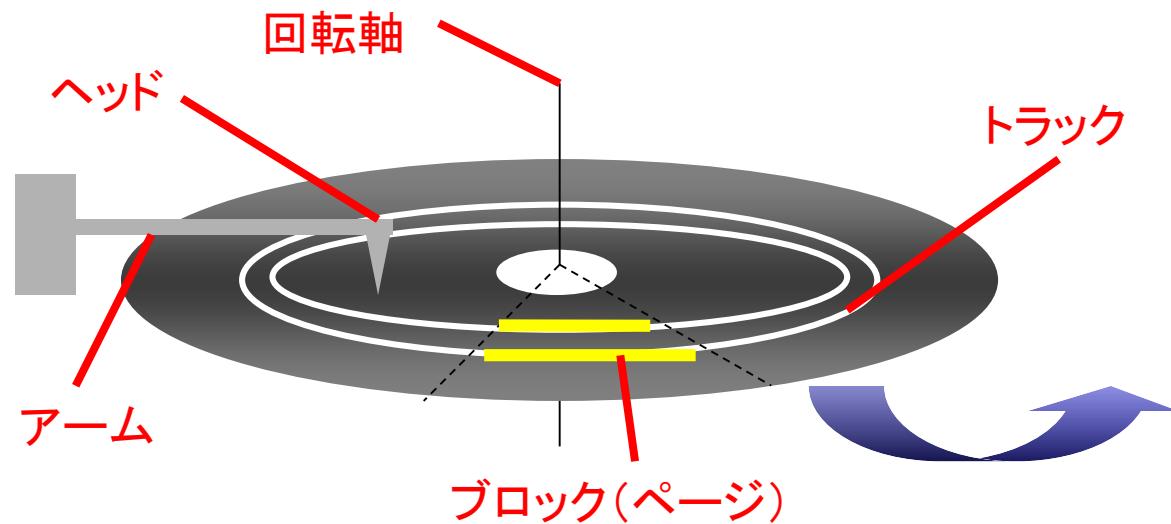


アーム ヘッド

アクチュエータ

回転軸

# HDDとファイル格納の原理



通常512バイト～4096バイト  
ブロックのアドレス =

- ディスク面の認識番号
- トラック番号
- (そのトラックでの)ブロック番号の組



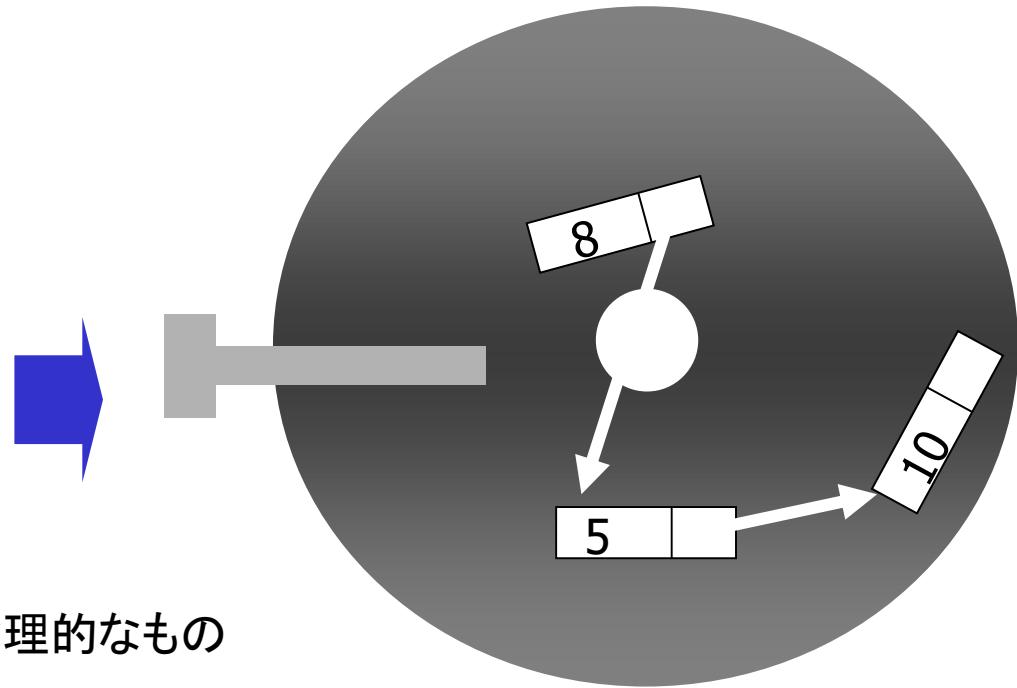
LOB型を含むレコード  
:大きいので複数ブロックに  
1レコード

(LOB: Large Object)

# HDDとファイル格納の原理

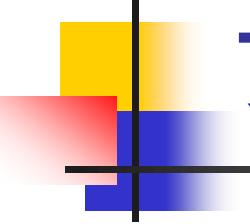
テーブル(≒ファイル)

列1		
8	-	-
5	-	-
9	-	-



ユーザーから見える  
上のテーブルは、実は論理的なもの

物理的には、右図のようにハードディスク  
内に分散し、個々のレコードはその値と  
次のレコードのアドレス(ポインタ)  
を持って格納されている。



# ファイルアクセス方式

- ファイルの中にあるレコードを探し出す方式
  - 一つのテーブルは、ひとつのファイルとして実現されることが多いことに注意
- 代表的な例は
  - スキャン(線型探索)
  - 探索
  - インデックス法
  - ハッシュ法

# スキャン

条件: 列1=8

列1	列2
10	
5	
8	
3	



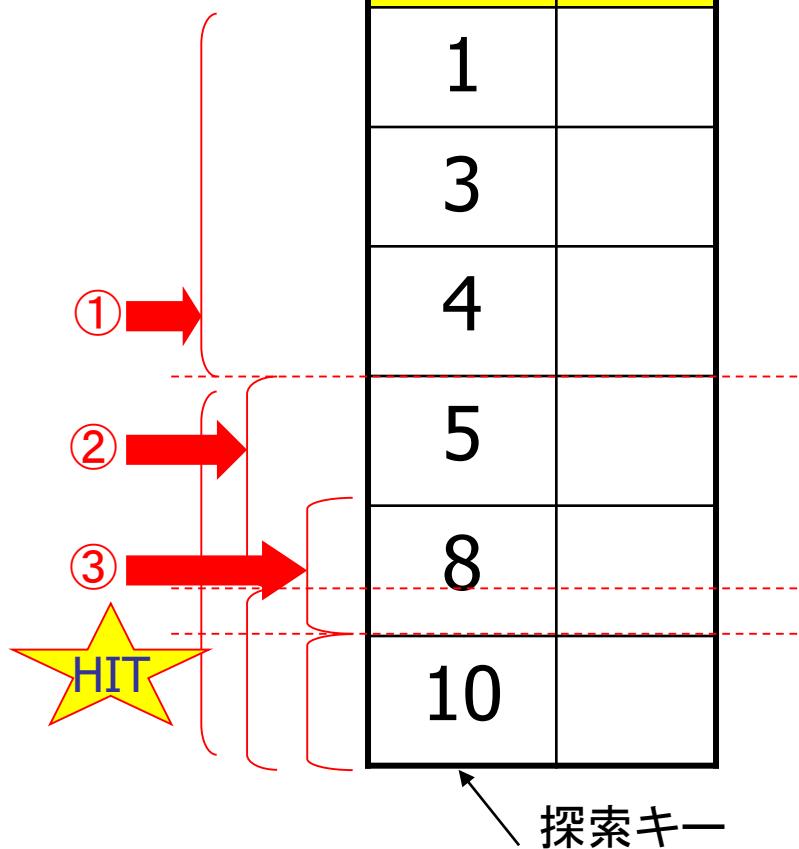
レコードが、この順番でディスクに格納されていることが前提

ある列(探索キー)をもとにレコードを探索する。  
探索は、テーブルの上から逐次、条件に合うか判定しながら進んでいく。

平均計算量は  $O(n)$  (n:レコード数)

# 探索(1:2分探索)

条件: 列1=8



レコードが、探索キーでソートされていることが前提

探索は、探索キーとなる列を半分ずつに区切りながら条件とマッチするまで進んでいく。

平均計算量は  $O(\log_2 n)$   
(n:レコード数)

# 探索(2: ブロック探索)

条件: 列1=8

列	列
1	2
1	
2	
4	
5	
7	
8	
25	
31	
55	

探索キー

レコードが、探索キーでソートされていることが前提

探索は、レコードをm個のブロックにわけ、まずその最後のレコードの探索キーの値と条件と比較する。

ブロックの最後のレコードの値が条件より小さければ次のブロックを探し、ブロックの最後のレコードの値が条件より大きければそのブロックをスキヤンする

平均計算量は  $O(\sqrt{n})$

( n: レコード数 )

計算量 =  $n/(2m) + m/2$  を最小にするmを選ぶ )

# インデックス法

インデックスとは

テーブルのある列の値およびその値をもつレコードへのポインタ  
の対からなるレコード群



# 順次フ

インデックスフィールドに関して  
順次か非順次かでインデックスの  
内部構造が変わる

# マイル

ある列に関して順次  
になっていて、かつ  
その列が候補キーなら  
**順序キー**

社員マスタ

社員番号	氏名	健保番号	部署コード
E1111	宇田川	A1111	A11
E2222	河瀬	A4055	A11
E3333	小池	A7032	B21
E4444	高橋	A4200	B21
E5555	西尾	A8099	C31

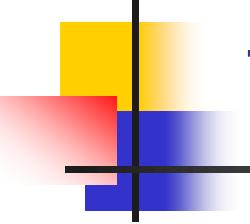
**順序フィールド**

社員マスタテーブル  
の各レコードは  
部署コードの順に  
並んでいる

社員マスタテーブル  
の各レコードは  
健康保険番号の順に  
並んでいない

社員マスタは  
健保番号に関して  
**非順次**

社員マスタは  
部署コードに関して  
**順次**



# 1次インデックス・2次インデックス

- 1次インデックス
  - 順序キー上に張ったインデックス
  - インデックスは元のテーブルのインデックスフィールドを順序キーとして持つ
- 2次インデックス
  - 順序キー以外の列の上に張ったインデックス
  - 順序フィールドではないが候補キーである場合も1次インデックスと同様
  - 候補キーでない場合は、一般に複数のレコードポインタを収めるためのブロックをはさむ。インデックスのレコードポインタは該当するブロックを指す。
  - 順序フィールドだが候補キーではない場合はクラスタリングインデックスと呼ぶ

# 1次インデックス

## ■ 1次インデックス

- 主キー上に張られたインデックス
  - 主キーの値が決まればレコードは一つ決まる
- インデックスは
  - 元のテーブルの主キーの値(インデックス上のキーの並びはソートされている)
  - 実際のレコードへのポインタ

の組を持ち、キーの値とレコードを直接結びついている

社員番号	ポインタ
E1111	◆
E3333	◆
E5555	◆

社員番号	氏名	健保番号	部署コード
E1111	宇田川	A1111	A11
E2222	河瀬	A4055	A11
E3333	小池	A7032	B21
E4444	高橋	A4200	B21
E5555	西尾	A8099	C31

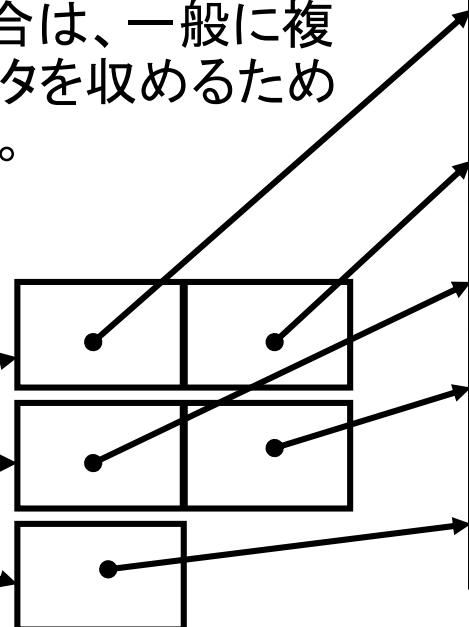
# 2次インデックス

## ■ 2次インデックス

- 順序キー以外の列の上に張ったインデックス
- 順序フィールドではないが候補キーである場合、1次インデックスと同様
- 候補キーでない場合は、一般に複数のレコードポインタを収めるためのブロックをはさむ。

X部署コード

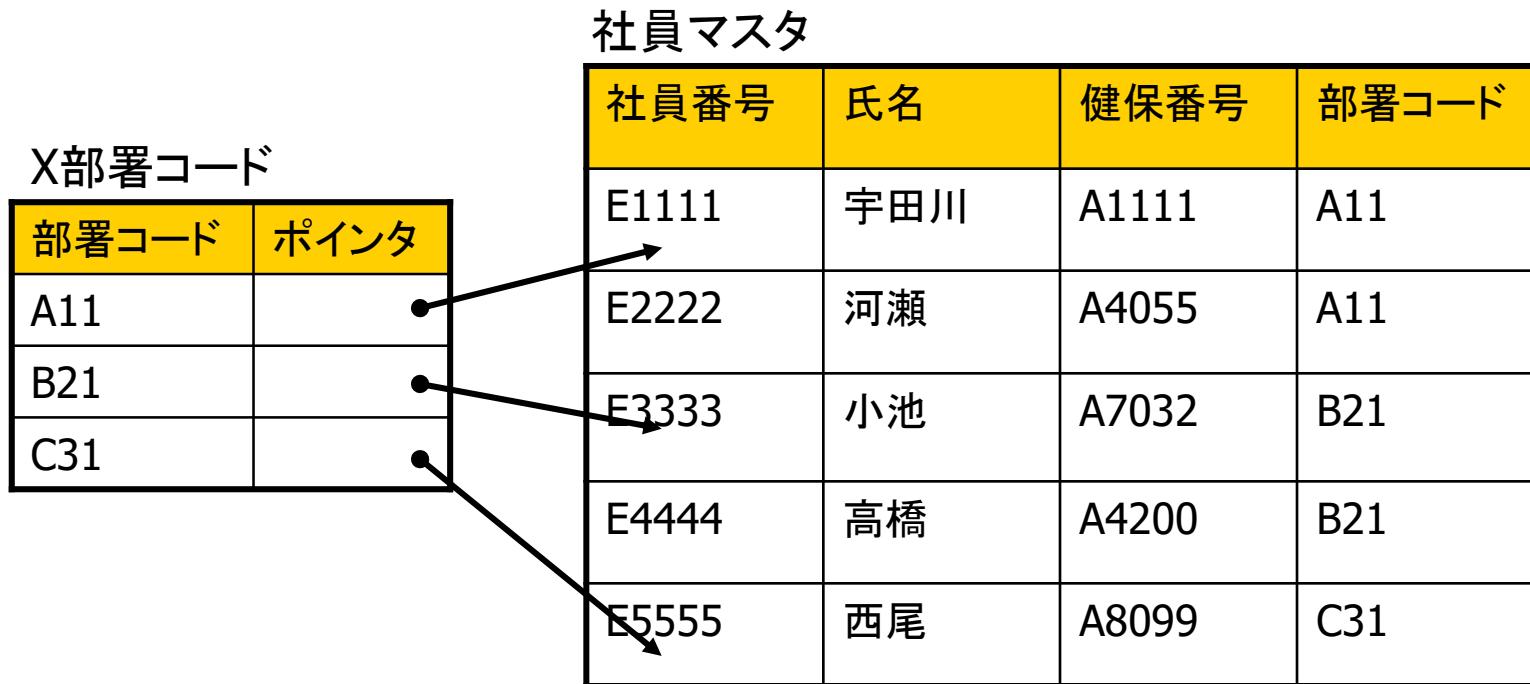
部署コード	ポインタ
A11	●
B21	●
C31	●



社員番号	氏名	健保番号	部署コード
E1111	宇田川	A1111	A11
E2222	河瀬	A4055	A11
E3333	小池	A7032	B21
E4444	高橋	A4200	B21
E5555	西尾	A8099	C31

# クラスタリングインデックス

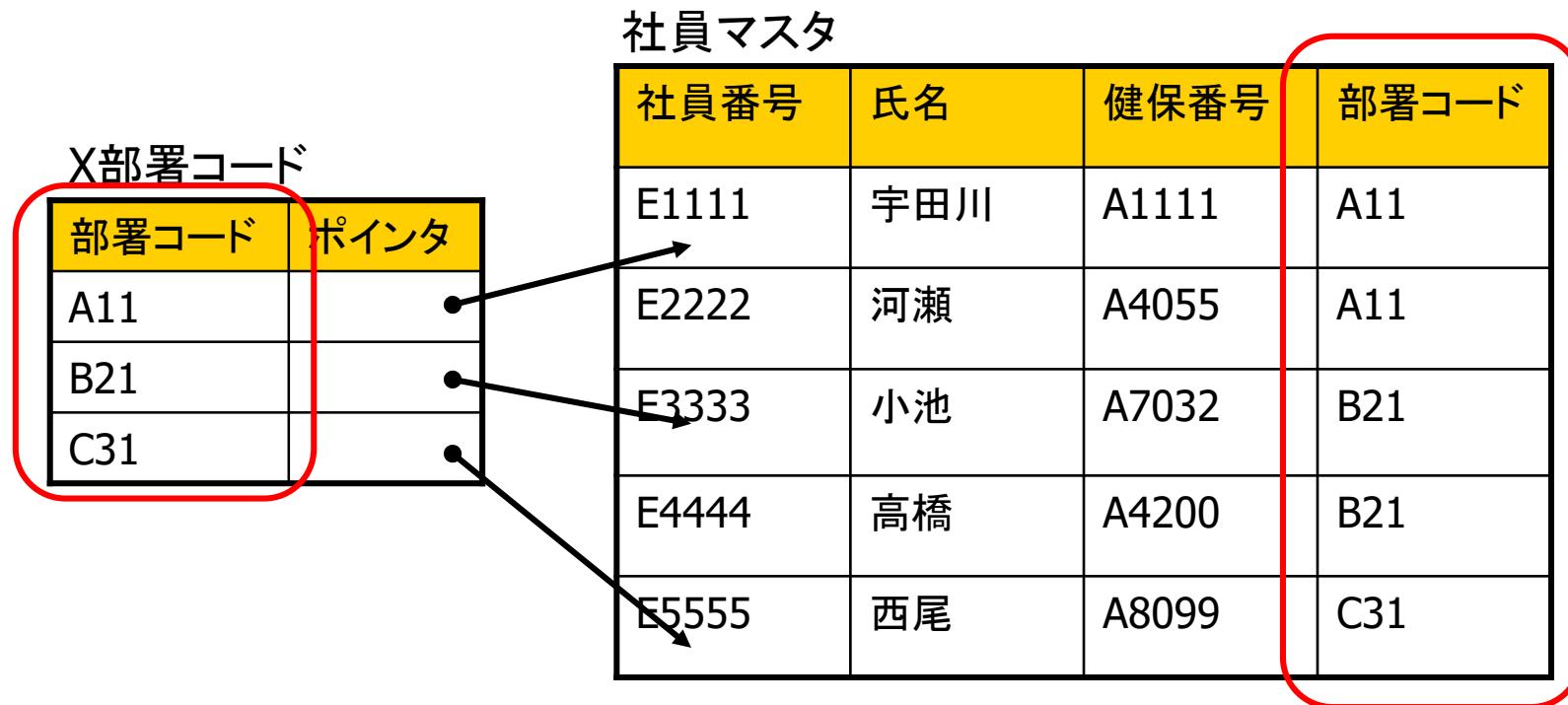
順序フィールドだが候補キーでない場合、その列の上で定義されたインデックスを特に「クラスタリングインデックス」と呼ぶ



# 密集インデックス法

順序フィールドのすべての値をインデックス内にもつ方法

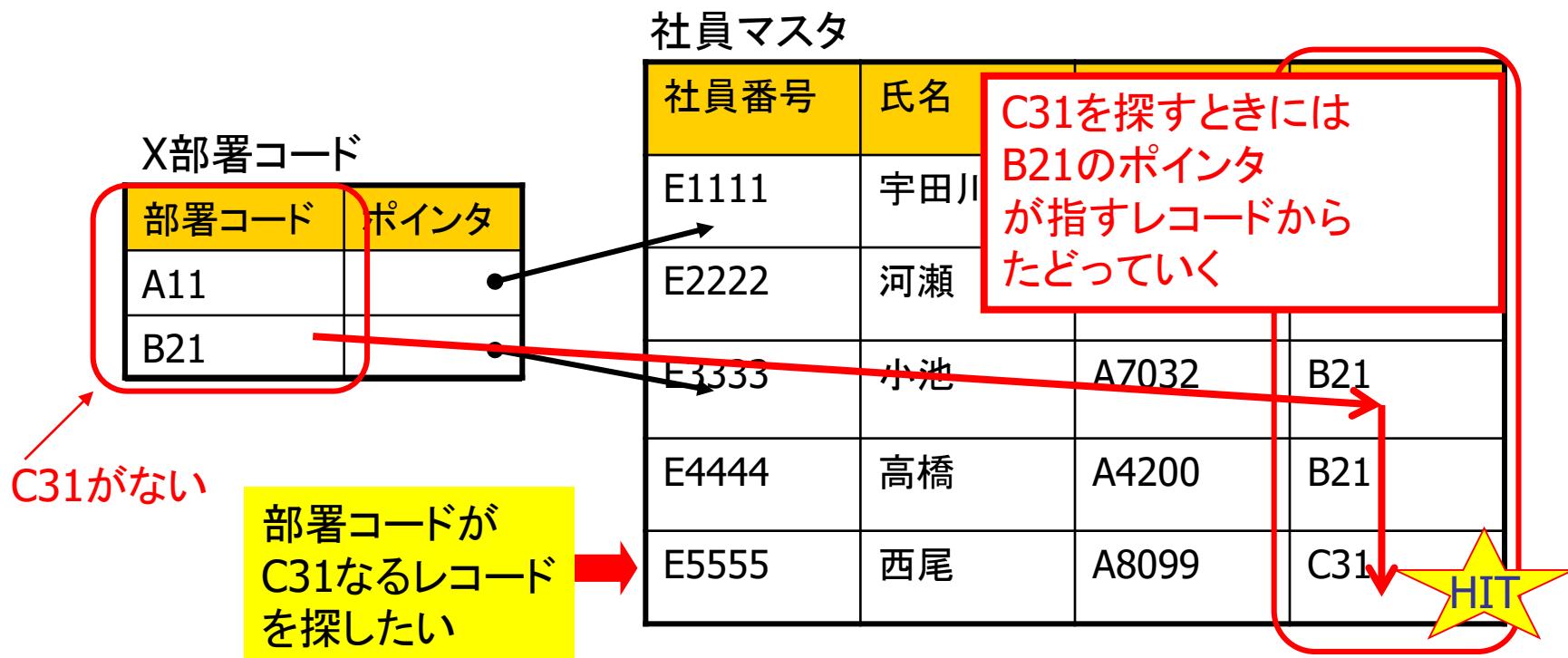
- 次の点在インデックスよりパフォーマンスはよい
- 記憶領域をよけいにくう

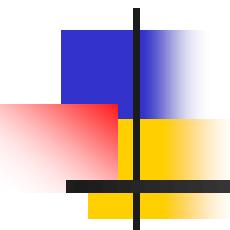


# 点在インデックス法

順序フィールドの一部の値のみをインデックス内にもつ方法

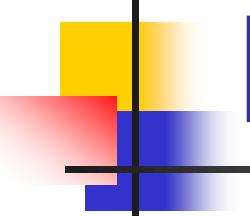
- 密集インデックスよりパフォーマンスは悪い
- 記憶領域は稼げる





# データベース（第9回）

情報工学科 木村昌臣



# 【復習】ファイルアクセス方式

- ファイルの中にあるレコードを探し出す方  
式
  - 一つのテーブルは、ひとつのファイルとして実  
現されることが多いことに注意
- 代表的な例は
  - スキャン(線型探索)
  - 探索
  - インデックス法
  - ハッシュ法

} 先回はここの話をした！

# 【復習】インデックス

テーブルのある列の値および  
その値をもつレコードへのポインタ  
の対からなるレコード群

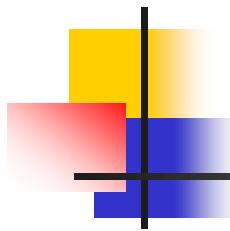
X住所

住所	
東京	
埼玉	
千葉	

社員番号	氏名	住所	部署
E1111	宇田川	東京	A11
E2222	河瀬	埼玉	A11
E3333	小池	東京	B21
E4444	高橋	埼玉	B21
E5555	西尾	千葉	C31

インデックスフィールド

この列を  
検索条件  
にする  
場合、  
この列に  
インデックス  
を張る

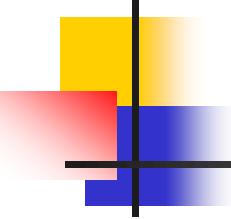


# 多段インデックス

- インデックスフィールドの値の種類が増えると、インデックスの「レコード数」が増えてしまう
  - 結果的にスキャンと変わらなくなってしまう
  - インデックスの参照量を減らしたい

もう一工夫!

多段インデックス



# 多段インデックス

## ■ ISAMインデックス

非平衡木であり、更新が速い

(=検索速度がインデックスなしと同等になってしまう場合もある)

インデックスフィールドが順序キーになっていることが前提

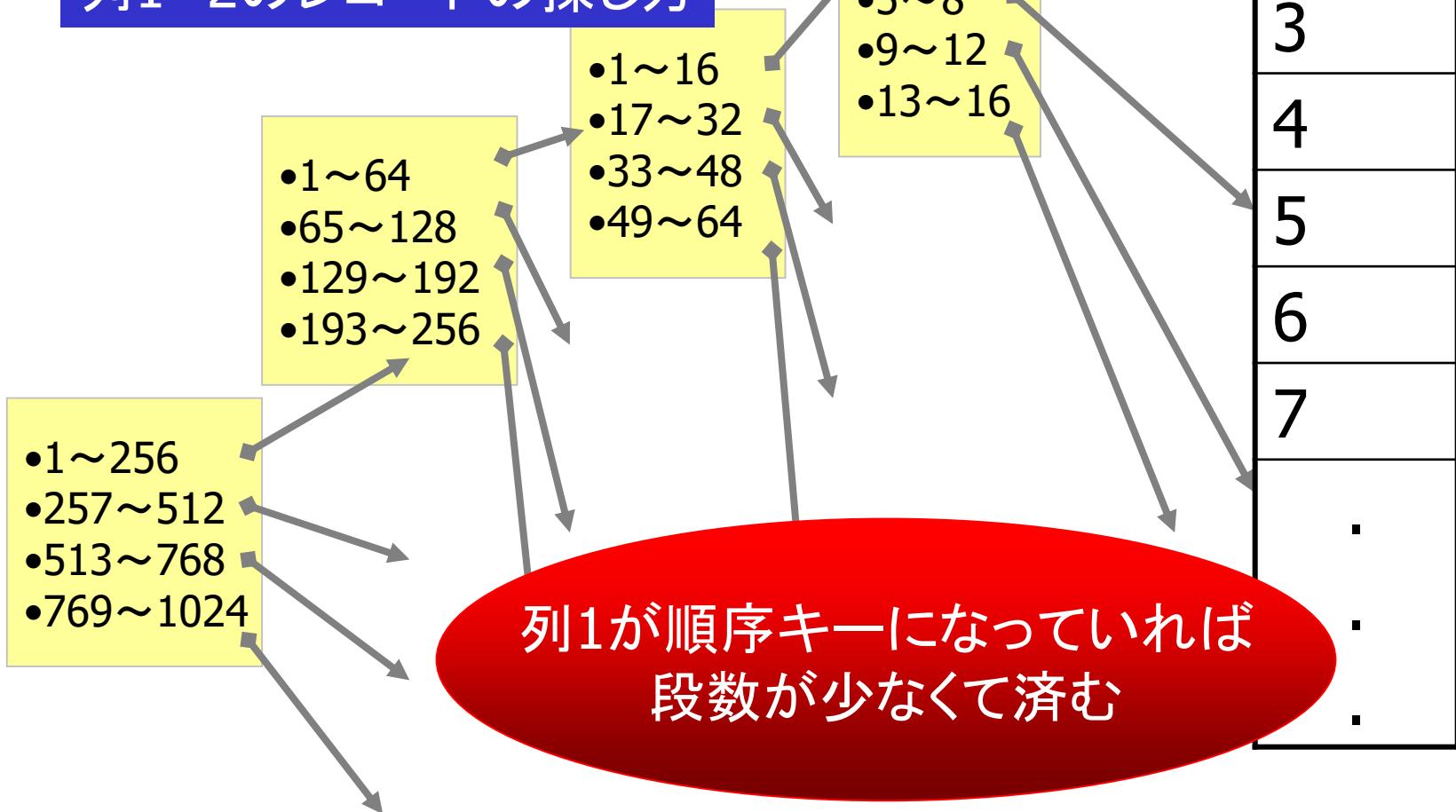
インデックスフィールドの値によって、レコードがソートされており且つ一意に決まる。

## ■ B+木インデックス

平衡木をつくり、常に検索が速い(更新が頻繁にあると遅い)

# ISAMインデックス

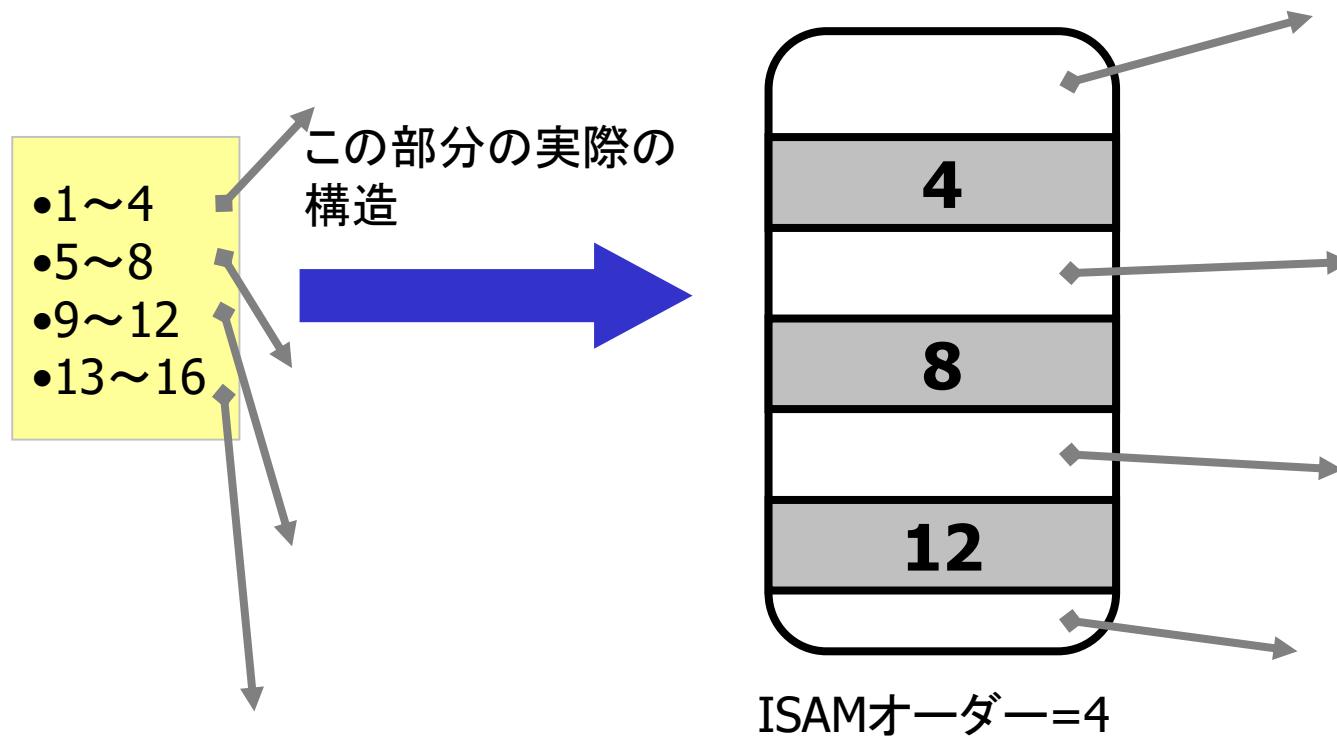
列1=2のレコードの探し方



ソート  
されて  
いる  
こと  
に  
注意

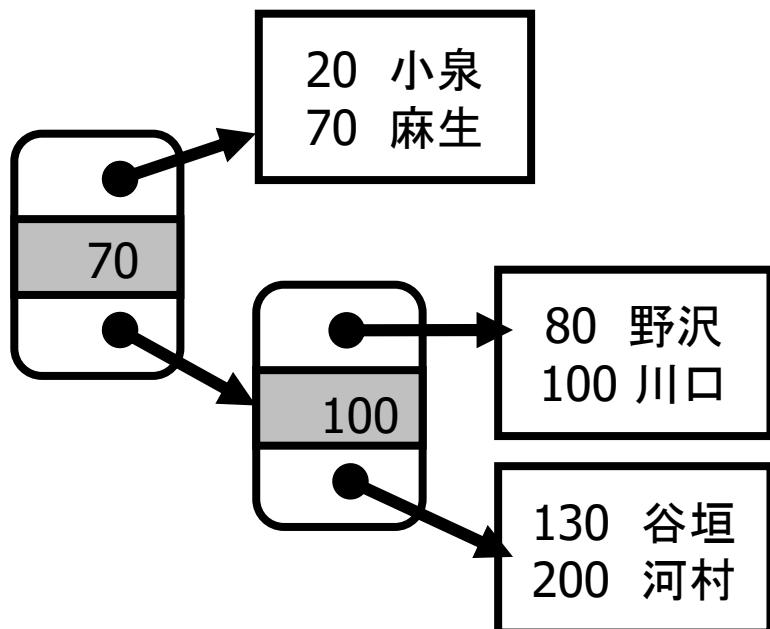
# ISAMインデックス

- インデックス付き順序アクセス法
  - Indexed Sequential Access Method



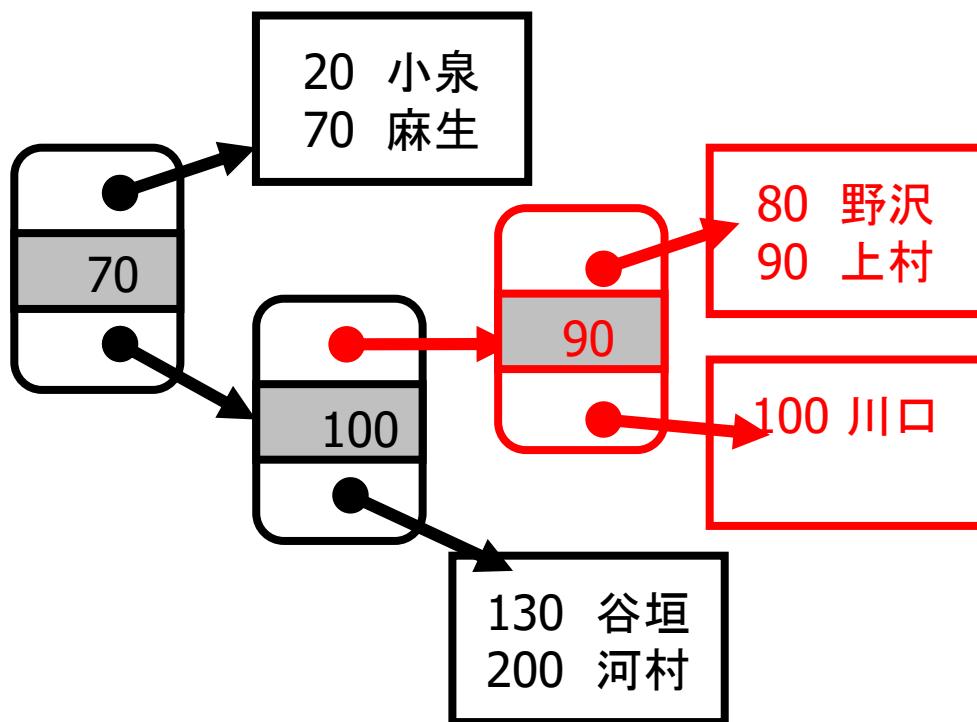
# ISAMインデックス

右のテーブルの、ISAMオーダーが2のISAMインデックスを考える。  
また、1ページ(ブロック)には2レコード入るとする。

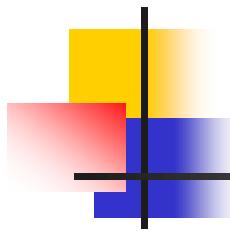


# ISAMインデックス

社員番号 90 名前 上村 という社員のレコードが追加されると……

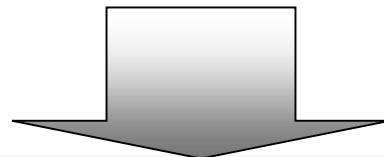


社員番号	名前
20	小泉
70	麻生
80	野沢
90	上村
100	川口
130	谷垣
200	河村



# ISAMインデックスの問題点

- 動的再配置されない
  - ある葉にレコードがたくさん追加されても別の葉に既存のレコードを移し変えることはない



一部のノードの下に  
レコードが集中してしまう  
**非平衡木**

例えば、社員番号70以下のレコードが増えると小泉・麻生がいたノードにレコードが集中してしまう

# B+木

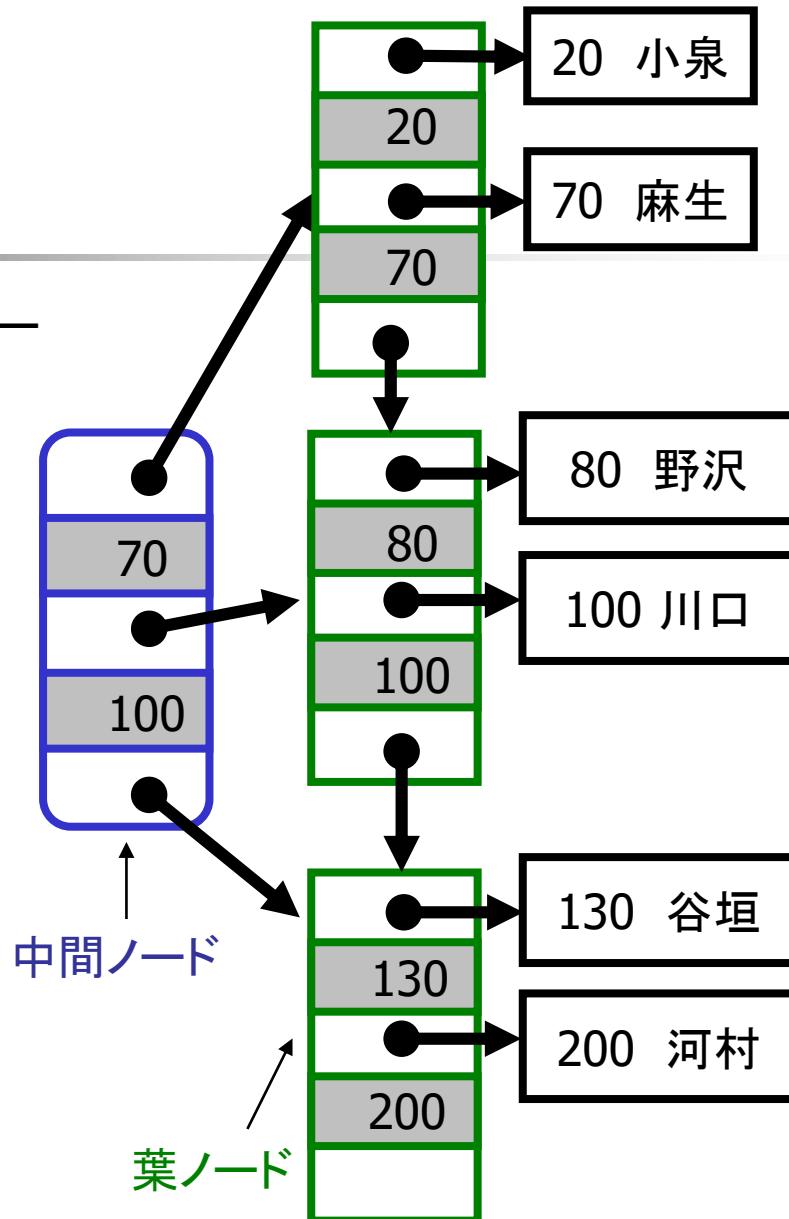
pをB+木の探索キー 社員番号のオーダーとする。(右の図ではp=3)

## 中間ノード:

- 他の中間ノードもしくは葉ノードへのポインタ(木ポインタ)をもつ
- $p/2 \sim p$ 個の木ポインタを持たなければならない

## 葉ノード:

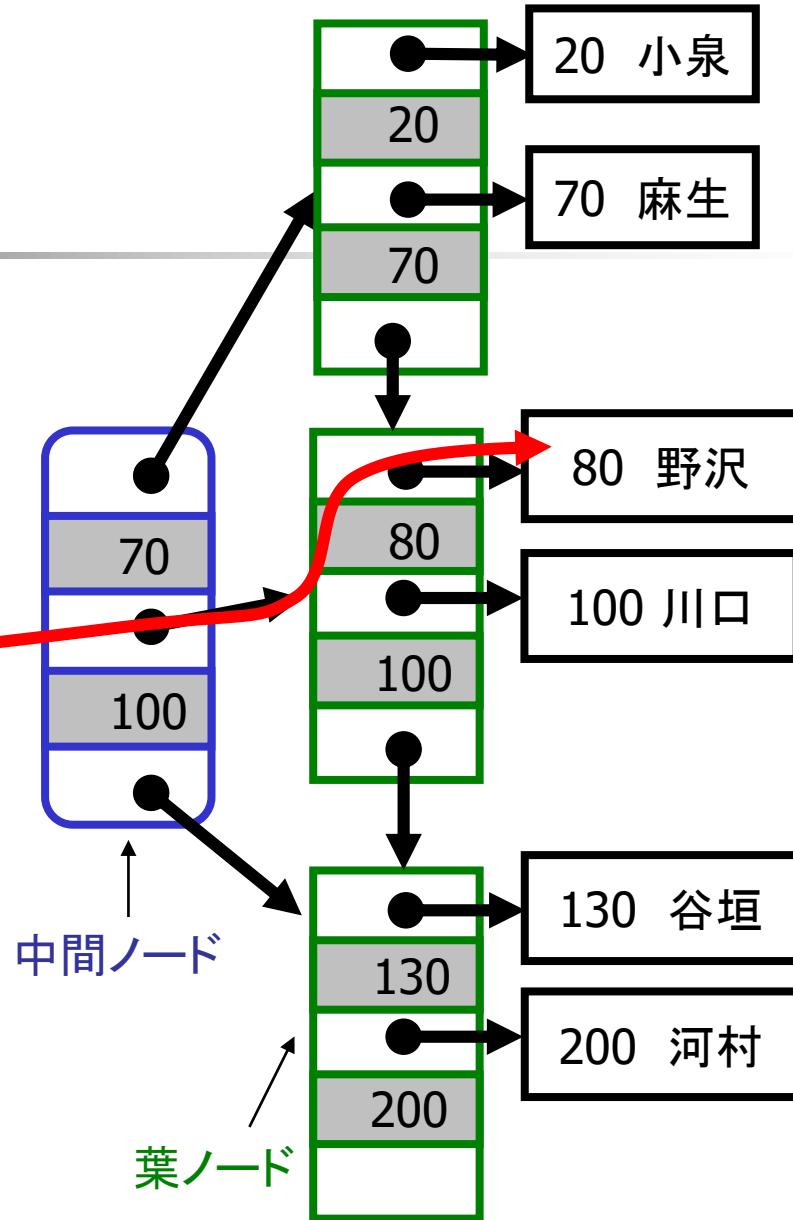
- データへのポインタ(データポインタ)は葉ノードのみ持つ
- 最後のポインタは隣の葉ノードをさす
- $(p-1)/2 \sim p-1$ 個のデータポインタを持たなければならない



# B+木での レコード探索

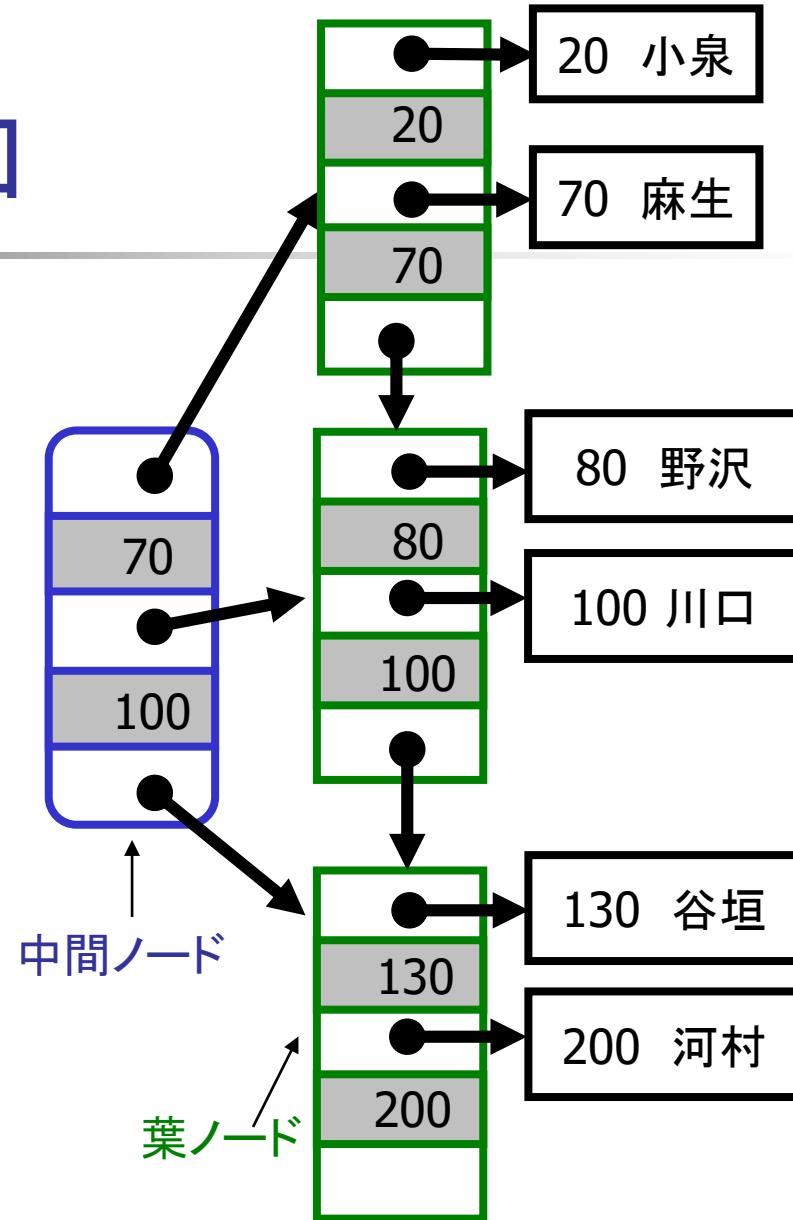
社員番号=80

1. 80を含む範囲を中間ノードから探し、そのポインタをたどる
2. ポインタの先が葉ノードであれば、80の値を探し、そのポインタからレコードに到達する



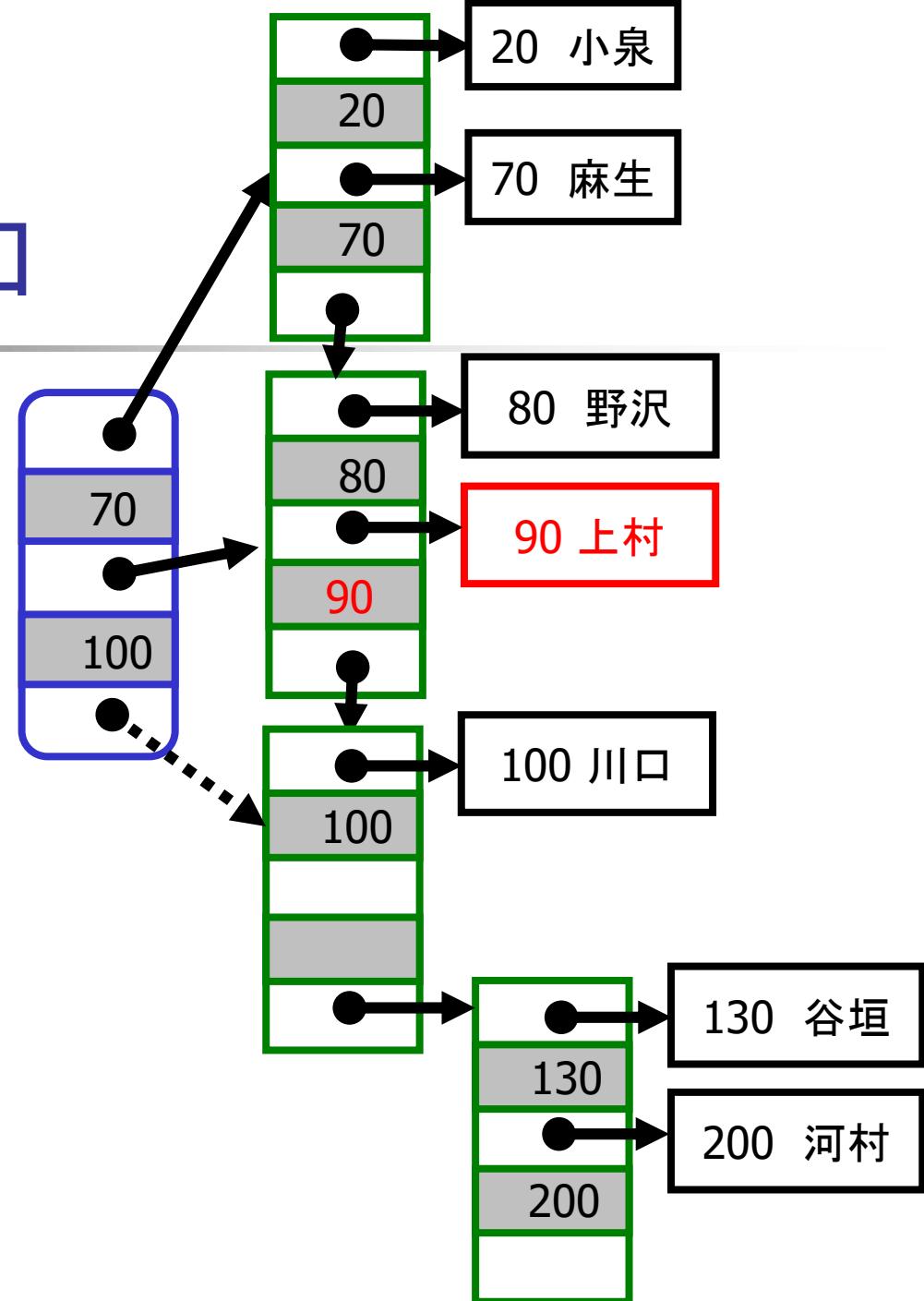
# B+木での レコード追加

社員番号 90 名前 上村 という  
社員のレコードが追加されると....



# B+木での レコード追加

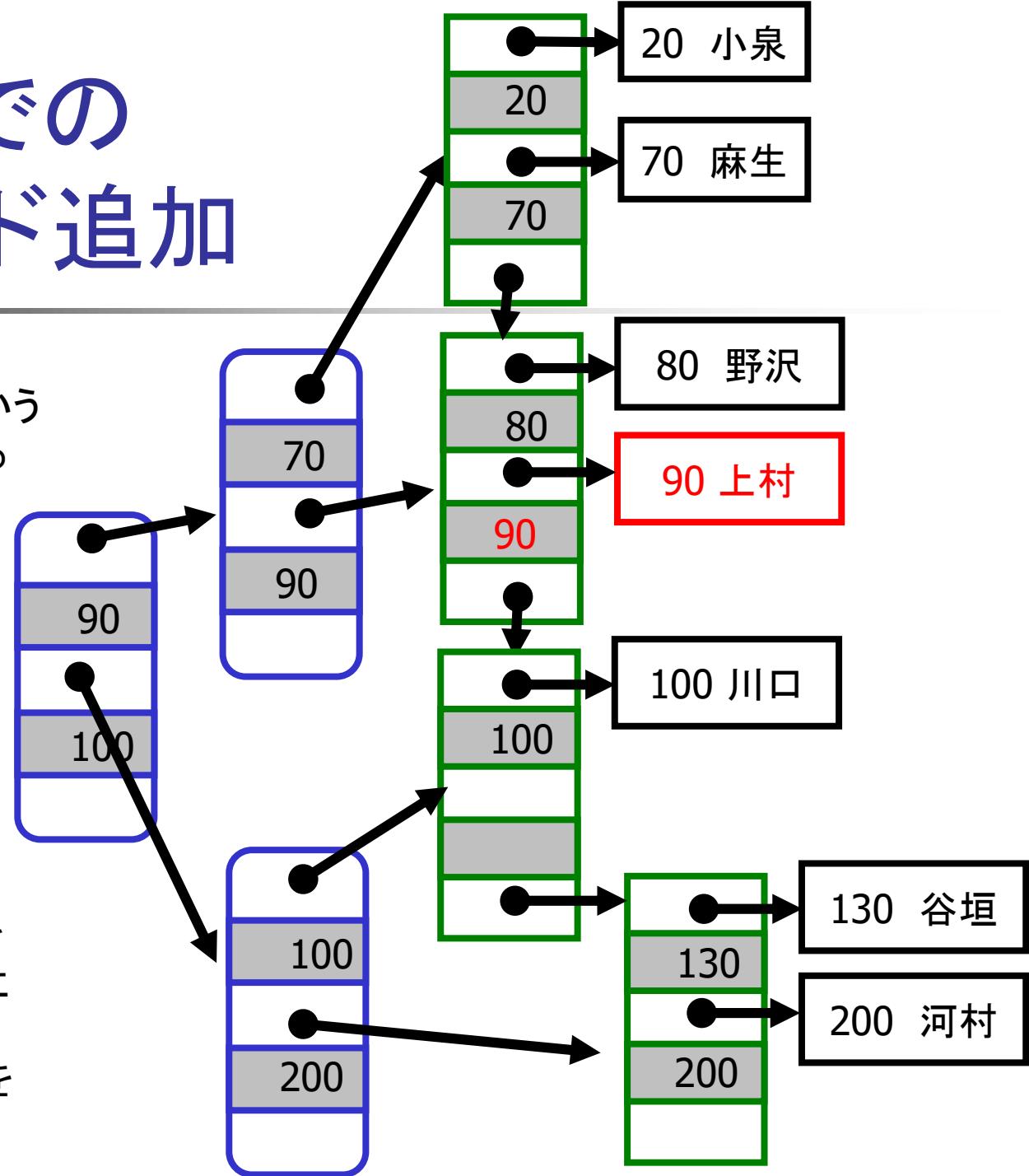
社員番号 90 名前 上村 という  
社員のレコードが追加されると……

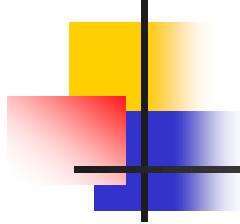


- 空きがないとき葉ノードを分ける。
- 親ノードのポインタの空きが足りなくなったら、さらにその親ノードを追加する。
- 子供の部分木の最大値をキーの値として振る。

# B+木での レコード追加

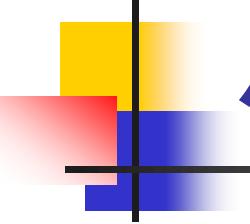
社員番号 90 名前 上村 という  
社員のレコードが追加されると……





# B+木とISAM

- 両方とも多段インデックス
- ISAMは非平衡木だが、B+木は平衡木
- インデックスフィールド
  - ISAMは検索する列の値によりレコードがソートされていることを前提としている
  - B+木は検索する列の値によりレコードがソートされている必要はない



# ハッシュ法

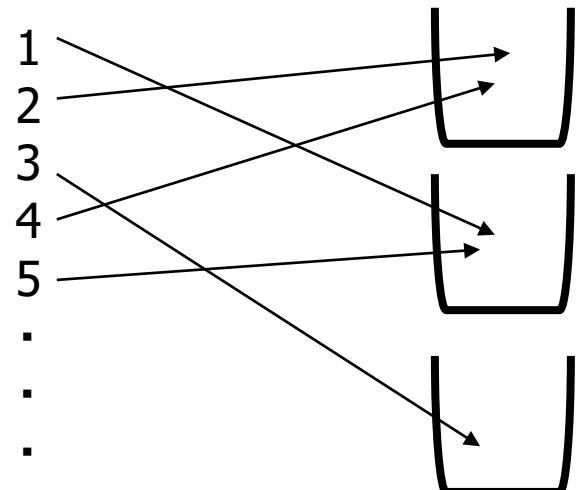
- ハッシュ関数を使って、検索条件に含まれる値から直接、レコードのありかを特定する方法
  - 当然、ファイル(=テーブルの物理的実体)はこれに見合うレコードの持ち方をしていなければならぬ。(=ファイル編成法)
- ハッシュ法を使うときの探索キー(検索条件に含まれる値)をハッシュキーという。
- 静的ハッシュ法と動的ハッシュ法がある。(後述)

# ハッシュ関数

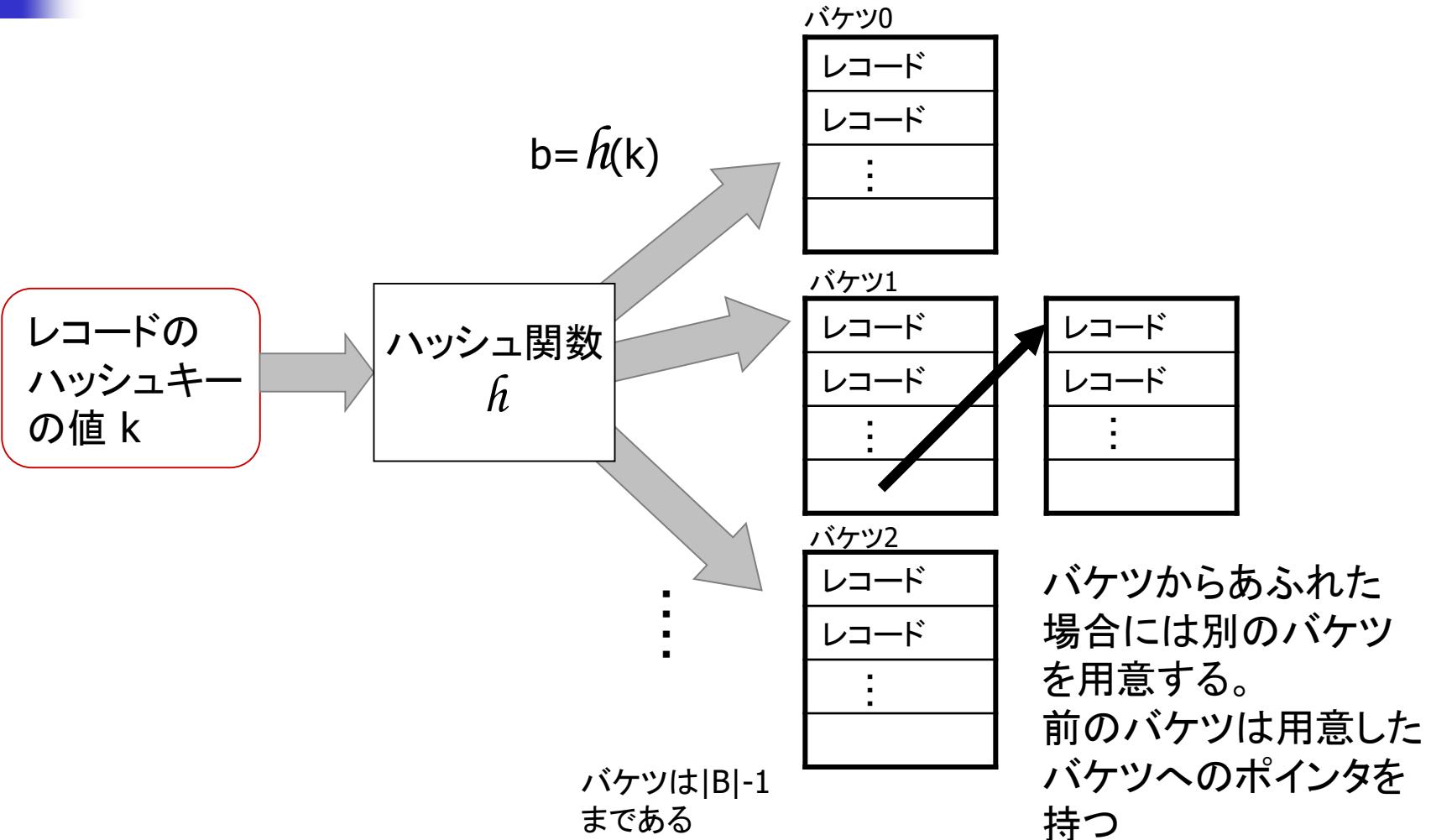
- レコードを格納する器をバケツと呼ぶ
- ハッシュキー(探索キー)の値の集合Kから、バケツの集合Bへの写像は

$$h : K \rightarrow B$$

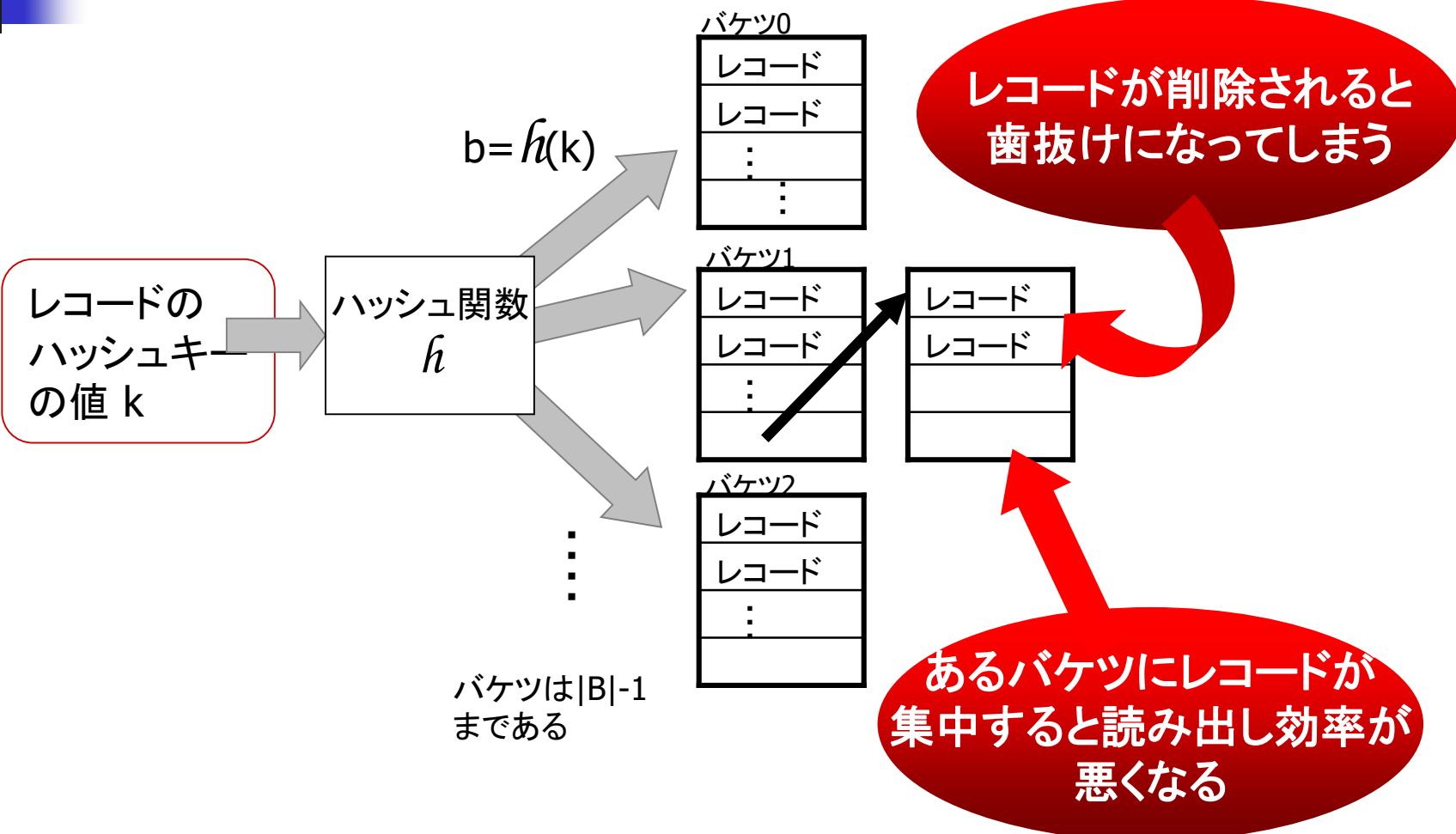
- このような写像のなかで次の条件を満たすものをハッシュ関数と呼ぶ
  - 個々のバケツへ写像されるハッシュキーの数が一定(均一性)
  - ハッシュキーの二つの値が近くても対応するバケツはバラバラ(ランダム性)

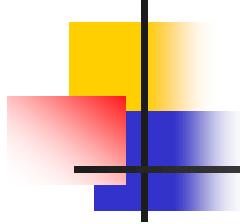


# 静的ハッシュ法



# 静的ハッシュ法の欠点





# 動的ハッシュ法

- ハッシュ関数を動的に変化させる手法
  - 線型ハッシュ法
  - 拡張可能ハッシュ法 ←広く使われている

# 動的ハッシュ法 (拡張可能ハッシュ法)

レコードの  
ハッシュ  
キー  
の値  $k$

$b = h(k)$   
の上位ビット

ハッシュ関数  
 $h$

バケツ帳

バケツ番号	バケツへの ポインタ
000	
001	
010	
011	
100	
101	
110	
111	

Global depth  $d=3$

バケツ0  $d'=1$

レコード
レコード
:

バケツ1  $d'=3$

レコード
レコード
:

バケツ2  $d'=3$

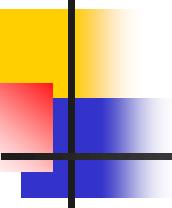
レコード
レコード
:

バケツ3  $d'=2$

レコード
レコード

Local depth

# 拡張可能ハッシュ法でのレコード追加



$b = h(k)$   
の上位ビット  
が  
**0010...**



バケツ帳

バケツ番号	バケツへの ポインタ
000	
001	
010	
011	
100	
101	
110	
111	

Global depth  $d=3$

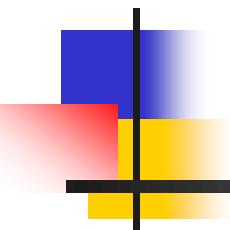
バケツ0-0  $d'=2$

バケツ0-1  $d'=2$

満杯

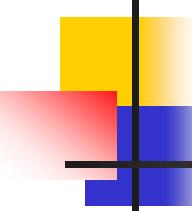
Local depth

⋮



# データベース(第11回)

情報工学科 木村昌臣



# 質問処理とは？

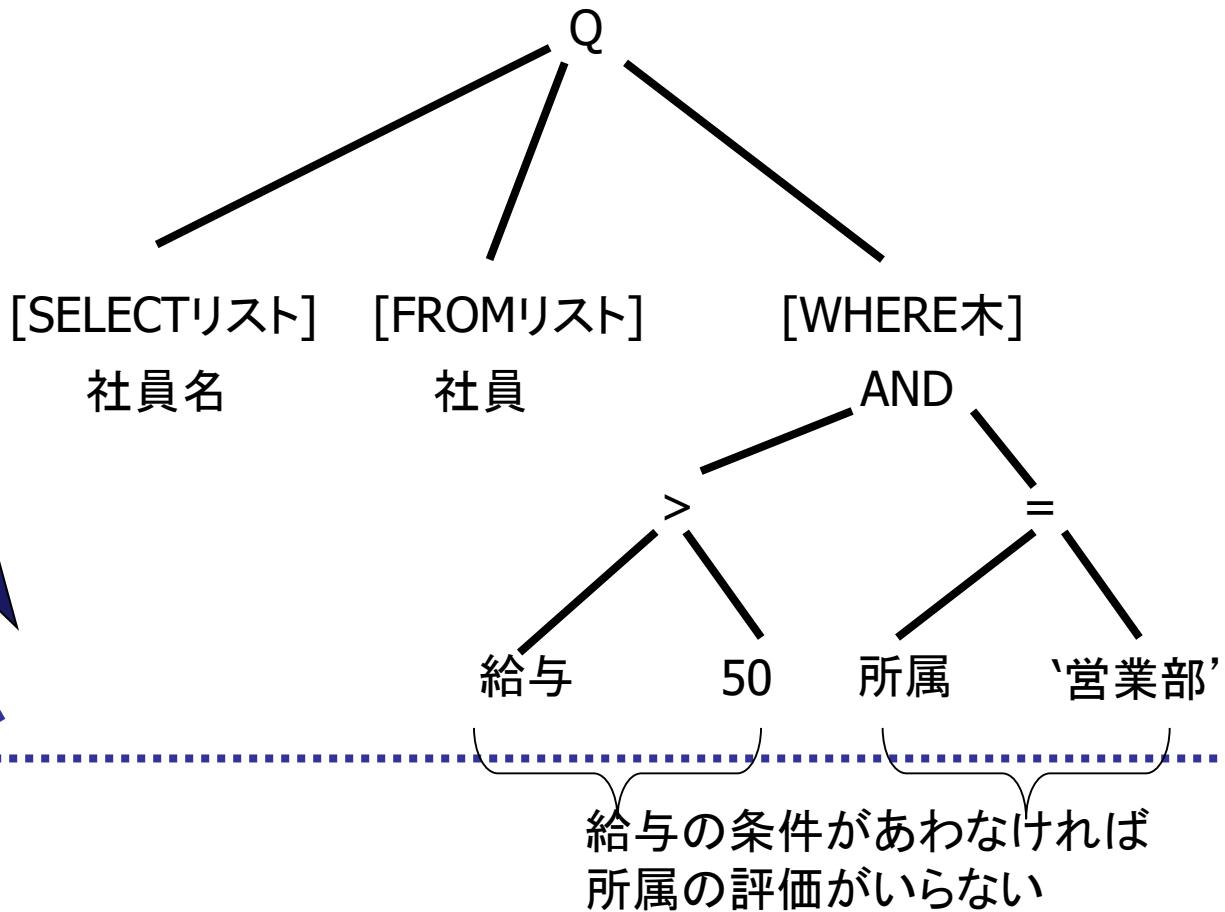
RDBMSの質問処理の質問処理は**非手続き的**  
(それ故) **処理コストが最小の内部スキーマレベルの  
手続き的コードの生成にRDBMSが責任を持たねばならない**

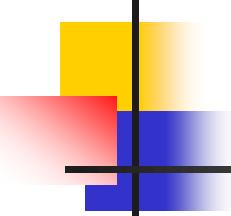
- ① 質問(SQL)の構文解析を行う[構文解析器]
- ② 最適化を行う[最適化器]
- ③ 内部スキーマレベルのオブジェクトコードを  
生成する[コード生成器]
- ④ オブジェクトコードを実行し結果リレーション  
を得る[実行系]

# 質問の構文解析

```
SELECT 社員名  
FROM 社員  
WHERE 給与>50  
      AND 所属='営業部'
```

これを  
最適化器へ





# 質問処理とそのコスト

- 次の基本的な質問について考える
  - 単純質問(simple query)
    - リレーショナル代数の「選択(SELECTION)」に相当
  - 結合質問(join query)
    - リレーショナル代数の「結合(JOIN)」に相当
- コストが高いと質問処理にかかる時間が長い(どこの方法が効率的かを定量化)

# 単純質問処理

```
SELECT *
FROM R
WHERE A='a'
```

→

A	B
b	
c	
a	
:	

どのように処理されるかは、ファイル(=テーブル)Rの編成による。列Aに対しインデックスXAが張られているとすると、

1. インデックスXAで、Aの値が' a'であるレコードを調べる
2. Aの値が' a'であるRのレコードへのポインタ集合Sを求める
3. Sが指示するファイルRのレコード内容をユーザーに返す

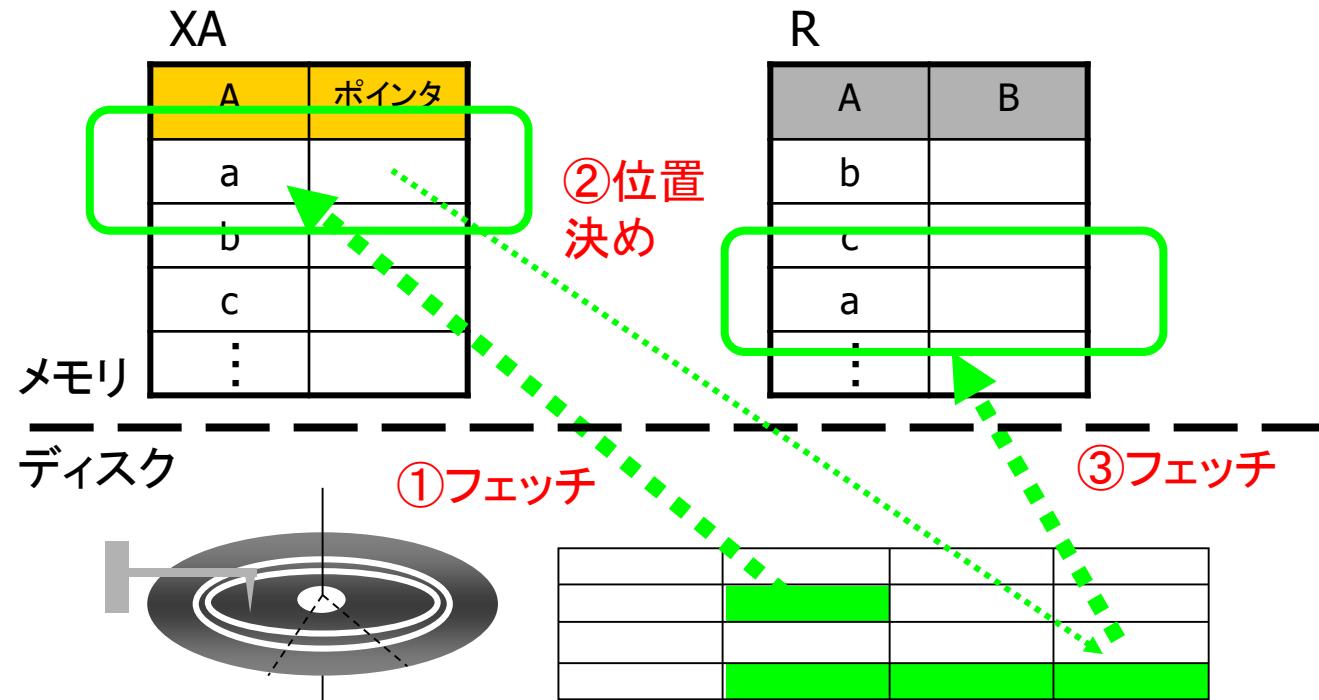
# 単純質問処理とコスト

1. インデックスXAで、Aの値が'a'であるレコードを調べる
2. Aの値が'a'であるRのレコードへのポインタ集合Sを求める
3. Sが指示するファイルRのレコード内容をユーザーに返す

物理的に考えると…

## 考慮すべき点

- テーブルも  
インデックスも  
ディスク上のブロックに  
格納されていること
- ディスクからメモリに  
データを読み込むのは  
大変にコストがかかる  
ということ



# 単純質問の質問処理コスト

- $n_i$ 
  - インデックスXAで、A='a'を満たすポインタ集合を得るためにフェッチしたインデックスページ総数
- $n_d$ 
  - ファイルRで当該レコードを得るためにフェッチしたデータページ総数
- $t_c$ 
  - 処理全体に要したCPU時間
- $w$ 
  - CPU時間をページ数に換算する重み係数



$$\text{コスト } C = n_i + n_d + w \times t_c$$

# 結合質問処理

```
SELECT R.A, R.B, S.C  
FROM R,S  
WHERE R.B=S.B
```



R (1000レコード)

A	B
b	1
c	5
a	3
:	:

S (1000レコード)

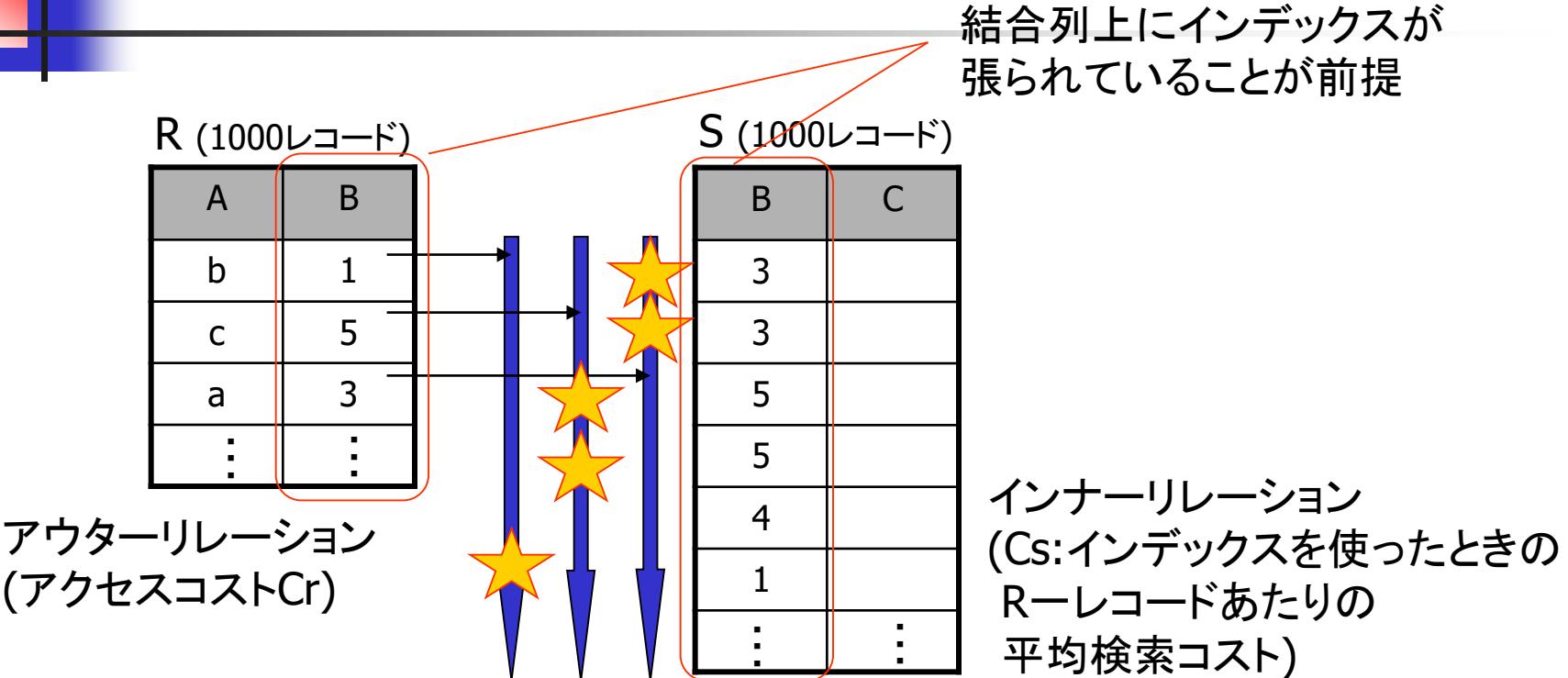
B	C
3	
3	
4	
:	:

理論上は次の操作で上の結合質問処理の結果を得ることができる。

1. リレーションRとSの直積をとる
2. R.BとS.Bの値が等しいレコードを残し中間リレーションを作る
3. この中間リレーションをR.A,R.B,S.Cへ射影する

が!! この中間リレーションはとてつもなく大きなものとなり、質問処理は非効率的になってしまう!!!!!

# 入れ子型ループ(Nested Loop) 結合法



Rで得た各レコードに対し、そのBの値と等しいSのレコードを検索する。  
N: Rのタップルのうち、Sと結合可能なレコード数すると

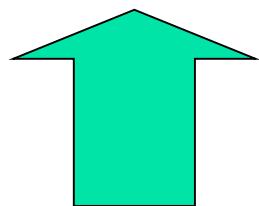
コスト  $C = Cr + N \times Cs + w \times t_c$

# ソートマージ(Merginal Sort)結合法

R (1000レコード)

A	B
b	1
c	5
a	3
d	3
:	:

アウターリレーション



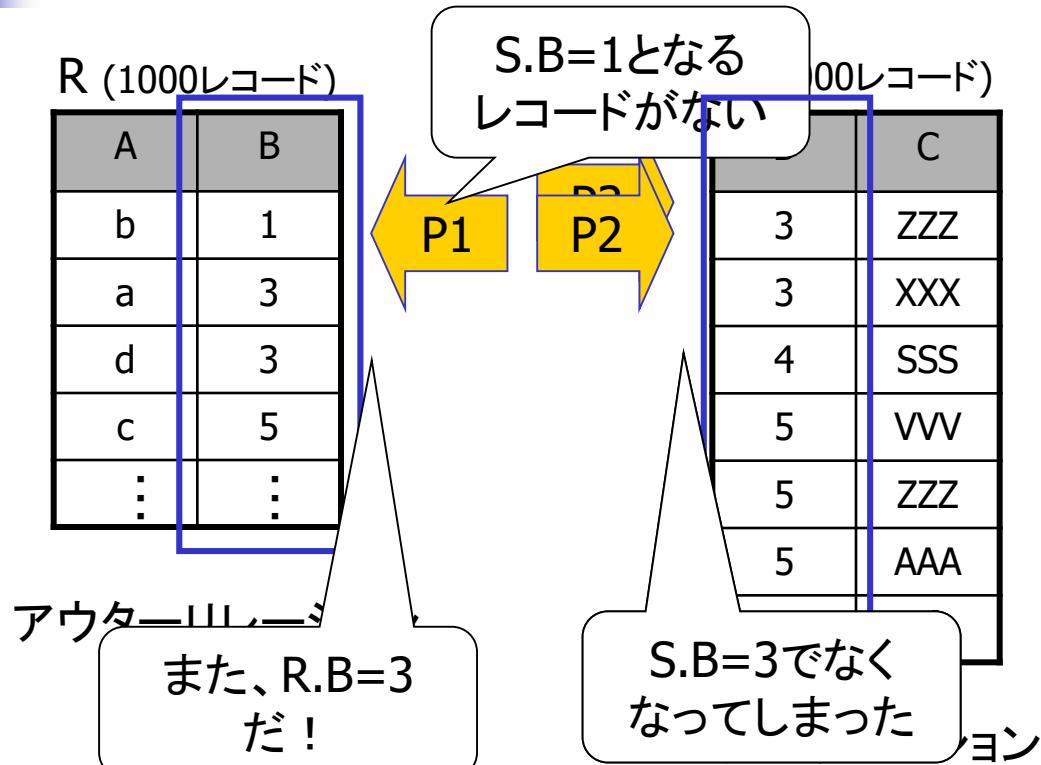
S (1000レコード)

B	C
3	ZZZ
3	XXX
5	VVV
5	ZZZ
5	AAA
4	SSS
:	:

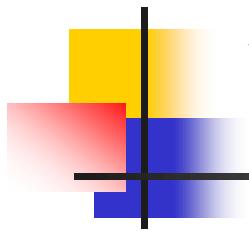
インナーリレーション

まず、Bについて、それぞれのテーブルのレコードをソートする

# ソートマージ(Merginal Sort)結合



P1～P3はレコードを指定するポインタ  
特にP3はプレースフォルダと呼ぶ

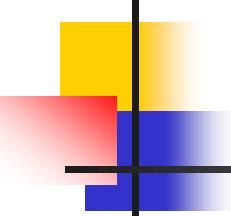


# ソートマージ法の質問処理コスト

- $C_{\text{SORT}}(R), C_{\text{SORT}}(S)$ 
  - それぞれのテーブルのソート処理コスト
- $C_{\text{merge}}(L_R, L_S)$ 
  - $L_R, L_S$ をそれぞれのテーブルをソートしたものとし、それらをマージするコスト

コスト  $C =$

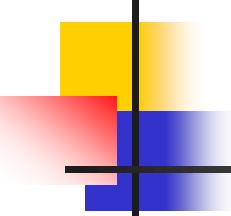
$$C_{\text{SORT}}(R) + C_{\text{SORT}}(S) + C_{\text{merge}}(L_R, L_S) + w \times t_c$$



# 質問処理の最適化

- NL法とMS法のように、与えられたSQLに対し、等価なオブジェクトコードは複数存在する。
- 複数のオブジェクトコードのなかで**コストが最小のものを選ぶべき**
  - コストが最小となるものを実行前に推定する
  - それを「実行プラン」とする

質問処理の最適化



# 質問処理の多様性

- リレーションナルデータモデルレベルに起因する多様性
- リレーションナル演算の処理アルゴリズムレベルに起因する多様性
- ファイルアクセスレベルに起因する多様性

# 質問処理の多様性

リレーションナルデータモデルレベルに起因する多様性

例)

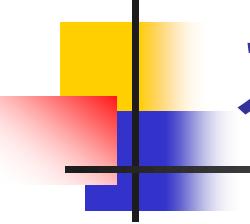
```
SELECT A,B  
FROM R  
WHERE A='a'
```

Rの A, Bへの射影を求める

A='a'を満たすタップルを求める

中間リレーションに対し、  
A='a'を満たすタップルを求める

中間リレーションの  
A, Bへの射影を求める



# リレーションナル演算の処理アルゴリズムレベルに起因する多様性

- 入れ子型ループ結合法
- ソートマージ結合法

のように、  
アルゴリズムレベルでは異なるが、  
リレーションナル代数レベルでは表現できない  
多様性がある。

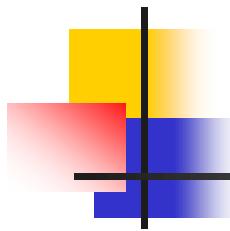
# ファイルアクセスレベルに起因する多様性

R	
A	B
b	1
c	5
a	3
:	:

```
SELECT *
FROM R
WHERE A='a'
AND B='3'
```

- インデックスを使わずテーブルRをスキャンする
- インデックスを使う
  - ◆ インデックスXAを使って処理する
  - ◆ インデックスXBを使って処理する

(RがAについて順次ファイルであれば、Bについては非順次であるなど、インデックスを使っても処理コストは変わることがある)



# 質問処理コストの推定

- 同じSQLを実現するオブジェクトコードが複数あることが解った
- この複数のコードの中で一番コストが低いものを選択し、実行プランとすべき
- 単純質問と結合質問の処理コストの一般的な推定方法について説明する

# 単純質問処理コストの推定(1)

```
SELECT *  
FROM R  
WHERE A='a'
```

R	XA
A	
b	
c	
a	
:	

コスト  $C = n_i + n_d + w \times t_c$

NCARD(R)

= Rの総レコード数

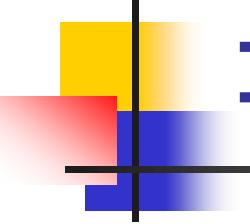
NIDX(X,A)

= インデックスXAのインデックスページの総数

ICARD(X,A)

= Rの異なったA値の数 とすると、

→  $C = (NCARD(R) + NIDX(X,A)) / ICARD(X,A) + w \times t_c$



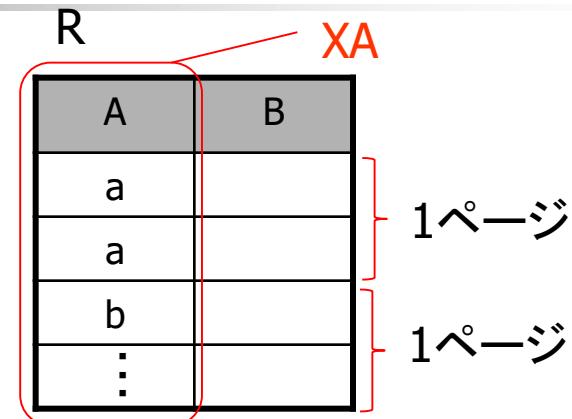
## コストの内訳

- $A='a'$ となるレコードが含まれる平均ページ数  
=平均的な $A='a'$ となるレコード数  
=全レコード / 値の種類  
$$NCARD(R) / ICARD(X,A)$$
- $A='a'$ となるインデックスが含まれる平均ページ数  
$$NINDX(X,A)/ICARD(X,A)$$

# 単純質問処理コストの推定(2)

```
SELECT *  
FROM R  
WHERE A='a'
```

$$\text{コスト } C = n_i + n_d + w \times t_c$$



$A$ が順序フィールドとなっており、 $A='a'$ となる複数レコードが1ページに含まれる場合

$\text{TCARD}(R)$  =  $R$ のレコードを含むページ総数

$\text{NINDX}(X,A)$  = インデックス  $XA$  のインデックスページの総数

$\text{ICARD}(X,A)$  =  $R$ の異なった  $A$  値の数 とすると、

$$C = (\text{TCARD}(R) + \text{NINDX}(X,A)) / \text{ICARD}(X,A) + w \times t_c$$

# 結合質問処理コストの推定

```
SELECT 社員.社員番号, 社員.所属  
FROM 社員, 部門  
WHERE 社員.所属=部門.部門番号  
AND 社員.職種='SE'  
AND 部門.所在地='横浜'
```

社員番号	所属	職種
001	K11	SE
003	K11	SE
025	K32	営業
012	K12	営業
:	:	

部門番号	所在地
K11	横浜
K12	東京
K32	大宮
:	:

この例での  
仮定

~~Cscan(社員)~~ > Cindex(社員.所属) > Cindex(社員.職種)

Cindex(部門.部門番号) > Cscan(部門)

入れ子型ループ法  
を使うときには要る

データページが少量で  
インデックスページもフェッチする  
よりコストが低い場合

# 入れ子型ループ法

アウターリレーション

(社員, 部門)

インナーリレーション

(部門, 社員)

インデックス  
X社員.職種

N<sub>1</sub>

インデックス  
X部門.部門番号

N<sub>3</sub>

①

Cindex(社員.職種)  
+N<sub>1</sub> × Cindex(部門.部門番号)

スキャン  
部門

N<sub>2</sub>

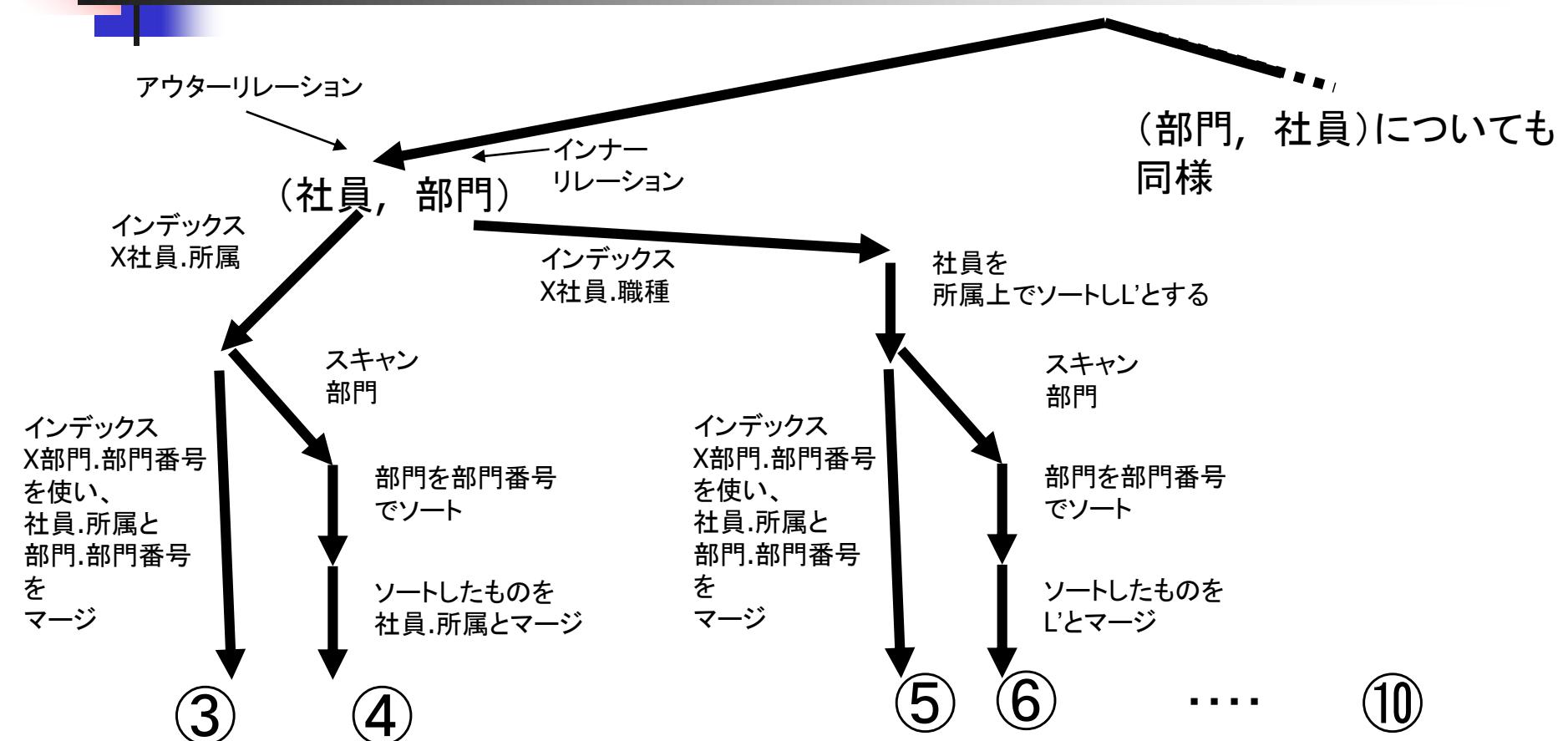
インデックス  
X社員.所属

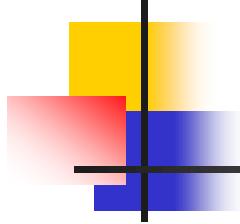
N<sub>3</sub>

②

Cscan(部門)  
+N<sub>2</sub> × Cindex(社員.所属)

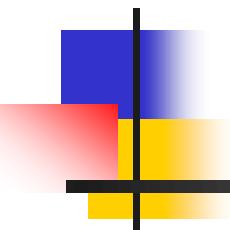
# ソートマージ結合法





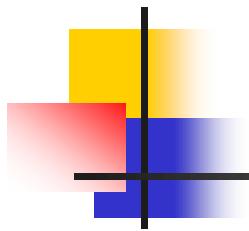
# 最適化

- ①～⑩の検索方法に相当するコストを統計情報等から算出
- 算出したコストのなかで一番低いものを、「実行プラン」として採用する。



# データベース(第12回)

情報工学科 木村昌臣



# トランザクションとは

データベースに対する  
アプリケーションプログラムレベルでの  
ひとつの原子的な作用のこと



原子的…これ以上分解できること  
作用 …データを読み書きする操作のこと

# トランザクションとは

データベースに対する

アプリケーションプログラムレベルでの

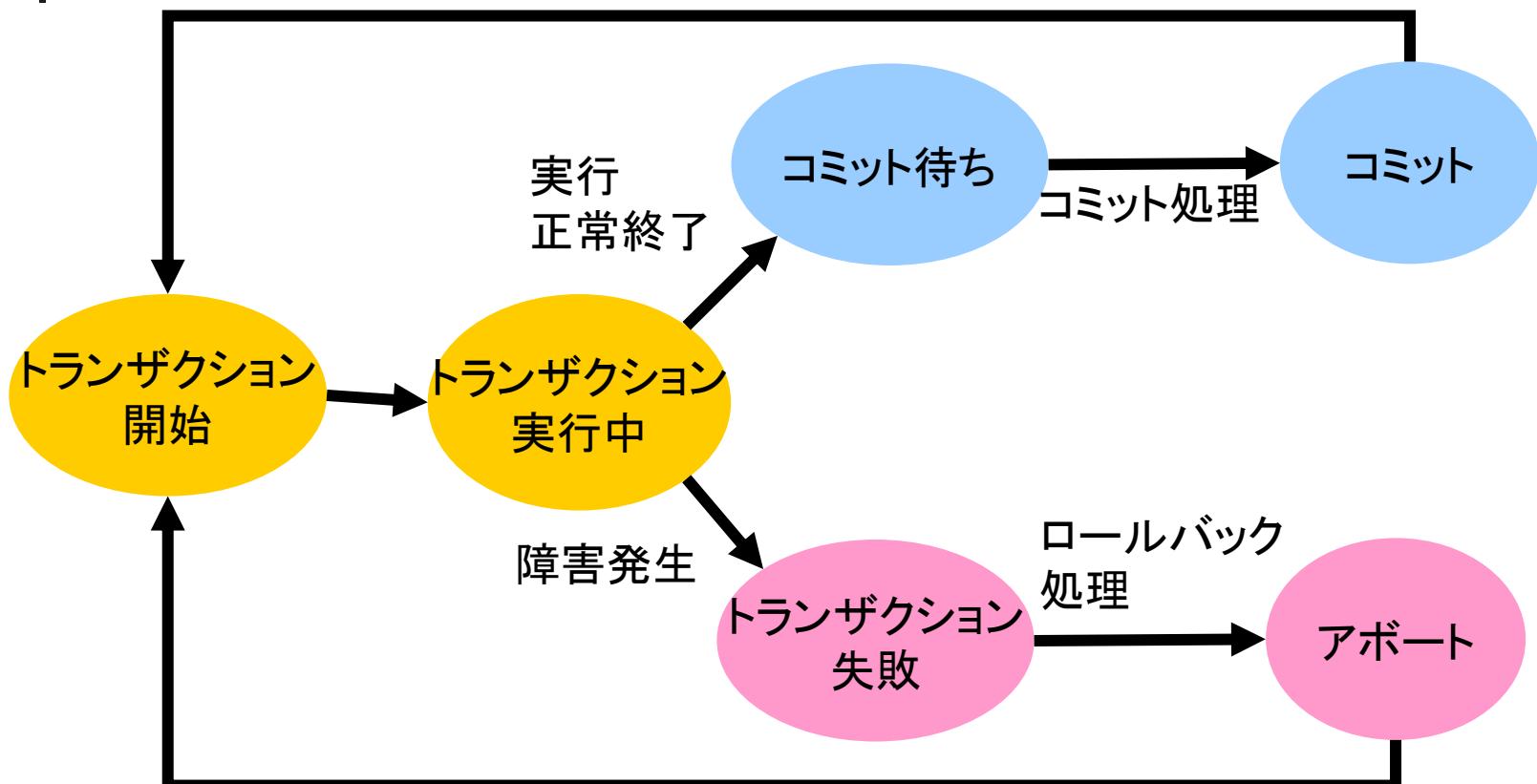
ひとつの原子的な作用のこと



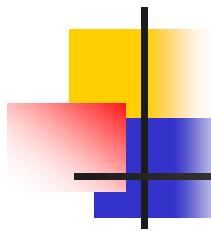
たとえば、A社が自社の口座からB社に100万円振り込む場合、最低UPDATE文を二つ発行する必要がある。

この場合、**アプリケーションプログラムレベル**では、各々のSQL単独では意味がなく、二つセットで意味がある。

# トランザクションの状態遷移



この時点で、データベースはトランザクション開始前の状態に戻る

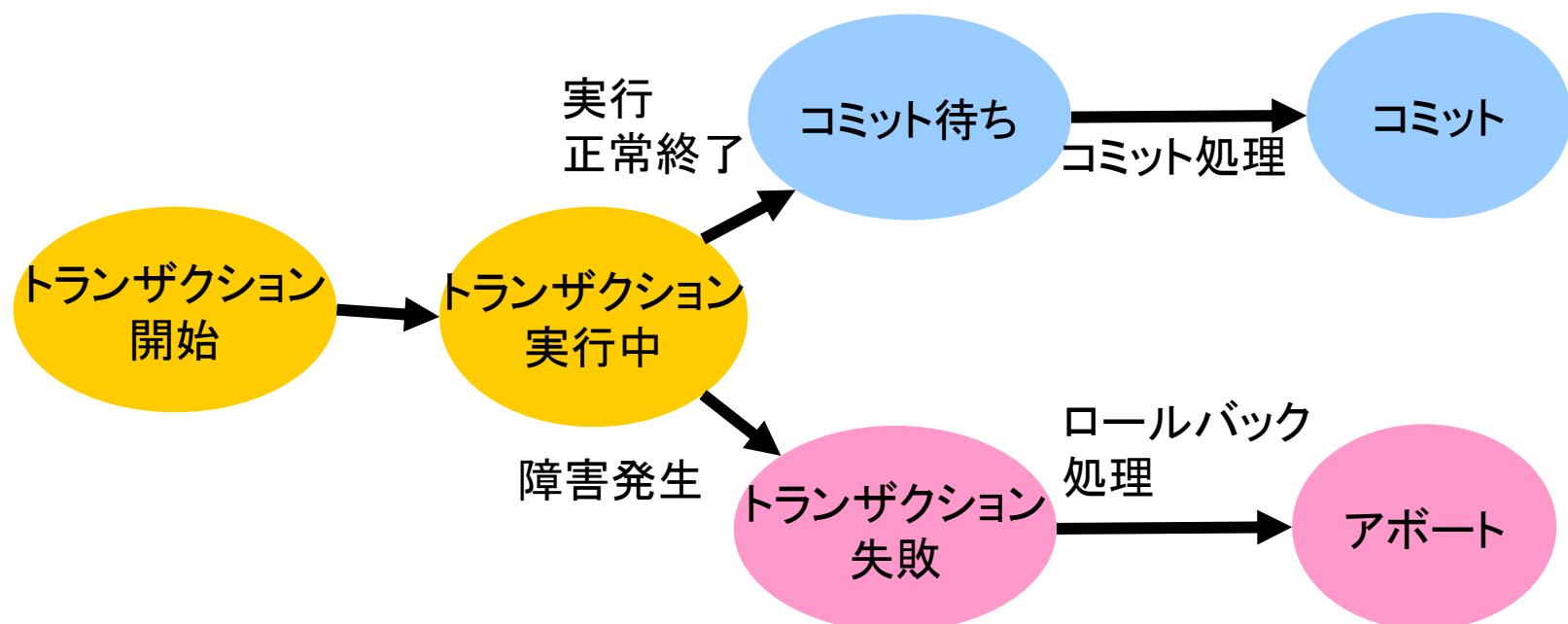


# ACID特性

- トランザクションを実行する際に保証されるべき性質
  - 原子性 (Atomicity)
  - 一貫性 (Consistency)
  - 隔離性 (Isolation)
  - 耐久性 (Durability)

# 原子性(Atomicity)

- トランザクション内の全処理が成功してコミットされるか、ひとつでも失敗してロールバックされるかのどちらかでなければならぬ



※トランザクション内の処理は分けて実行されることはない(all or nothing!)

# 一貫性(Consistency)

- トランザクション終了時点において、必ずアプリケーションレベルでデータが「正しい」状態を維持されていること
- 実行前後で整合性を持っていること

A社からB社へ100万円振り込み送金をするトランザクション

一貫している



口座	残高
A	500
B	200

Aから  
100引く



一貫していない



口座	残高
A	400
B	200

Bへ  
100足す

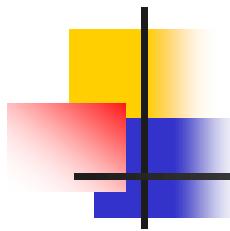


一貫している



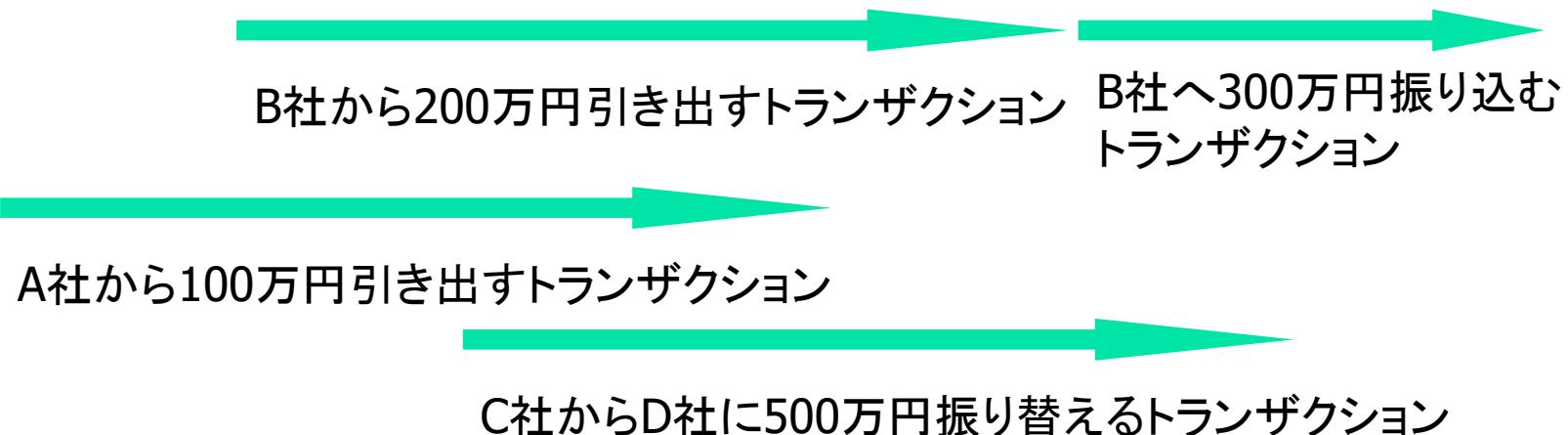
口座	残高
A	400
B	300

※データが中途半端な状態で処理が終わることはない



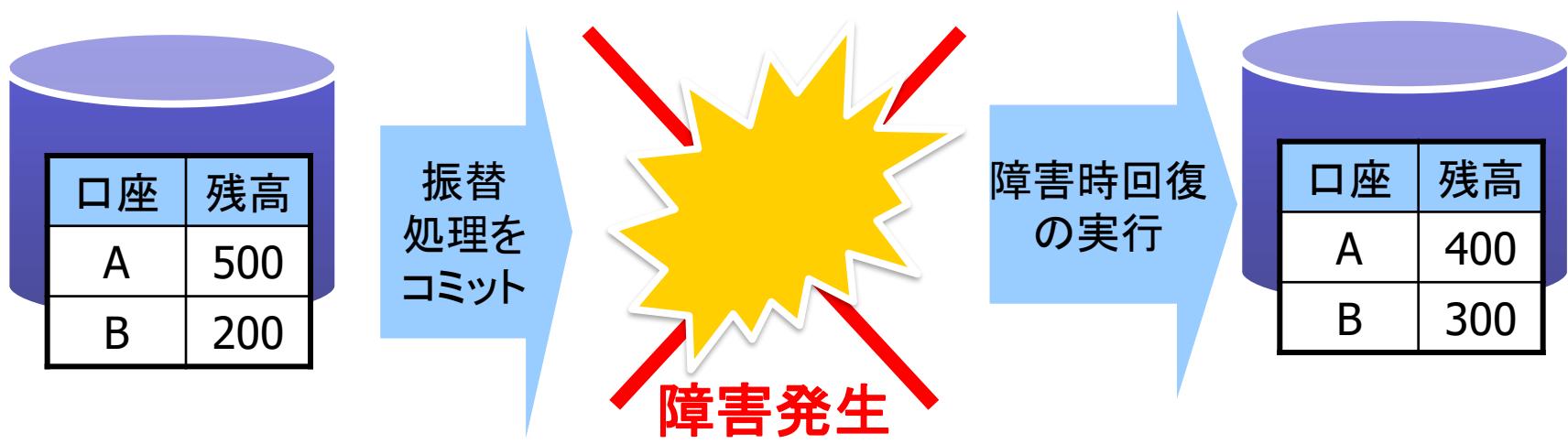
# 隔離性(Isolation)

- 個々のトランザクションが、他に処理されているトランザクションの影響を受けず実行されること



# 耐久性(Durability)

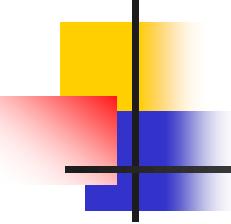
- トランザクションが一度コミットされれば、その実行中に生じたデータベースへの変化は、いかなる障害がおきても失われることがない、ということ



A社からB社へ100万円振替送金をするトランザクション

# トランザクション指向の障害時回復

- トランザクション実行時にいろいろな障害が起こる可能性がある
  - トランザクション障害
  - システム障害
  - メディア障害
- 耐久性を保証するため、これらの障害からデータベースを守り、一貫性を保証する必要がある。  
(トランザクション指向の障害時回復)

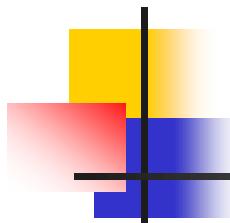


# 障害の分類

- トランザクション障害
  - データが見つからない、計算途中でのオーバーフローの発生などによって、トランザクションが異常終了する場合
- システム障害
  - DBMSやOSレベルの障害(暴走など)、ハードウェアの誤動作による電源断などの場合
- メディア障害
  - ディスククラッシュなど、2次記憶装置の障害



# ログを使った障害時回復



# 実際の障害時回復方法

- ログ法
- シャドウページ法

一般に使われているデータベースは  
ログ法を使っているので、本講義ではログ法のみ  
説明する

# ログ法

T1

T2

```
SELECT 給与  
FROM 社員  
WHERE EMPNO='007'
```

```
UPDATE 社員  
SET 給与=100  
WHERE EMPNO='007'
```

commit

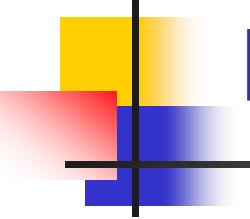
```
UPDATE 社員  
SET 給与=80  
WHERE EMPNO='002'
```

commit

# ログ

Log SN	TransID	作用	表 ID	Rec ID	列 ID	旧 値	新 値
100	T1	開始					
101	T1	読み込	社員	007	給与		
102	T2	開始					
103	T1	書出	社員	007	給与	90	100
104	T2	書出	社員	002	給与	75	80
105	T1	コミット					
106	T2	コミット					

トランザクションの一連の作用などを  
時系列にそってすべて記録

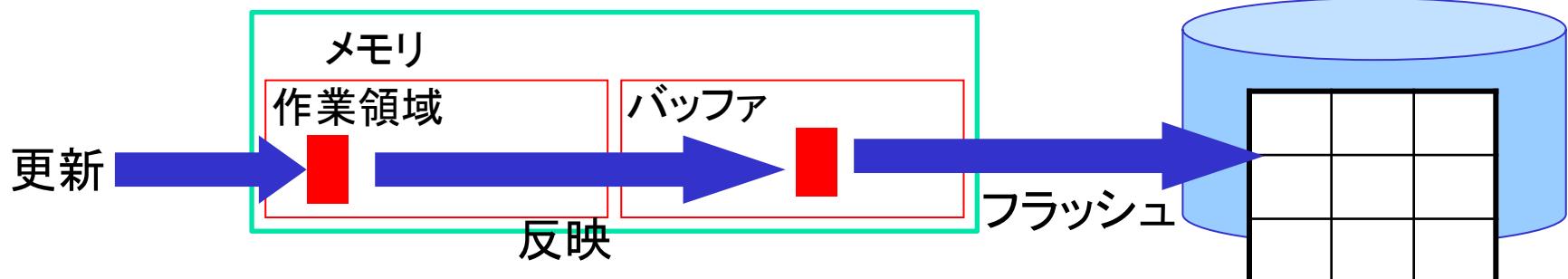


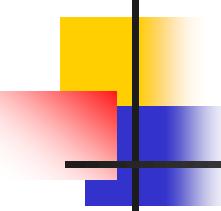
# ログ

- トランザクションがデータベースに対して行った一連の作用をすべて記録した時系列データファイル
- このデータをもとに、REDOやUNDOを実施
- 障害回復の手がかりとなる重要なものなので、保管は二重化されたディスクなどに保存

# ログのディスクへの書き込みタイミング

- 一般に、実際のデータを更新する場合、メモリ上で変更されたものが、**バッファに反映**され、ディスクにフラッシュされて変更が終了
- フラッシュはOS任せなので、そこで障害が起きたらアウト
- ログは障害回復の要なので、更新が発生したら強制的にディスクにフラッシュ





# WALプロトコル

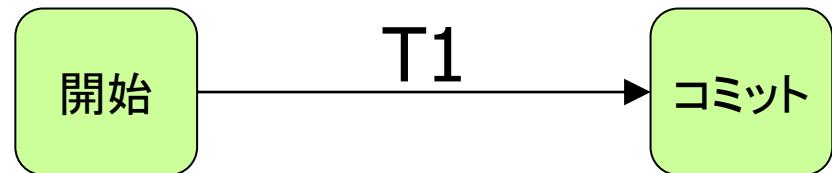
- 以下の二つは独立に行われる
  1. データベースバッファを経由するデータベース更新作業
  2. ログバッファを経由するログの更新作業
- 1.が先行すると、障害が発生した時点でログに手がかりがないので困る
- まずログに書き、その後データベースを更新するべき。

**WAL(Write Ahead log)プロトコル**

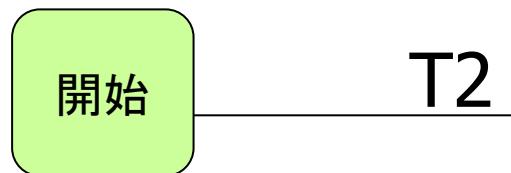
# データベースに障害が起こった ら…

更新途中で  
障害が発生！  
どうしよう？

障害発生



コミット済みなので  
ログには書き込み済み  
だが、データベースへの  
書き込みは途中かも？



コミット前なのでログへの  
書き込みも  
まだ完了していない！

時間

# 障害時回復の基本

## ■ UNDO

- トランザクション前の一貫性がある状態に戻す

## ■ REDO

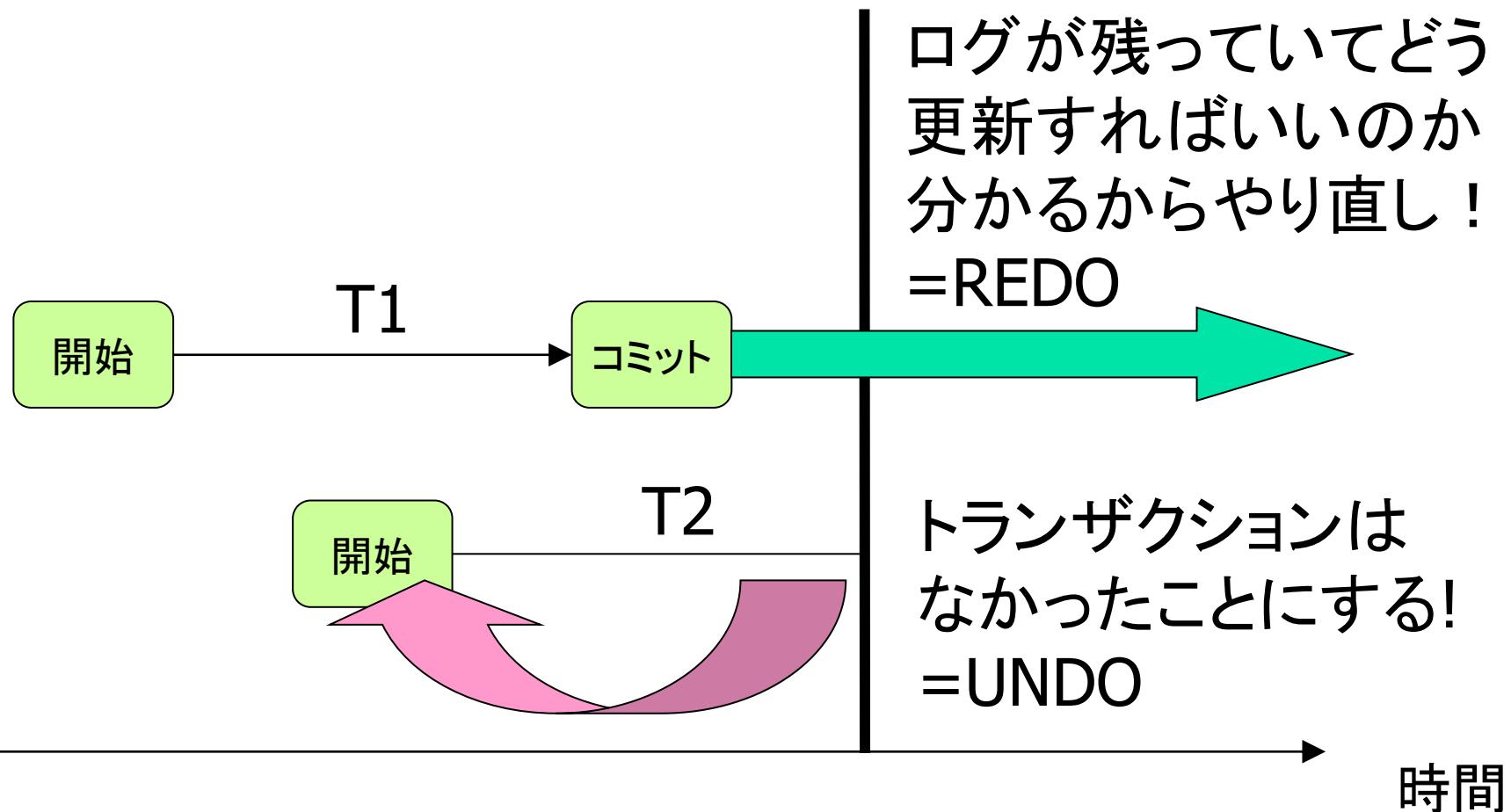
- トランザクションをコミット後に遷移するはずだった一貫性がある状態に遷移する

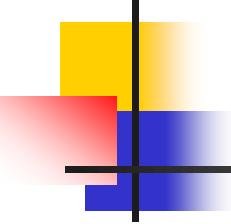


※コミット、ロールバックとは異なる概念であるので混同しないこと

# UNDO/REDO障害時回復法

障害発生時点



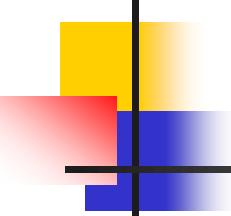


# 障害の分類

- トランザクション障害
  - データが見つからない、計算途中でのオーバーフローの発生などによって、トランザクションが異常終了する場合
- システム障害
  - DBMSやOSレベルの障害(暴走など)、ハードウェアの誤動作による電源断などの場合
- メディア障害
  - ディスククラッシュなど、2次記憶装置の障害

# トランザクション指向の障害時回復の分類

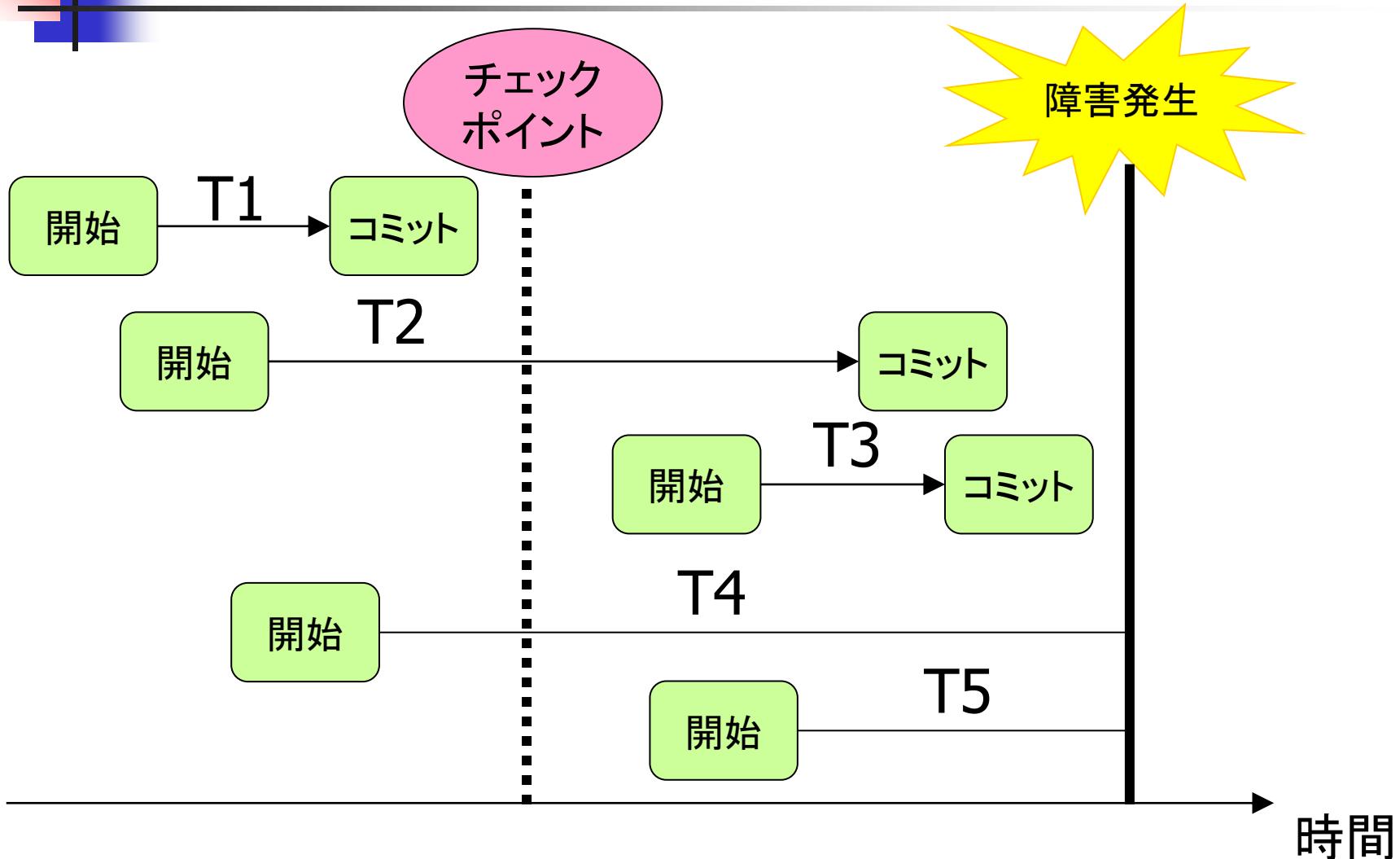
- トランザクションUNDO
  - トランザクション障害が発生した場合、異常終了したトランザクションをUNDOする
- 全局的UNDO
  - システム障害発生時点で、まだ終了していなかったトランザクションをシステム再スタート時にUNDOする
- 局所的REDO
  - コミットはされていたがシステム障害により完全にデータベースが更新されていない場合、そのトランザクションをシステム再スタート時にREDOする
- 全局的REDO
  - メディア障害発生により、ディスクをとりかえて保管用ダンプ(バックアップ)から、ログなどを使ってそれ以降にコミットしたすべてのトランザクションをREDOする

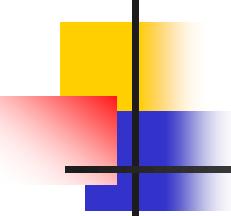


# チェックポイント法

- チェックポイント
  - コミットされたトランザクションについて、その時点できちんとデータベースへの書き込みが終わっていることを保証する時点
  - 以下の四つの作業を行う
    - 実行中の全トランザクションを一時停止
    - データベースバッファの内容をデータベースへ強制書き出し
    - チェックポイントレコードをログに書き出し
    - 停止していたトランザクションを再開

# チェックポイント法 (次の場合どうしたらしいい?)

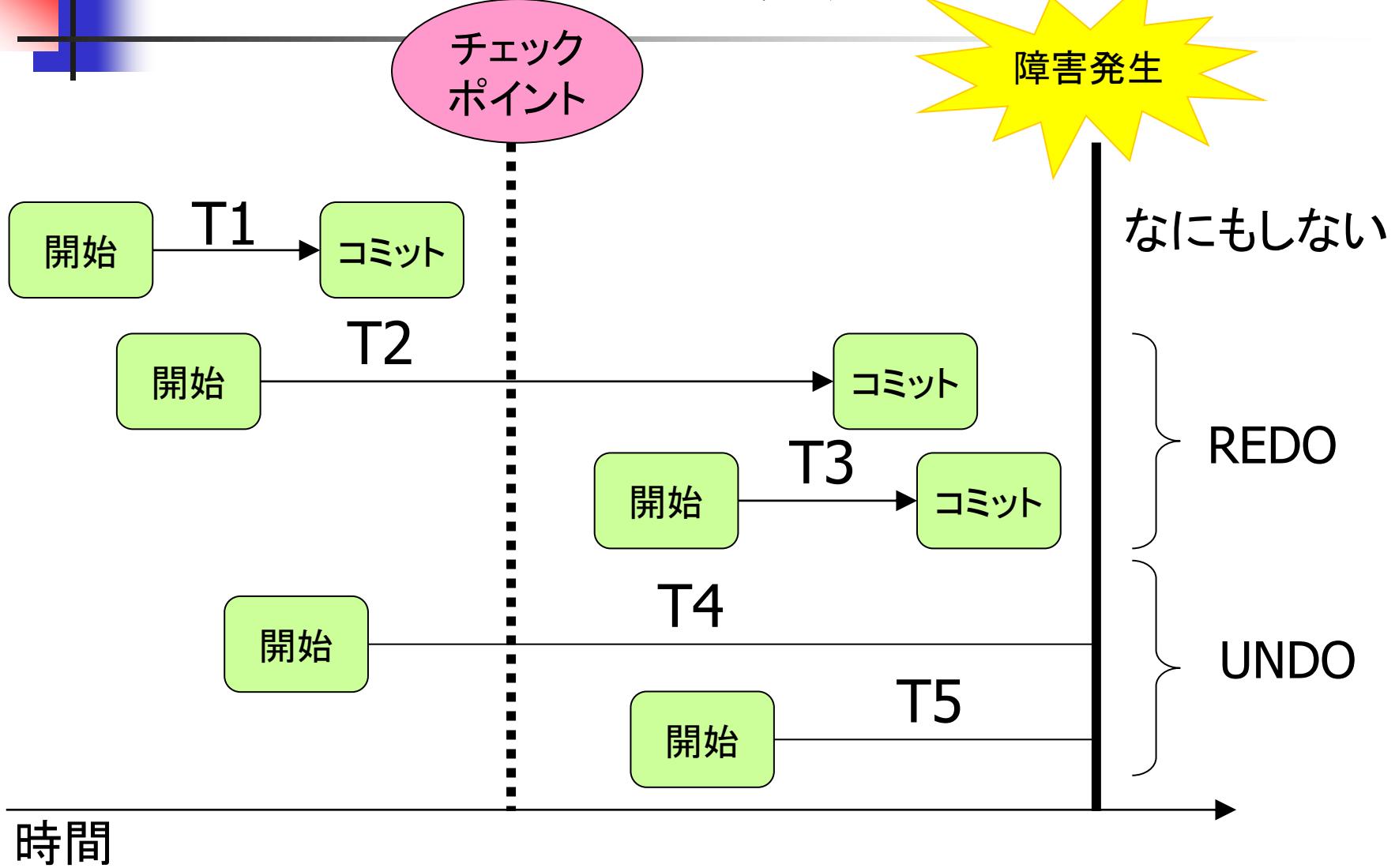


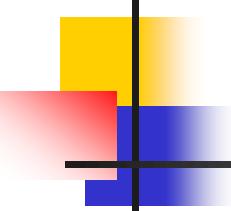


# チェックポイント法

- チェックポイント前にコミット済みであれば、すでにデータベースへの書き込みが完了しているため、全てのコミット済みトランザクションについてREDOする必要なし
- チェックポイント後についてはUNDO/REDO法と同じ考え方
  - チェックポイント前にコミット済みであれば、すでにデータベースへの書き込みが完了しているので**そのまま**
  - チェックポイント後にコミット済みであれば、**REDO**
  - 未コミットであれば**UNDO**

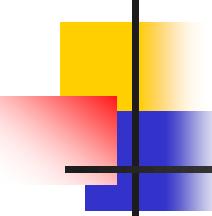
# チェックポイント法





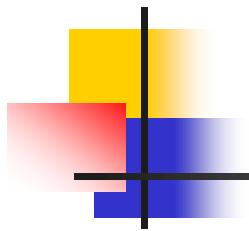
# チェックポイント法の注意点

- チェックポイントの頻度は要件に応じて適切に決める必要がある(トレードオフ)
  - チェックポイントの頻度が多いと、通常のデータベース処理の処理効率が落ちる
  - チェックポイントの頻度が少ないと、障害後に回復するまでの時間がかかる



# ACID特性

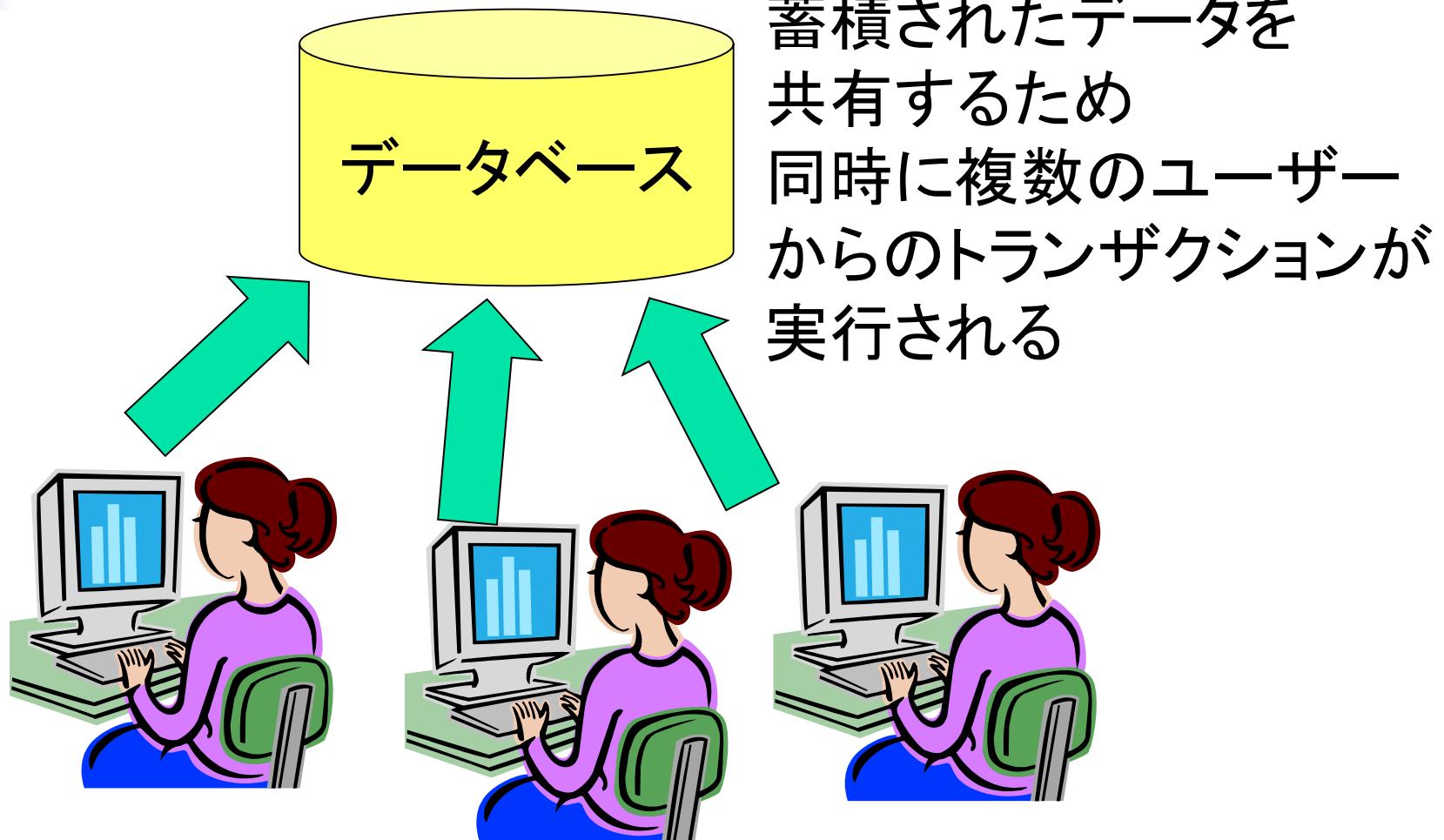
- トランザクションを実行する際に保証されるべき性質
  - 原子性 (Atomicity)
  - 一貫性 (Consistency)
  - 隔離性 (Isolation)
  - 耐久性 (Durability)

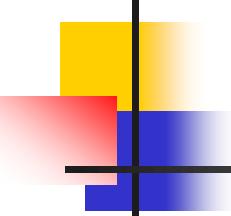


# DBMSの三大機能

- メタデータ管理
  - 質問処理
  - トランザクション管理
    - 障害時回復
    - 同時実行制御
- } 今日はここ!

# 同時実行制御の必要性





# 今回の授業での約束

## ■ トランザクションの簡易記法

- 開始:begin
- レコードAの読み取り:read(A)
- レコードAへの書き出し:  
write(A:=2\*A) など
- 終了:end

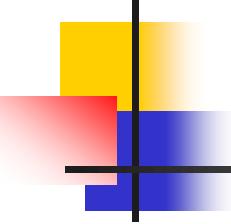
通常は、あるレコードについて値を読み取ってから計算等の処理を行い結果を書き出す。  
`write(A:=2*A)` はレコードAの値を2倍して書き出すことを表す

口座番号	金額
....	....
A	250,000
B	10,000
C	3,400
D	100,000
....	....

# ちゃんとトランザクションの同時実行制御をおこなわないと…

レコードAが50で、  
30と20を別々のトランザクションで引きたいが、  
結果は30になってしまふ（遺失更新異常）

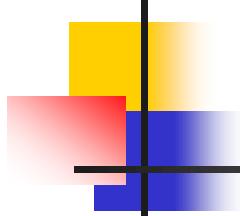
時刻	トランザクションT1	トランザクションT2
1	read(A)	
2		read(A)
3	write(A:=A-30)	
4		write(A:=A-20)
5	COMMIT	
6		COMMIT



# 正しく更新を行うには

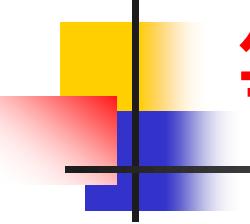
前ページではT1の途中の状態の値をT2が使ったので  
正しくない処理になってしまった。

時刻	トランザクションT1	トランザクションT2
1	read(A)	
2	write(A:=A-30)	
3		read(A)
4		write(A:=A-20)
5	COMMIT	
6		COMMIT



# 直列化可能スケジュール

- 一貫性を保証するためには、トランザクションを1つずつ実行するのが確実(直列スケジュール)
- ただし、性能を向上させるには複数のトランザクションを同時に動かしたい
  - 直列スケジュールと「等価」であれば同時に動かしても大丈夫(直列化可能スケジュール)



# 等価であるべき条件(ビュー等価)

- 同一レコードにアクセスするとトランザクションについて
  1. あるトランザクション( $T_1$ )の書き込み(write)の結果を別のトランザクション( $T_2$ )が読み込んでいる(read)場合、その順番は変わってはいけない
  2. 最後の書き込み(write)は必ず最後に書き込まれなければならない

# 条件1

時刻	トランザクションT1	トランザクションT2
1	read(A)	
2		read(A)
3	write(A:=A-30)	
4		write(A:=A-20)
5	COMMIT	
6		COMMIT

時刻	トランザクションT1	トランザクションT2
1	read(A)	
2	write(A:=A-30)	
3		read(A)
4		write(A:=A-20)
5	COMMIT	
6		COMMIT

readとwriteの順番を変えるとT2の読み込む値が変わってしまい、等価とはいえない

# 条件2

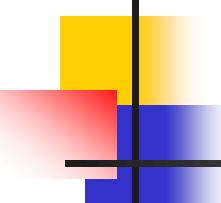
Aの初期値が  
100とすると  
結果は80

時刻	トランザクションT1	トランザクションT2
1	read(A)	
2		read(A)
3	write(A:=A-30)	
4		write(A:=A-20)
5	COMMIT	
6		COMMIT

readして得た  
値は同じでも  
writeの順番  
が変わると  
最終的な値が  
変わってしま  
い、等価とは  
いえない

時刻	トランザクションT1	トランザクションT2
1	read(A)	
2	write(A:=A-30)	
3		read(A)
4		write(A:=A-20)
5	COMMIT	
6		COMMIT

Aの初期値が  
100とすると  
結果は50



# 相反グラフ

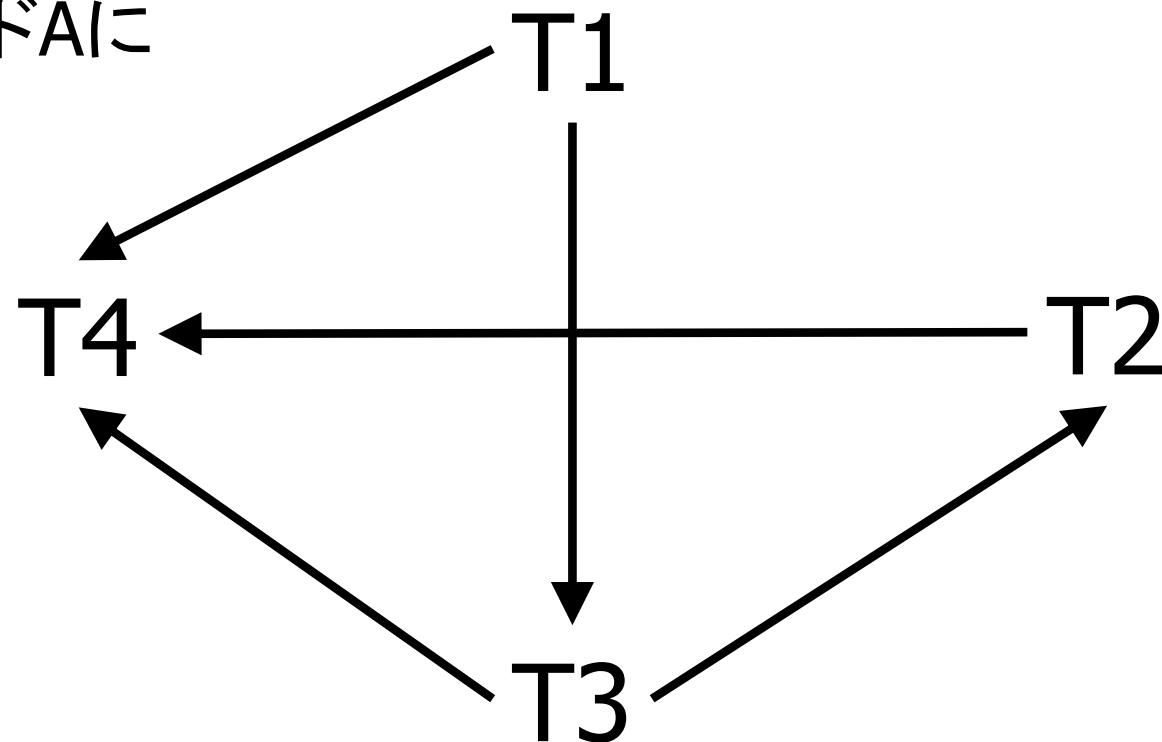
- ビュー直列化可能であるための判定法(正確には十分条件)
  - トランザクションをノードとする
  - 以下の条件を満たす時、有向リンクでノードを結ぶ
    - $T_i$ の $\text{read}(A)$ が $T_j$ の $\text{write}(A)$ より先に実行される
    - $T_i$ の $\text{write}(A)$ が $T_j$ の $\text{read}(A)$ より先に実行される
    - $T_i$ の $\text{write}(A)$ が $T_j$ の $\text{write}(A)$ より先に実行される
  - 非巡回グラフであれば、ビュー直列化可能(相反直列化可能)

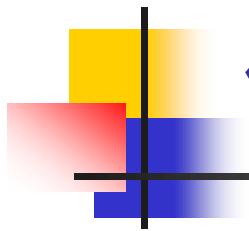
# 例題(相反グラフ)

時刻	T1	T2	T3	T4
1			read(A)	
2	read(A)			
3			write(A)	
4		read(A)		
5	read(B)			
6				read(A)
7	write(B)			
8		read(B)		
9				write(A)
10		write(B)		

# 例題(相反グラフ)-1

レコードAに  
着目





## 例題(相反グラフ)-2

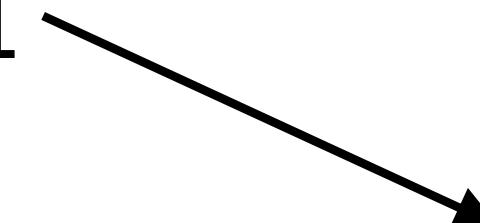
レコードBに  
着目

T4

T1

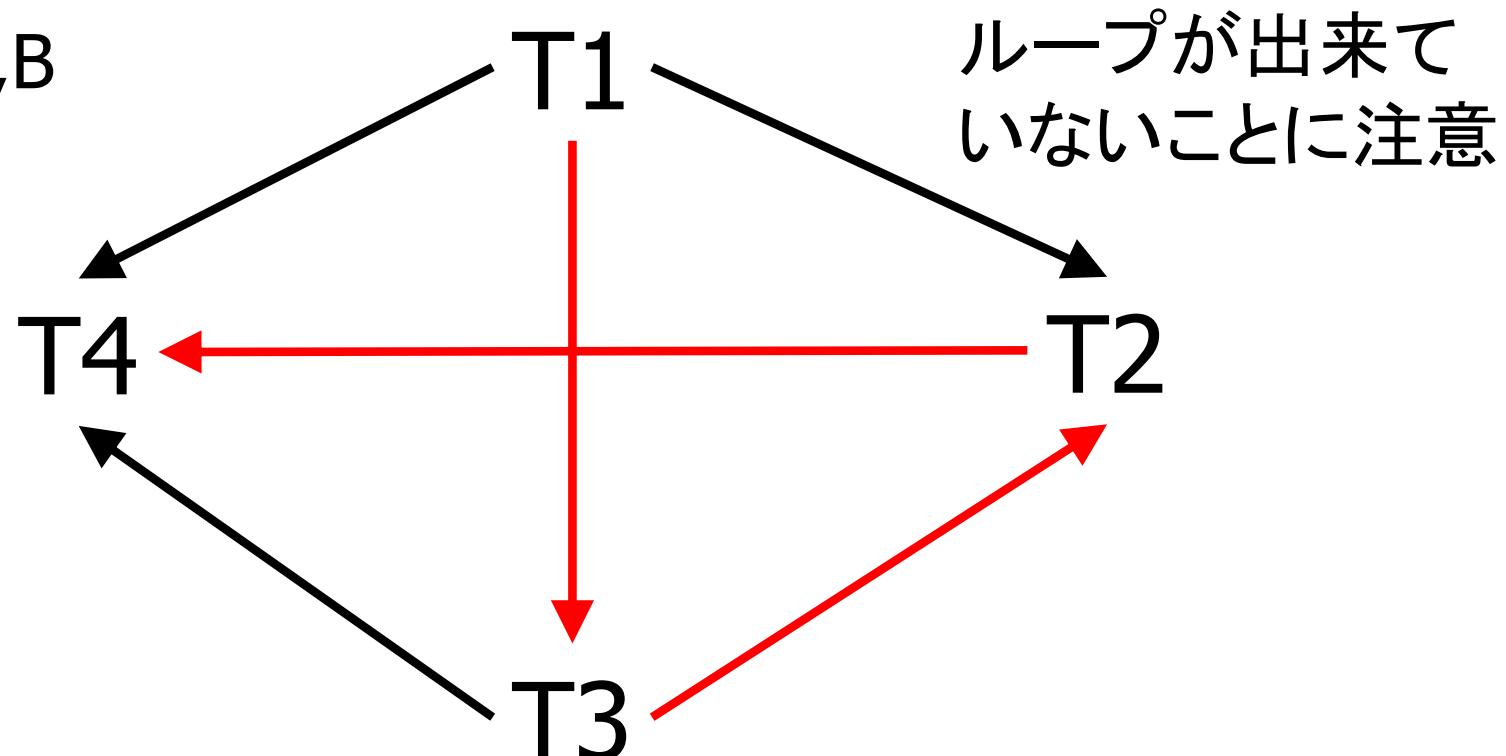
T3

T2



# 例題(相反グラフ)-最終形

レコードA,B  
の結果を  
合わせて



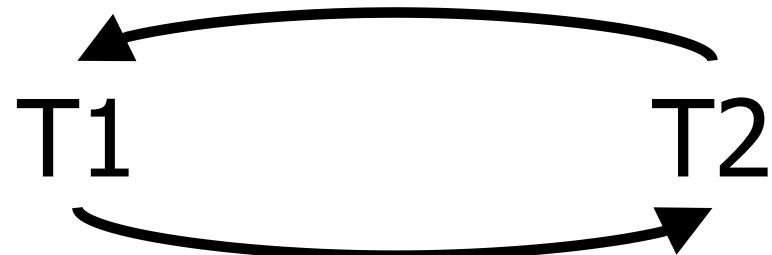
ループが出来て  
いないことに注意

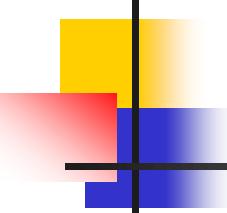
一筆書きでたどれる順序が等価な直列化スケジュール

$$T1 \rightarrow T3 \rightarrow T2 \rightarrow T4$$

# 直列化スケジュールと等価でない例

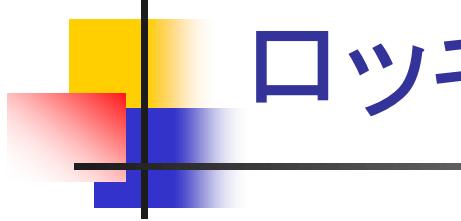
時刻	トランザクションT1	トランザクションT2
1	read(A)	
2		read(A)
3	write(A:=A-30)	
4		write(A:=A-20)
5	COMMIT	
6		COMMIT





# ロッキングプロトコル

- あらかじめどんなトランザクションがくるか分かっている場合(バッチ処理など)はこれまでの方法でよい
- どんなトランザクションがくるか分からない状況で直列化可能性を保証できる方法をかんがえる
  - ロッキングプロトコル
  - 時刻印順序プロトコル
  - 楽観的制御法



# ロックングプロトコル

## ■ 手順

- あるトランザクションがレコードAに対し読み書き(read/write)するときに、ロックする
  - 他のトランザクションがロックしている場合はロックできない
  - 読み書きが終了したらロックを解除する(アンロック)
- ただし、これだけでは直列化可能性は保証できない(直列化不可能なスケジュールがありうる)

# 直列化可能性が保証されないスケジュール(ロックは使っている)

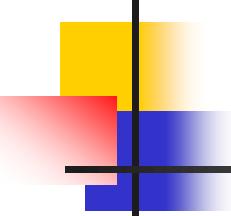
時刻	T1	T2
1	lock(A)	
2	read(A)	
3	unlock(A)	
4		lock(A)
5		read(A)
6		write(A:=A-20)
7		unlock(A)
8	lock(A)	
9	write(A:=A-30)	
10	unlock(A)	

T1の読み書きが一通り終わる前にT2の読み書きが実行されてしまって、ロックをかけている意味がない

# 2相ロックングプロトコル(2PL)

- ひとつのトランザクションにおいて、読み書きのために必要な**全てのロックが完了する以前にそれらがアンロックされない**、というロックのかけ方

時刻	T1	
1	lock(A)	
2	read(A)	⌚成長層 (ロックをかけていく)
3	lock(B)	
4	read(B)	
5	write(A)	
6	write(B)	
7	unlock(A)	⌚縮退層 (全て終わったらロックを外す)
8	unlock(B)	



# 2PLだと何故 直列化可能性が保証されるのか

- あるトランザクションがあるレコードにロックをかけている間は、同じレコードに対し他のトランザクションは読み書きできない
  - 異なるレコードに対してであれば読み書きの順序が変わっても直列化可能性には影響しない
  - 同じレコードに対しては、アンロックされた後でないとアクセスできないから先の例のように中途半端な状態での更新はない

# デッドロック

T1:  
begin  
    read(A)  
    read(B)  
    write(A:=A+B)  
end

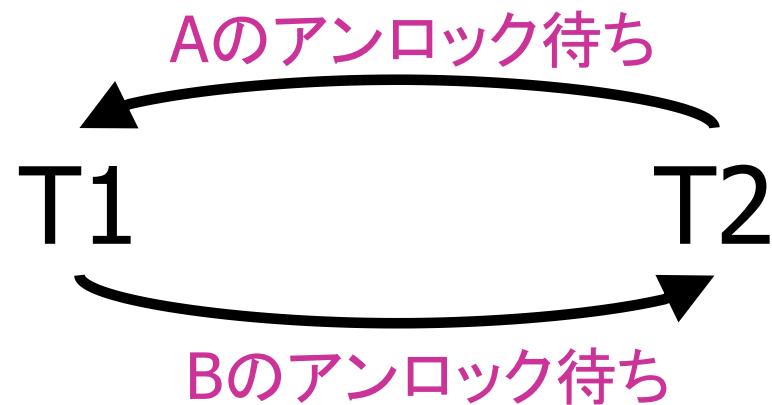
T2:  
begin  
    read(B)  
    read(A)  
    write(B:=B-A)  
end

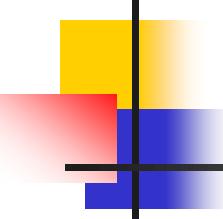
時刻	T1	T2
1	lock(A)	
2		lock(B)
3	read(A)	
4		read(B)
...	Bをreadしたいが×	Aをreadしたいが×

# [デッドロックの検出法]

## 待ちグラフ

- ノードは各トランザクション
- トランザクションTiが、別のトランザクションTjがロックしているレコードにアクセスするためのアンロック待ちの場合にTiからTjに有向リンクを張る
- ループが発生したらデッドロック





# デッドロックの発生防止

- 発生したら片方のトランザクションを「いけにえ」としてロールバックする
- 効率改善のため二種類のロックを導入する場合あり(デッドロックの発生が減る)
  - **共有ロック**: readするためのロック(他のトランザクションがreadすることは許すがwriteは許さない)
  - **専有ロック**: writeするためのロック(他のトランザクションのread/writeはゆるさない)