



# SQL

株式会社 AI Shift 三宅 悠太

# Contents

1. データベース
2. SQL I
3. トランザクション
4. データベース設計
5. インデックス
6. 実行計画
7. SQL II



1

# データベース



# データベースとは

---

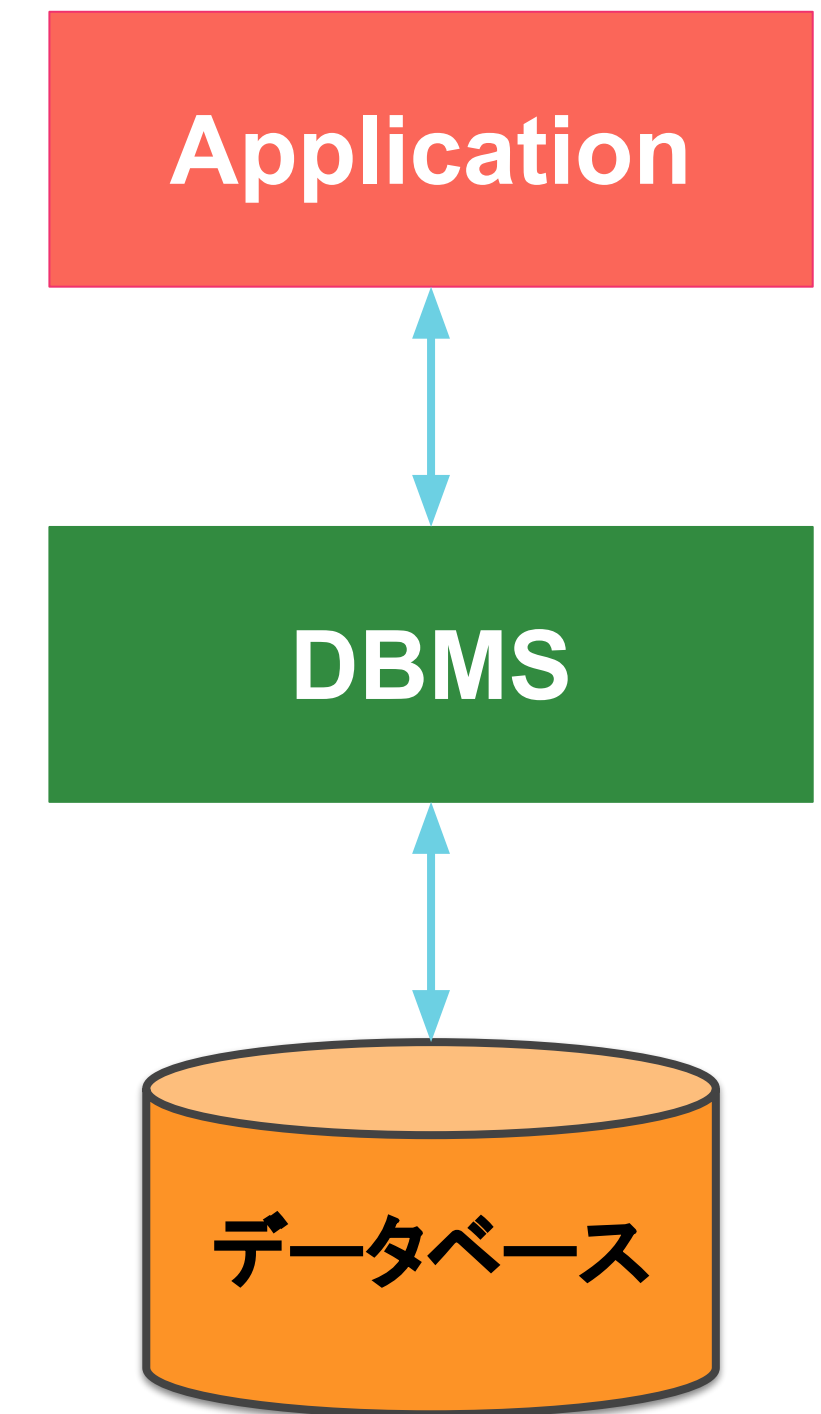
“A *database* is an organized collection of inter-related data that models some aspect of the real-world “ (CMU)

データベースとは、実世界のある側面をモデル化した、秩序だった、相互に関連したデータの集まり

# DBMS

---

- データベース管理システム(DBMS)は、データベースを管理するソフトウェア
  - 例: MySQL, Oracle Database, SQLite, MongoDB
- DBMSの目的は、アプリケーションが**簡単に**データベースにデータを保存したり、保存されたデータにアクセスしたりできるようにすること
- しかし、初期のDBMSは構築・保守する側も、利用する側も苦労した
  - 最大の原因は、**論理層と物理層の密結合**



# 論理層と物理層の密結合問題の例

- 4
- レコードの検索/追加/変更時は、都度**ファイル**を**パース**しなきゃ...
  - 物理ファイルの構成が変更されたら、こちらも変更しなきゃ...

- 3
- データベースの抽象化**が上手くできないため、**ファイル単位**でやり取り

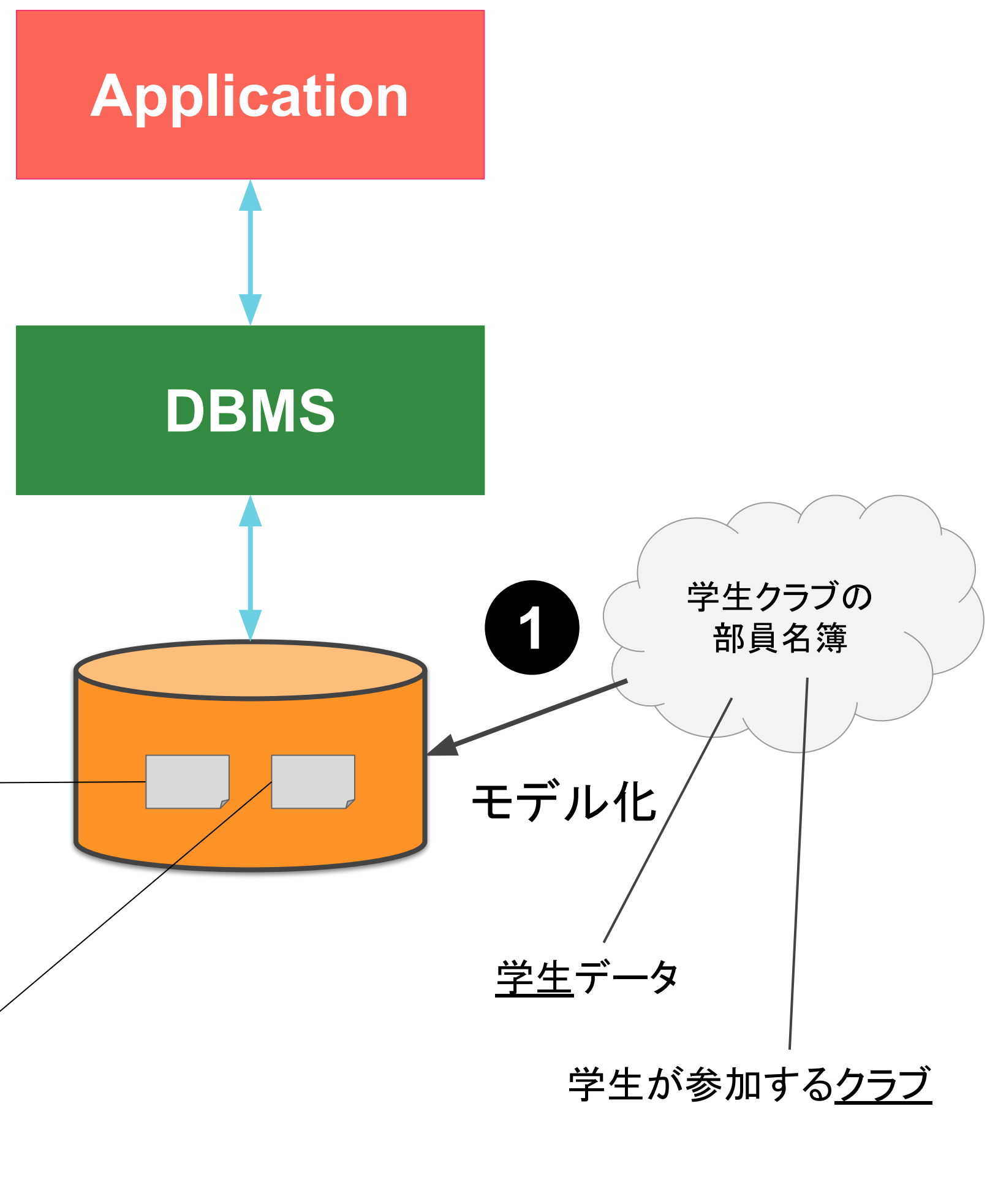
- 2
- 2つの**CSVファイル**から構成

学生(学生番号、氏名、年齢)

1, '佐藤', 20  
2, '山田', 19  
3, '金子', 18

クラブ(クラブ名、学生番号)

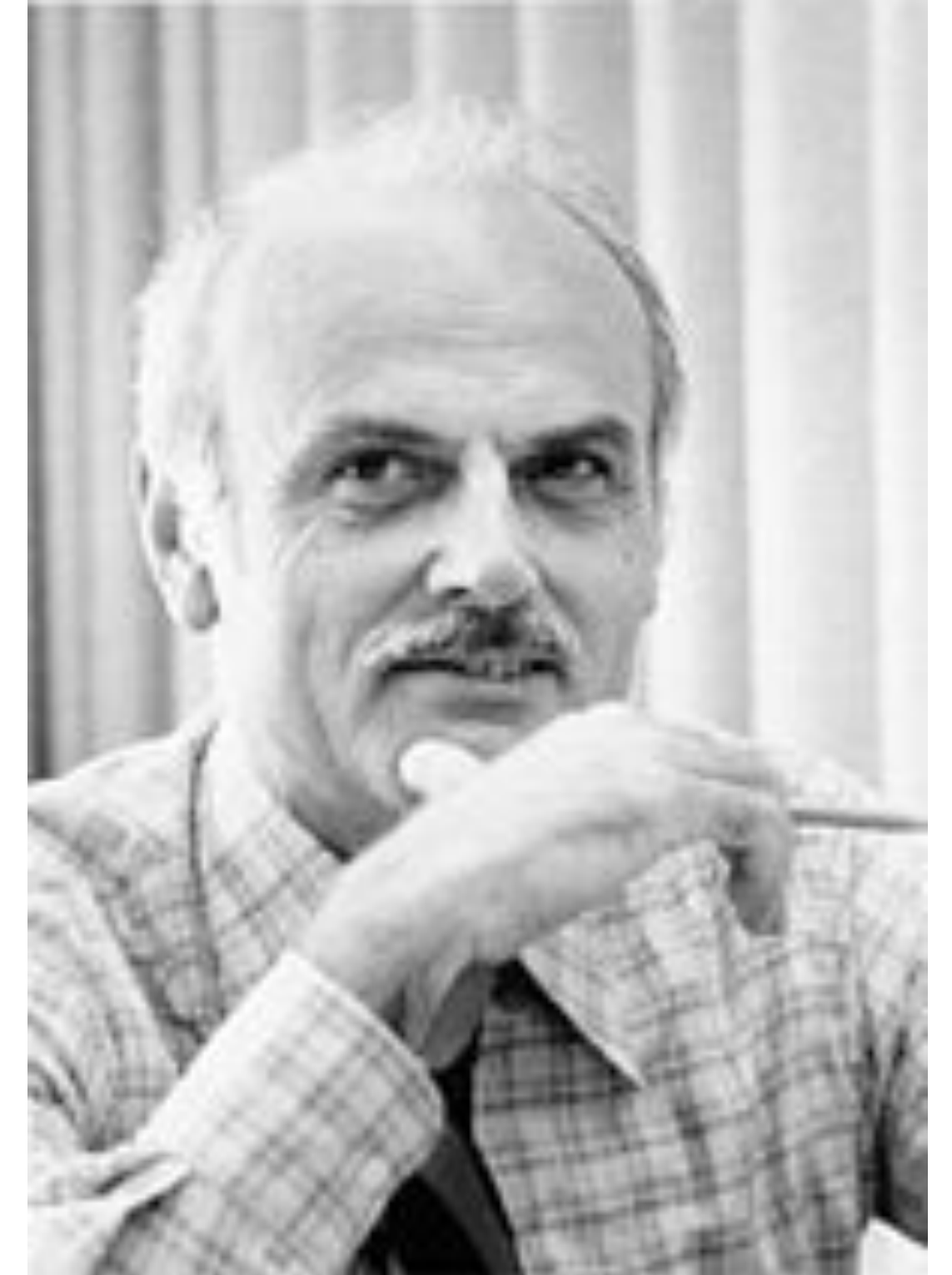
'写真部', 1  
'軽音部', 2  
'茶道部', 3



# リレーショナルモデル

---

- 1970年にエドガー・コッド博士が考案
- データベースを抽象化し、例に挙げたような問題を回避
  - シンプルなデータ表現・データ構造 → リレーション
  - 高水準な言語によるデータ操作 → SQL
  - **データの独立性**
- 彼の論文発表以降、関係モデルに準拠するDBMSの開発・商業化が進み、デファクトになっていく
  - 伝統的に、DBMSといえば関係モデル準拠を指す（正確にはRDBMS）

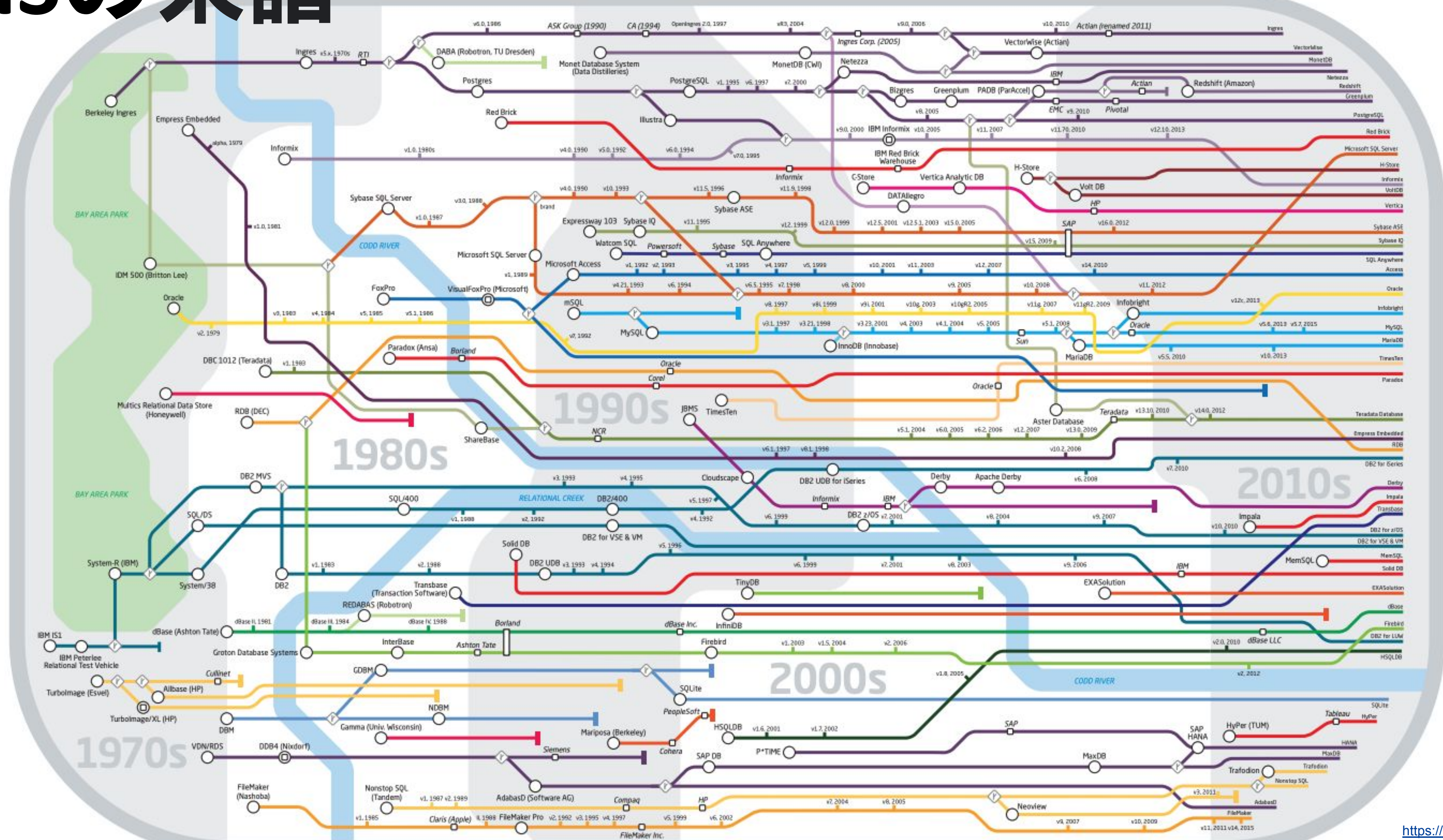


エドガー・コッド博士



# DBMSの系譜

## Genealogy of Relational Database Management Systems



<https://hpi.de/naumann/projects/rdbms-genealogy.html>



# DBMSの系譜

## Genealogy of Relational Database Management Systems





# データモデル

- データモデル

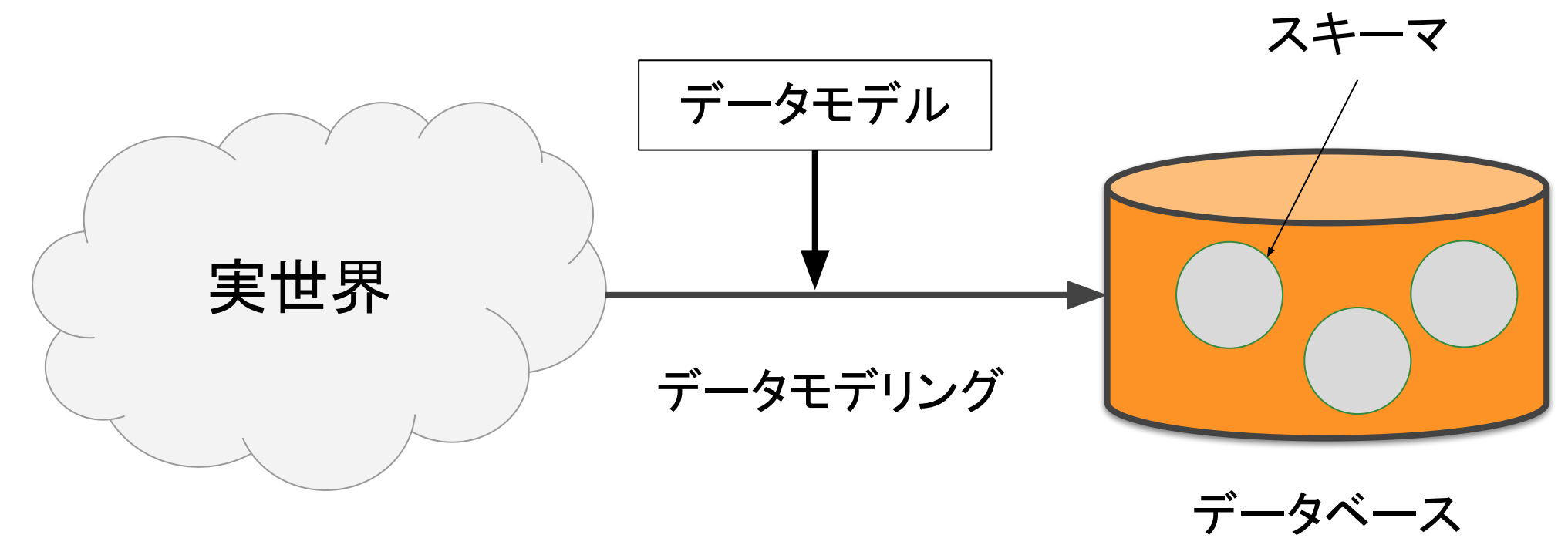
- 対象の実世界をデータで表現するための概念の集まり

- スキーマ

- データモデルで記述された特定のデータの集まり

- 代表的なデータモデルの種類

- ネットワークモデル
- 階層モデル
- リレーショナル
- ドキュメント
- キーバリュー
- グラフベース
- カラムファミリー



データベースをデータモデルで  
モデル化した場合の関係性

# データモデル

---

- データモデル

- 対象の実世界をデータで表現するための概念の集まり

- スキーマ

- データモデルで記述された特定のデータの集まり

- 代表的なデータモデルの種類

- ネットワークモデル

- 階層モデル

準拠 -> 第1世代DB

- リレーショナル

- ドキュメント

- キーバリュー

- グラフベース

- カラムファミリー

# データモデル

---

- データモデル

- 対象の実世界をデータで表現するための概念の集まり

- スキーマ

- データモデルで記述された特定のデータの集まり

- 代表的なデータモデルの種類

- ネットワークモデル

- 階層モデル

- リレーショナル

**準拠 -> 第2世代DB この研修の中心**

- ドキュメント

- キーバリュー

- グラフベース

- カラムファミリー



# データモデル

---

- データモデル

- 対象の実世界をデータで表現するための概念の集まり

- スキーマ

- データモデルで記述された特定のデータの集まり

- 代表的なデータモデルの種類

- ネットワークモデル
- 階層モデル
- リレーショナル
- ドキュメント
- キーバリュー
- グラフベース
- カラムファミリー

-> NoSQL (“Not Only SQL”)

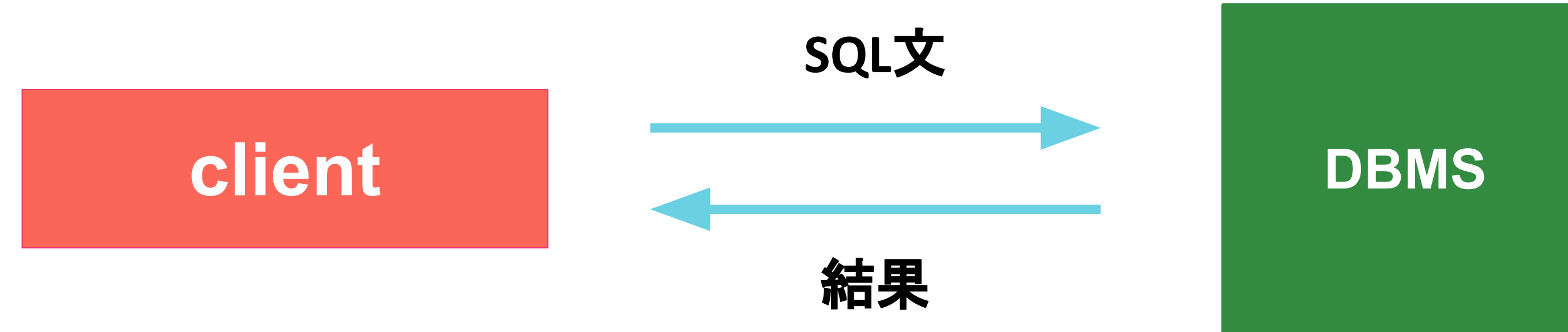
2

SQL I



# SQL

---



DBMSにおいて、データの操作や定義を行うための高水準な言語

# SQLの歴史

- 1970年、IBMから関係モデルの論文が公表される
- 1970年代、IBMが「SEQUEL」を開発
  - System R の操作言語
- 1980年代、DBMSの商業化が進む
- 1986-87年、ANSI/ISOによる言語の標準化
  - Structured Query Language (SQL)
  - 数年おきに更新
  - 最新は2016

標準規格一覧（一部抜粋）

標準規格	主要な追加機能
SQL:1999	正規表現、トリガー、ユーザー定義関数
SQL:2003	XML、ウィンドウ関数、シーケンス
SQL:2008	TRUNCATE
SQL:2011	テンポラルテーブル
SQL:2016	JSON、ポリモーフィックテーブル



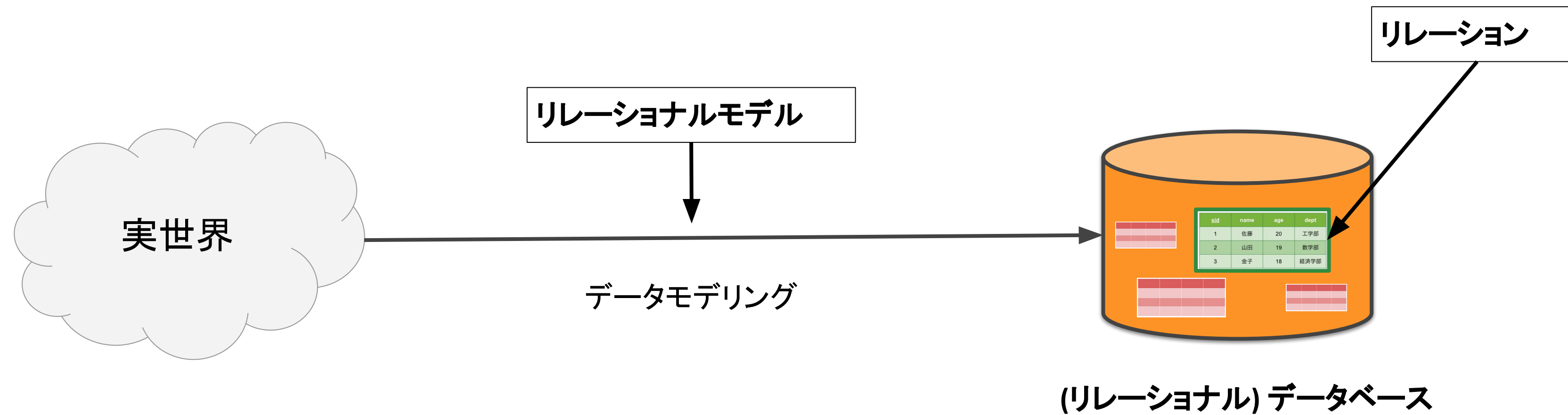
# SQLの特徴

---

- 実装の多様性
  - SQLはDBMS向けの標準規格ではあるが。。
  - 準拠の程度や実装はベンダー依存
  - **利用するDBMSの製品仕様を読む**
- 汎用的・機能豊富
  - 長年の機能拡張により
- **宣言型**
  - SQLユーザーはデータをどうやって得るかではなく、**何が欲しいか**を記述すればOK
    - データ独立万歳

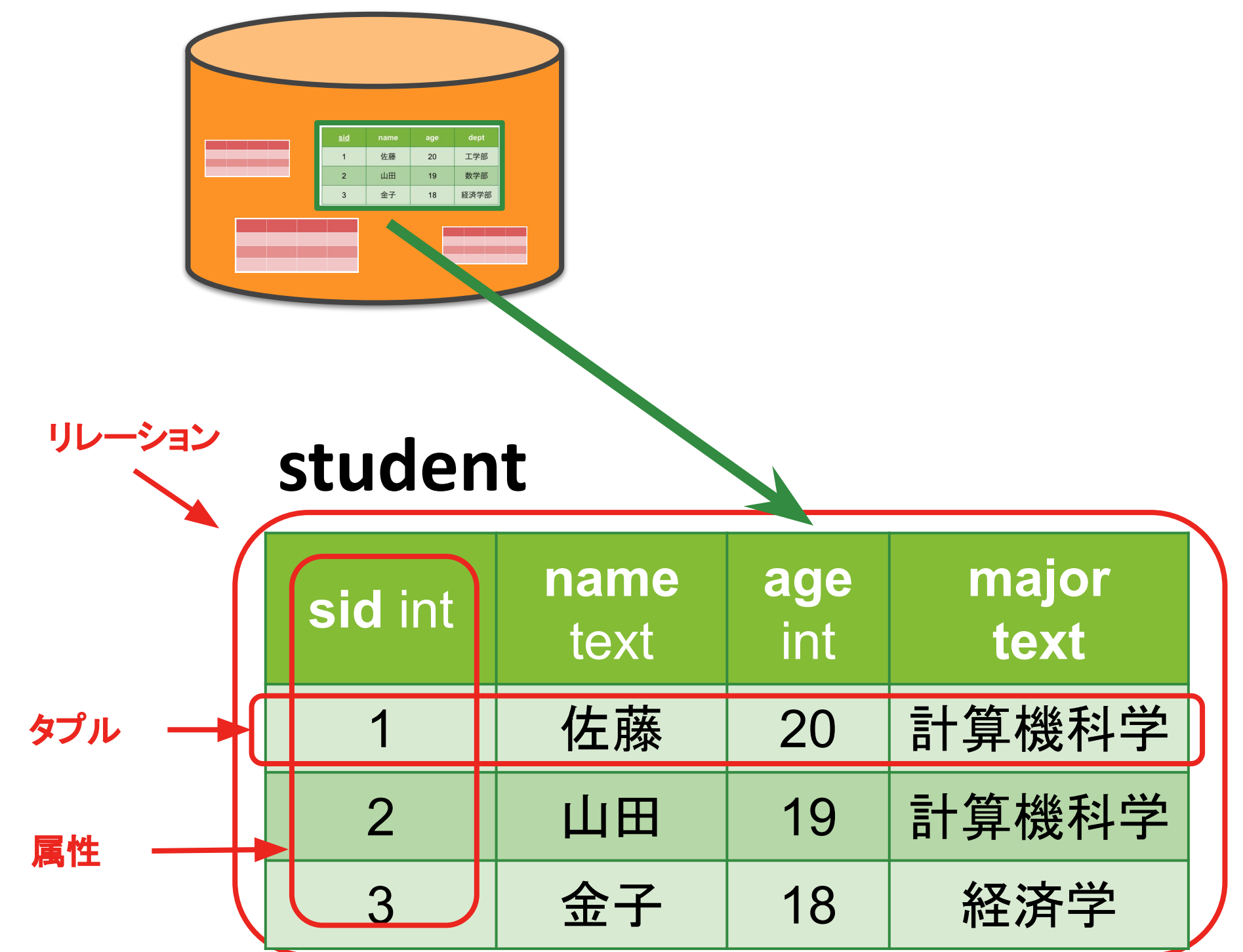
# SQLが扱うデータの単位は何か

- DBMSはリレーショナルモデル準拠
- リレーションがデータの基本単位



# リレーション

- リレーション(テーブル)
  - スキーマ: データの構造や制約の定義(メタデータ)
  - インスタンス: スキーマを満たす実際のデータ集合
- 属性(列、フィールド)
  - ドメイン: 有効な値の範囲
- タプル(行、レコード)
  - 属性値の集まり、何らかの”事実”を表す
  - インスタンス=タプルの集合=”似たような事実”の集まり



例: 学生リレーションを二次元表として可視化した図

# SQLテーブルとの主な違い

---

- SQLテーブルの特徴
  - **行の重複** OK (“事実”が複数保存できてしまう...)
  - 属性値が不明な状態 (**NULL**) OK
  - 集合ではなくタプルの多重集合



# 用語の整理

- 文脈によって使い分けが必要となときがくるかもしれないので、対応表を頭にいれておくとも良いかも

用語の対応表

リレーショナルモデルの 公式用語	非形式的な日常用語	
	SQL	データの内部表現
関係(リレーション)	表(テーブル)	ファイル
組(タプル)	行(ロウ)	レコード
属性(アトリビュート)	列(カラム)	フィールド

# SQLのサブ言語

---

- データ定義言語 (DDL)
  - **CREATE**, ALTER, DROP, ...
- データ操作言語 (DML)
  - **SELECT**, INSERT, UPDATE, DELETE, ...
- ...

# CREATE

---

- CREATE文は、テーブルの構造(スキーマ)を定義する時に利用

**student**

sid	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学

**club**

cid	name	budget
1	写真部	300
2	軽音部	450000
3	茶道部	9000000

**club\_member**

cid	sid	joined_at
1	1	2021/4/15
2	1	2021/4/15
3	3	2021/5/1

# CREATE

---

```
CREATE TABLE student (  
  sid INTEGER,  
  name CHAR(20),  
  age INTEGER,  
  major CHAR(20);
```

sid	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学

- 基本的に、テーブル名と(属性名、属性のデータ型)のリストを書くだけ



# CREATE: 主キー

---

```
CREATE TABLE student (  
  sid INTEGER,  
  name CHAR(20),  
  age INTEGER,  
  major CHAR(20),  
  PRIMARY KEY (sid));
```

<u>sid</u>	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学

## 主キー (Primary Key)

- テーブル内の1行を一意に識別するための”検索キー”
- 主キー列では、値の重複が禁止される(主キー制約)
- 複数の列からも構成できる(複合主キー)
  - 例: PRIMARY KEY(name, age)

# CREATE

```
CREATE TABLE student (  
  sid INTEGER,  
  name CHAR(20),  
  age INTEGER,  
  major CHAR(20),  
  PRIMARY KEY (sid));
```

```
CREATE TABLE club (  
  cid INTEGER,  
  name CHAR(20),  
  budget INTEGER,  
  PRIMARY KEY (cid));
```

```
CREATE TABLE club_member (  
  sid INTEGER,  
  cid INTEGER,  
  joined_at DATE,  
  PRIMARY KEY (sid, cid));
```

<u>sid</u>	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学

<u>cid</u>	name	budget
1	写真部	300
2	軽音部	450,000
3	茶道部	9,000,000

<u>sid</u>	<u>cid</u>	joined_at
1	1	2021/4/15
1	2	2021/4/15
2	3	2021/5/1

Q. なぜ PKはsidだけとか、cidだけではだめだったのか

# CREATE: 外部キー

子テーブル

```
CREATE TABLE club_member (  
  sid INTEGER,  
  cid INTEGER,  
  joined_at DATE,  
  PRIMARY KEY (sid, cid),  
  FOREIGN KEY (sid) REFERENCES student(sid));
```

外部キー →

<u>sid</u>	<u>cid</u>	joined_at
1	1	2021/4/15
1	2	2021/4/15
2	3	2021/5/1

参照  
(references)

親テーブル

<u>sid</u>	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学

<u>cid</u>	name	budget
1	写真部	300
2	軽音部	450000
3	茶道部	9000000

## 外部キー (Foreign Key)

- 特定のテーブルの列を参照するためのキー (“ポインター”)
- 2つのテーブルの間に関連 (親子関連) をもたらす
- 主キーと違い複数設定できる

# CREATE: 外部キー

子テーブル

```
CREATE TABLE club_member (  
  sid INTEGER,  
  cid INTEGER,  
  joined_at DATE,  
  PRIMARY KEY (sid, cid),  
  FOREIGN KEY (sid) REFERENCES student(sid),  
  FOREIGN KEY (cid) REFERENCES club(cid));
```

親テーブル

<u>sid</u>	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学

親テーブル

<u>cid</u>	name	budget
1	写真部	300
2	軽音部	450000
3	茶道部	9000000

## 外部キー (Foreign Key)

- 特定のテーブルの列を参照するためのキー (“ポインター”)
- 2つのテーブルの間に関連(親子関連)をもたらす
- 主キーと違い複数設定できる

<u>sid</u>	<u>cid</u>	joined_at
1	1	2021/4/15
1	2	2021/4/15
2	3	2021/5/1

参照  
(references)

# CREATE: 外部キー制約

## 外部キー制約(参照整合性制約)

- 1. 子テーブルの外部キーの値は親テーブルの主キーに存在する値でないといけない
- 2. 親テーブルの参照されるキーに対してのUPDATE/DELETEを禁止

子テーブル

<u>sid</u>	<u>cid</u>	joined_at
1	1	2021/4/15
1	2	2021/4/15
2	3	2021/5/1

1. INSERT (4,...)



親テーブル

<u>sid</u>	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学

2. DELETE sid=1



<u>cid</u>	name	budget
1	写真部	300
2	軽音部	450000
3	茶道部	9000000

# MySQLの参照アクション

- 参照アクションは、親テーブルの参照されるキーに対してUPDATE/DELETEした場合の子テーブルの外部キーに与える影響を指す
- UPDATE/DELETE、それぞれ4つのアクションから選べる

記述方法

FOREIGN KEY (sid)  
REFERENCES student  
ON UPDATE RESTRICT  
ON DELETE RESTRICT

アクション	ON UPDATE	ON DELETE
RESTRICT	親テーブルへのUPDATEは拒否される (デフォルト)	親テーブルへのDELETEは拒否される (デフォルト)
NO ACTION	RESTRICTと同様	RESTRICTと同様
CASCADE	親テーブルへのUPDATEが実施され、子 テーブルの一致する行を自動的に UPDATE	親テーブルへのDELETEが実施され、子テーブルの 一致する行を自動的にDELETE
SET NULL	親テーブルへのUPDATEが実施され、子 テーブルの外部キーを NULLに設定	親テーブルへのDELETEが実施され、子テーブルの 外部キーを NULLに設定



# PKとFKは設定しておこう

---

- 基本的に主キーと外部キーは設定しよう！
- 主キーのメリット
  - 行の重複が防げる
  - クエリで個別の行を参照できる
  - 外部キー参照をサポート
- 外部キーのメリット
  - 参照整合性が保たれる
  - アプリには難しいCASCADEアクション
- 上記をApp側でやるのは手間なので、DBに任せよう

# ここまでの要点

---

- DBMSはリレーショナルモデルがベース
  - **構造**: テーブル
  - **操作**: SQL
  - **整合性**: 主キー制約、外部キー制約
- SQL
  - データ定義言語
    - **CREATE TABLE**でスキーマを定義
      - テーブル名、属性名、属性のデータ型
      - 主キー制約、外部キー制約（基本は設定しよう！）
  - データ操作言語
    - 次は**SELECT**

# SELECT文

---

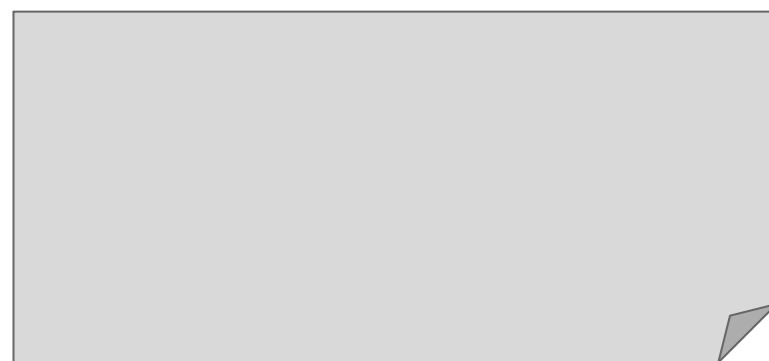
- SELECT文はデータを取得するために利用する
- テーブルを引数にテーブルを返す関数みたいなもの
  - クエリ結果は0行以上のテーブル



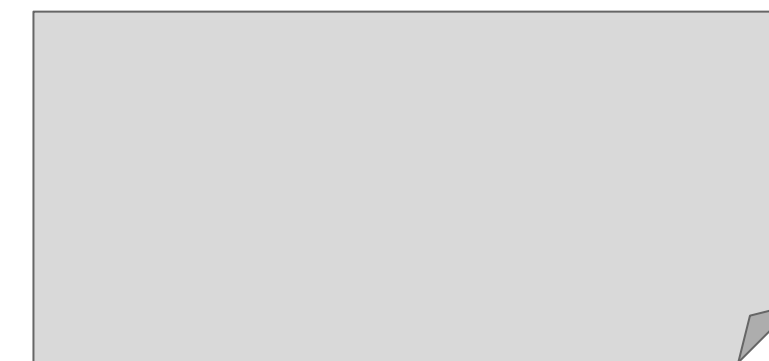
SELECT文







Shell  
Command



# 単純クエリ

---

単純クエリの構文

```
SELECT <式のリスト>  
FROM <テーブル名>  
[WHERE <述語>];
```

- SELECT句
  - クエリ結果として出力する行を形成するための<式のリスト>
  - 式に書けるのは列名、計算式、定数など(例:「age」,「age + 2」,「1」)
- WHERE句
  - 検索対象の行を絞り込むための<述語> (検索条件)
  - 述語＝真偽値を返す関数 (例: age = 20, age >= 19 AND age <= 22 ...)

# SELECT DISTINCT

例: 20歳未満の学生一覧 (氏名の重複なし)

```
SELECT DISTINCT name
FROM student
WHERE age < 20;
```

- SELECT DISTINCTはクエリ結果(SELECT句の処理結果)から重複行を除外する修飾子
  - 出力を制御する役割

sid	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学
4	金子	18	経済学



sid	name
2	山田
3	金子

# 出力制御: ORDER BY

---

- クエリ結果を任意の列で昇順にソート

例: 氏名の辞書順で並び替え

```
SELECT sid, name
FROM student
WHERE age < 20
ORDER BY name;
```

- デフォルトでは昇順 (ASC) だが、降順 (DESC) にもできる
- 複数のASCとDESCのミックス

例: 年齢順、名前順で並び替え

```
SELECT sid, nam
FROM student
WHERE age < 20
ORDER BY age DESC, name ASC;
```



# 出力制御: LIMIT

---

- クエリ結果を最初の <整数> 行に絞る
- ORDER BY句と一緒に使おう
  - クエリ結果の順序が非決定的になるのを避ける

例: 学生一覧(名前順)の上位5つだけ返す

```
SELECT sid, name  
FROM student  
WHERE age < 20  
ORDER BY name;  
LIMIT 5;
```

# 集約関数

- SELECT句に書けるもう一つのもの
- 複数行の列の値を集計し1つの値を返す関数
- 関数名(<値式>)の形
- AVG, SUM, COUNT, MAX, MINなど

例: 計算機科学を専攻する学生の平均年齢

```
SELECT AVG(age)
FROM student
WHERE major = '計算機科学';
```

sid	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学
..	..	..	..



集約＝複数行を1行にまとめる

AVG(age)
21.9000

クエリ結果

# 集約関数: COUNT

---

Q. 計算機科学を専攻する学生の数を知りたい

```
SELECT COUNT(*)  
FROM student  
WHERE major = '計算機科学';
```



COUNT(*)
30

```
SELECT COUNT(age)  
FROM student  
WHERE major = '計算機科学';
```



COUNT(age)
30

```
SELECT COUNT(1)  
FROM student  
WHERE major = '計算機科学';
```



COUNT(1)
30

# 集約関数: COUNT

---

Q. 計算機科学を専攻する学生の数を知りたい

```
SELECT COUNT(*)  
FROM student  
WHERE major = '計算機科学';
```



COUNT(*)
30

```
SELECT COUNT(age)  
FROM student  
WHERE major = '計算機科学';
```



COUNT(age)
30

```
SELECT COUNT(1)  
FROM student  
WHERE major = '計算機科学';
```



COUNT(1)
30

- COUNT(\*)とCOUNT(定数)は全行数
- COUNT(列名)はNULLを除外した行数

# 複数の集約関数

---

```
SELECT AVG(age), COUNT(*)  
FROM student  
WHERE major = '計算機科学';
```



AVG(age)	COUNT(*)
21.9000	30

# DISTINCT 集約関数

---

```
SELECT COUNT(DISTINCT name)
FROM student;
```

- 重複値を排除した上で集計できる
- COUNT、SUM、AVGがDISTINCTをサポート



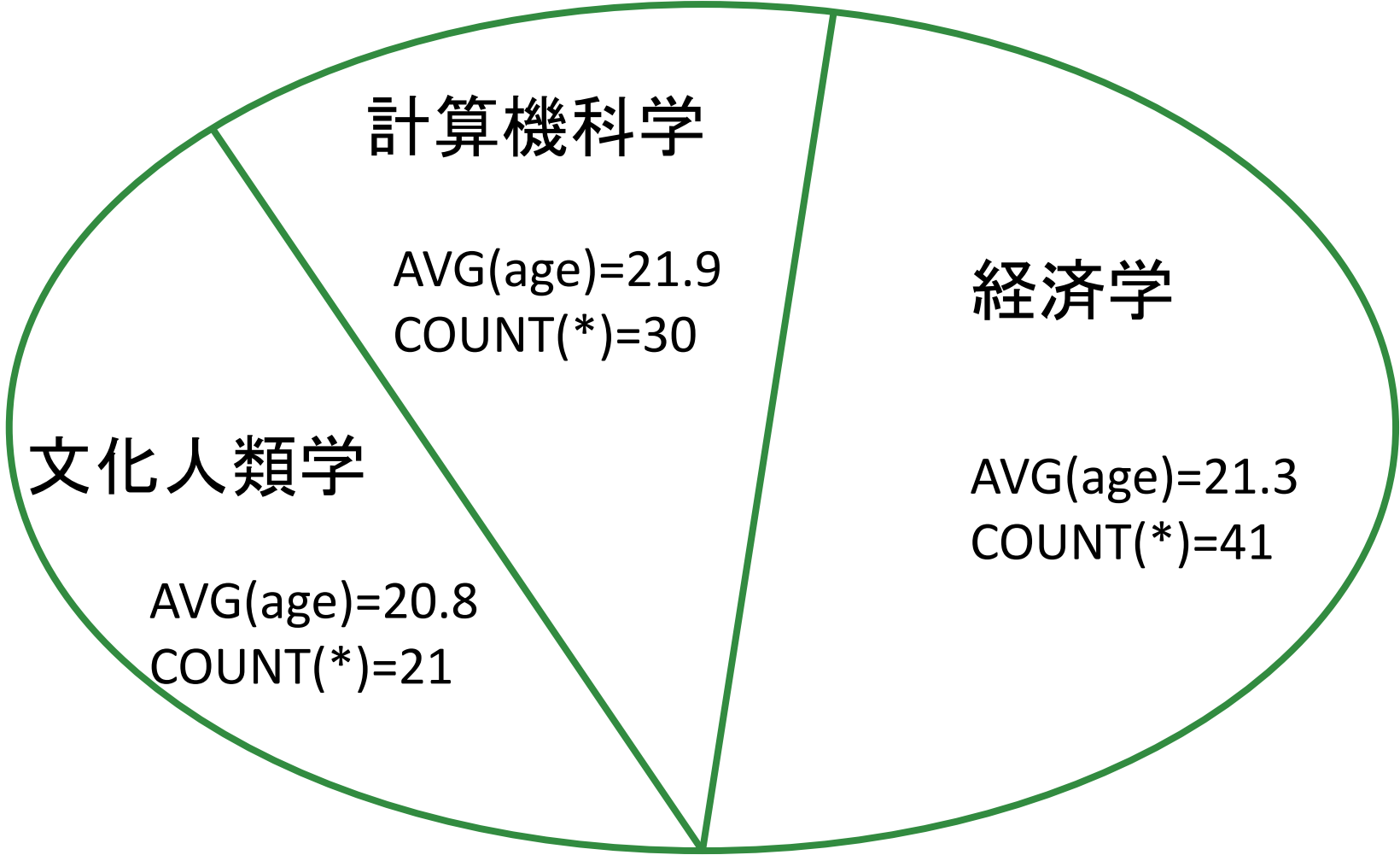
# GROUP BY

- グループ毎に集計を行いたい時に利用
- GROUP BY <列のリスト(集約キー)>
  - テーブルを集約キーでグループ分けし、グループ単位で集計  
(GROUP BY=**グルーピング+集約**)
  - クエリ結果の行数 = ユニークなグループの数

例: 学生の専攻毎に集計

```
SELECT major, AVG(age), COUNT(*)
FROM student
GROUP BY major;
```

集約キーで集合をカットしたイメージ



major	AVG(age)	COUNT(*)
計算機科学	21.9	30
経済学	21.3	41
文化人類学	20.8	21

# GROUP BY

---

- GROUP BY句の使用時にSELECT句に書けるもの
  - GROUP BY句で指定した集約キー
  - 集約関数
  - 定数

```
SELECT major, AVG(age), COUNT(*), age
FROM student
GROUP BY major, age;
```

# 個人と集団の属性の違い

---

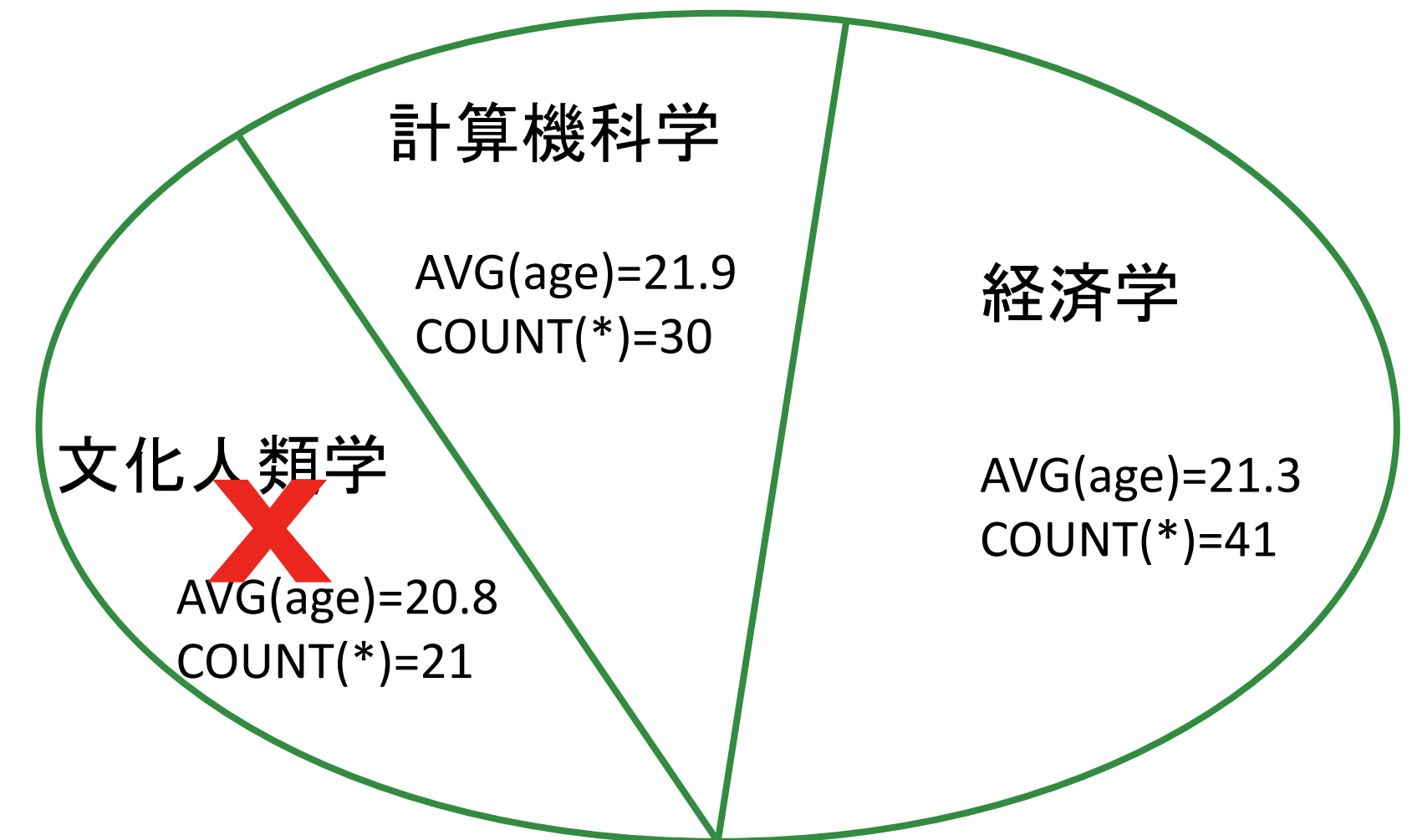
- GROUP BYを使うということは
  - 個人の属性(列)は気にしない
    - 例:Aさんの年齢
  - 集団の属性だけ気にする
    - i.e., 統計的な属性
    - 例:チームXの平均年齢
- GROUP BY以降は、頭を**集合指向**に切り替えよう！

# HAVING

- 検索対象のグループを絞り込みたい時に利用
- 条件に含められるもの
  - GROUP BY句で指定した集約キー、集約関数、
- GROUP BY句と一緒にしか使えない

例: 集計対象の専攻を学生数30以上のものに絞る

```
SELECT major, AVG(age), COUNT(*)  
FROM student  
GROUP BY major  
HAVING COUNT(*) >= 30;
```



# 全部使ってみる

---

```
SELECT major, AVG(DISTINCT age) AS avg_age
FROM student
WHERE age <= 20
GROUP BY major
HAVING COUNT(*) >= 10
ORDER BY avg_age DESC
LIMIT 1;
```

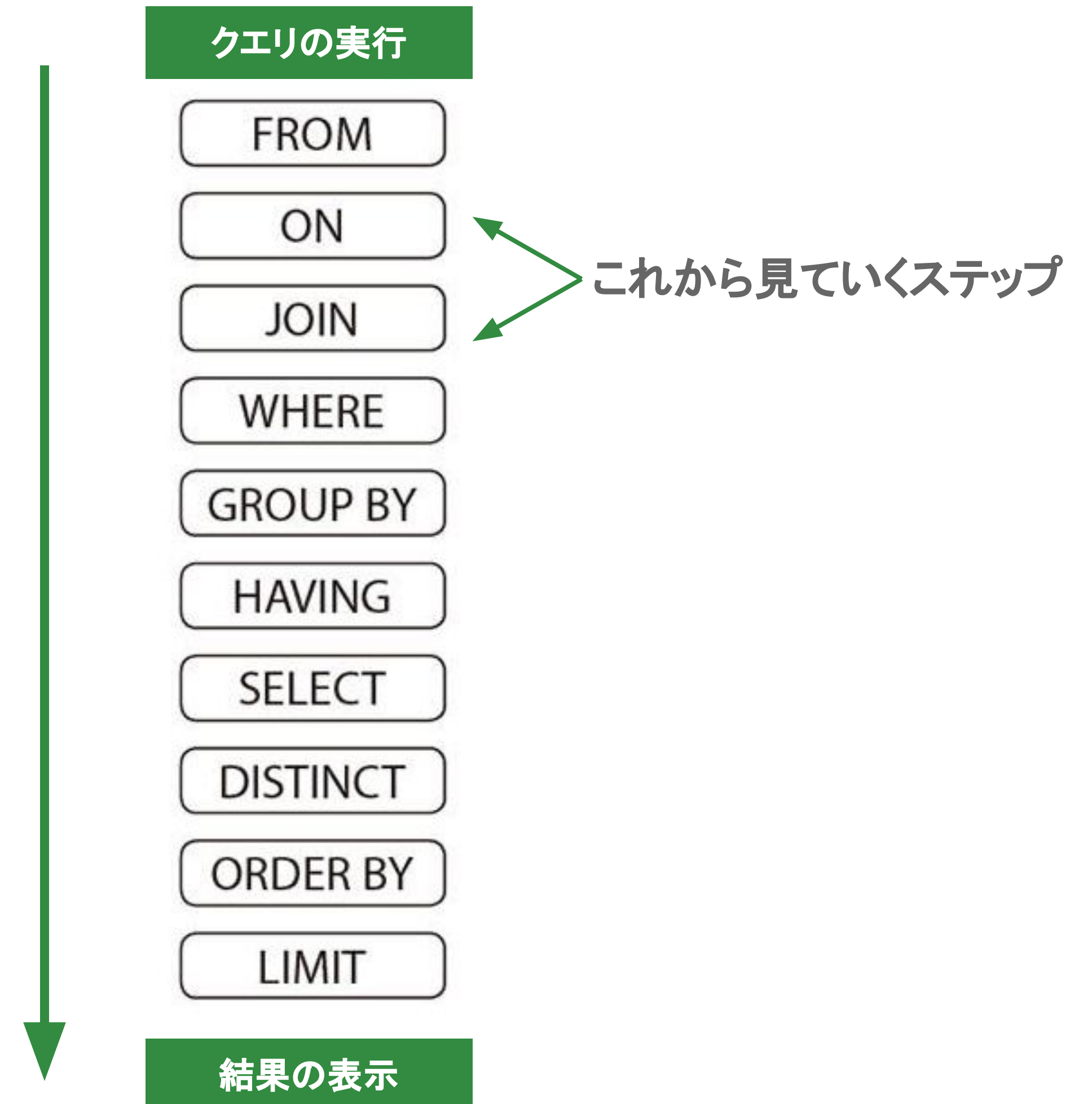
Q. このクエリが実行されると、どう  
いった処理がされるか



# SELECT文の評価順序

```
SELECT major, AVG(DISTINCT age) AS avg_age
FROM student
WHERE age <= 20
GROUP BY major
HAVING COUNT(*) >= 10
ORDER BY avg_age DESC
LIMIT 1;
```

- SELECT文には論理的な評価順序(実行順序)がある
  - 実際の実行順序はDBMSが決定
  - もちろん、製品によって評価順序は違う



# 結合 (JOIN)

---

## 結合クエリの構文

```
SELECT <値式のリスト>  
FROM <テーブル名のリスト>  
WHERE ...;
```

- 結合は1つのクエリで複数テーブルのデータを扱いたい時に利用
- 結合の種類は3つ
  - クロス結合 (CROSS JOIN)
  - 内部結合 (INNER JOIN)
  - 外部結合 (OUTER JOIN)

# CROSS JOIN

```
SELECT *  
FROM student, club_member;
```

<u>sid</u>	name	age	major
1	佐藤	20	計算機科学
2	山田	19	計算機科学
3	金子	18	経済学

<u>sid</u>	<u>cid</u>	joined_at
1	1	2021/4/15
1	2	2021/4/15
2	3	2021/5/1



CROSS JOIN (クロス結合)

- クロス結合の結果は、全てのテーブルの行の全ての組み合わせを網羅したもの
  - 例: 各学生に対して、部活メンバーテーブルの3つの行との組み合わせが考えられるので、合計9行

## クエリ結果

<u>sid</u>	name	age	major	<u>sid</u>	<u>cid</u>	joined_at
1	佐藤	20	計算機科学	1	1	2021/4/15
1	佐藤	20	計算機科学	1	2	2021/4/15
1	佐藤	20	計算機科学	2	3	2021/5/1
2	山田	19	計算機科学	1	1	2021/4/15
2	山田	19	計算機科学	1	2	2021/4/15
2	山田	19	計算機科学	2	3	2021/5/1
3	金子	18	経済学	1	1	2021/4/15
3	金子	18	経済学	1	2	2021/4/15
3	金子	18	経済学	2	3	2021/5/1

# INNER JOIN

```
SELECT s.*, cm.cid  
FROM student s, club_member cm  
WHERE s.sid = cm.sid;
```

- 内部結合はクロス結合の部分集合
  - “内部”とはそういう意味
- 内部結合の結果は、クロス結合から結合条件を満たす行だけに絞ったものになる
- 結合条件の分類
  - 等値結合 (=)
  - 非等値結合 (>, <, <>, !=, ...)

条件マッチ

<u>sid</u>	name	age	major	<u>sid</u>	<u>cid</u>	joined_at
1	佐藤	20	計算機科学	1	1	2021/4/15
1	佐藤	20	計算機科学	1	2	2021/4/15
1	佐藤	20	計算機科学	2	3	2021/5/1
2	山田	19	計算機科学	1	1	2021/4/15
2	山田	19	計算機科学	1	2	2021/4/15
2	山田	19	計算機科学	2	3	2021/5/1
3	金子	18	経済学	1	1	2021/4/15
3	金子	18	経済学	1	2	2021/4/15
3	金子	18	経済学	2	3	2021/5/1

クエリ結果

<u>sid</u>	name	age	major	<u>cid</u>
1	佐藤	20	計算機科学	1
1	佐藤	20	計算機科学	2
2	山田	19	計算機科学	3

# JOIN 構文

---

## クロス結合

```
1. SELECT *  
   FROM student, club_member;
```

```
2. SELECT *  
   FROM student CROSS JOIN club_member;
```

## 内部結合

```
1. SELECT s.*, cm.cid  
   FROM student s, club_member cm  
   WHERE s.sid = cm.sid;
```

```
2. SELECT s.*, cm.cid  
   FROM student s INNER JOIN club_member cm  
   ON s.sid = cm.sid;
```

# LEFT OUTER JOIN

```
SELECT s.*, cm.cid  
FROM student s LEFT OUTER JOIN club_member cm  
ON s.sid = cm.sid;
```

- 内部結合と同様に、結合条件を満たす行を全て返す
- さらに、マッチしなかった左側のテーブルの行も残す
  - cidはNULLで埋める

<u>sid</u>	name	age	major	<u>sid</u>	<u>cid</u>	joined_at
1	佐藤	20	計算機科学	1	1	2021/4/15
1	佐藤	20	計算機科学	1	2	2021/4/15
1	佐藤	20	計算機科学	2	3	2021/5/1
2	山田	19	計算機科学	1	1	2021/4/15
2	山田	19	計算機科学	1	2	2021/4/15
2	山田	19	計算機科学	2	3	2021/5/1
3	金子	18	経済学	1	1	2021/4/15
3	金子	18	経済学	1	2	2021/4/15
3	金子	18	経済学	2	3	2021/5/1

条件に合うものなし

## クエリ結果

<u>sid</u>	name	age	major	<u>cid</u>
1	佐藤	20	計算機科学	1
1	佐藤	20	計算機科学	2
2	山田	19	計算機科学	3
3	金子	18	経済学	NULL



# OUTER JOIN

---

- 外部結合の種類
  - 左外部結合 (LEFT OUTER JOIN)
  - 右外部結合 (RIGHT OUTER JOIN)
  - 完全外部結合 (FULL OUTER JOIN)
- 右と左の違いは、どちらをマスタに選ぶか
  - マスタのデータ(行)はすべて残る
- 両方ともマスタなのが、FULL OUTER JOIN

# JOINの要点

- 結合の種類は3つ
  - クロス結合 (CROSS JOIN)
  - 内部結合 (INNER JOIN)
  - 外部結合 (OUTER JOIN)
    - 左外部結合 (LEFT OUTER JOIN)
    - 右外部結合 (RIGHT OUTER JOIN)
    - 完全外部結合 (FULL OUTER JOIN)
- 補足
  - クロス結合を実務で使うことはほぼない
  - 自己結合もできる(同じテーブルに別名をつけて結合するイメージ)
- 結合したからと言って、それ以降の処理の流れは変わらない

## クエリの実行

FROM

ON

JOIN

WHERE

GROUP BY

HAVING

SELECT

DISTINCT

ORDER BY

LIMIT

## 結果の表示

# クエリを書いてみよう



Q1. 31アイスクリームのフレーバーの組み合わせ一覧がほしい  
("ダブルサイズ"の選択肢をください)

- ただし、同じフレーバーはなし
- 同じ組み合わせは一覧にいれないでほしい
- カロリーの合計もついでにほしい

name	kind	calorie
ロッキーロード	THE 31	162
ナッツトウユー	ELEGANT	168
チョップドチョコレート	CHOCOLATE	175
...	...	...



Q2. 上の一覧から最適な組み合わせを一つ選んでほしい

- 合計カロリーが350以下
- どちらかのフレーバーの種類は必ずELEGANT
- この条件にあうもので一番カロリーが低いもの

flavor1	flavor2	total_calorie
ロッキーロード	ナッツトウユー	340
ナッツトウユー	チョップドチョコレート	343
チョップドチョコレート	ロッキーロード	337
...	...	...

Q3. 同じ感じで、最適なトリプルの組み合わせもほしいな...

3

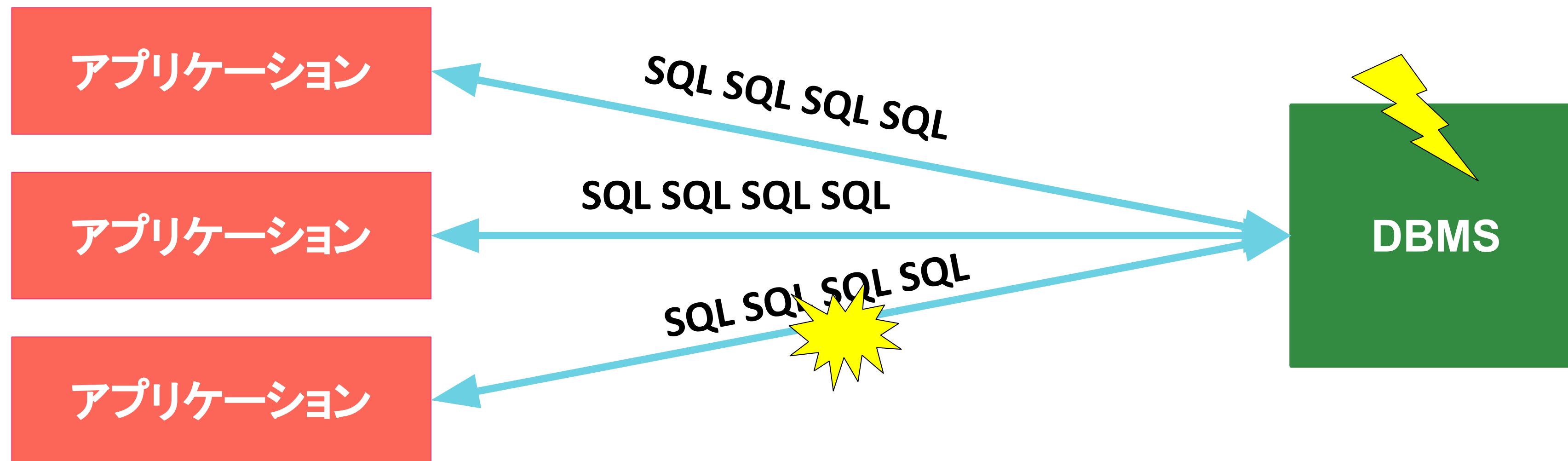
トランザクション



# 背景

---

- 同じレコードが同時に変更されることがある
  - 競合状態をどう回避したらいい？
- 口座振り込みの途中でシステム障害が起きることがある
  - DBをどんな状態に復旧したらいい？（正しい状態とは何か）

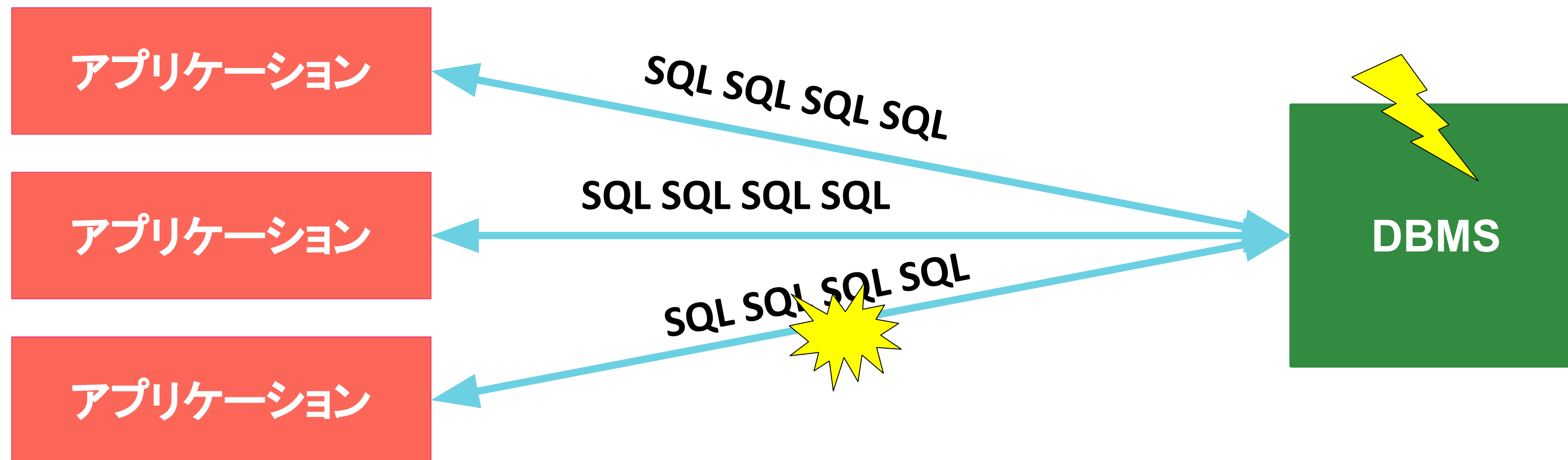


# 背景

- 同じレコードが同時に変更されることがある
  - 競合状態をどう回避したらいい？
- 口座振り込みの途中でシステム障害が起きることがある
  - DBをどんな状態に復旧したらいい？（正しい状態とは何か）

← 同時実行制御

← クラッシュリカバリ





# トランザクションの2つの機能

---

- 同時実行制御
  - 同時アクセスにより起こりうるデータの不整合を防ぐこと
- クラッシュリカバリ
  - 有事の際に、データベースを正しい状態で復旧すること

# トランザクション

---

- トランザクション (TX) とは、アプリケーションがDBに対する複数の読み書きを論理的な単位としてまとめる方法
- DBMSはTXを変更の単位とみなす
  - TXの変更処理は、全体として成功するか中断(失敗)するかのどちらか(シンプル！)
- これによってアプリケーションはよりシンプルに書ける
  - エラーが生じたら中断できるし、安全にリトライすることもできる
  - 潜在的なエラーや並行性の問題を気にしなくて良い

例：口座振り込みのトランザクション

1. Aさんの預金残高を確認
2. Aさんの口座から1万円を差し引く
3. Bさんの口座に1万円を振り込む

アプリケーション

DBMS

成功 or 中断

# トランザクション in SQL

---

- 新しいTXは、**BEGIN**で開始でき、**COMMIT** or **ROLLBACK** で終了できる
  - If **COMMIT** -> DBMSは全ての変更を永続化するか、中断する
  - If **ROLLBACK** -> 全ての変更が取り消される(なかったことになる)
- 中断は、アプリケーション自らが行うか、DBMSによって起こされるのがどちらか

トランザクションのコミット

```
BEGIN;  
SQL文;  
SQL文;  
SQL文;  
COMMIT;
```

トランザクションのアボート

```
BEGIN;  
SQL文;  
SQL文;  
SQL文;  
ROLLBACK;
```

# ACID特性

---

- TXが持つ特性、あるいはTXが提供する安全性の保証
  - ただし、分離性の意味は曖昧になりがち

**原子性 (Atomicity):** トランザクションに含まれる操作全てが成功か中断になる

**一貫性 (Consistency):** トランザクションを実行した前後ではデータの一貫性が損なわれない

**分離性 (Isolation):** 同時実行している複数のトランザクションは互いに独立している

**永続性 (Durability):** 一旦コミットが完了したトランザクションによる変更は永続化される

# 原子性と永続性

---

- 原子性

- TXが障害のために成功しなかったら、確実に中断されそれまでの操作が破棄されることを保証

- 永続性

- TXが成功したら、システム障害が起きようと変更は失われないことを保証
  - 完全な永続性は存在しない

例: 口座振り込みのトランザクション

1. Aさんの預金残高を確認
2. Aさんの口座から1万円を差し引く
3. Bさんの口座に1万円を振り込む

原子性がないと、この間で障害が起きると1万円消える

永続性がないと、TX終了後、気づいたら1万円振り込んでいなかったことになったりする。。

# 原子性と永続性

---

- 代表的な実現方法はロギング
  - すべての操作ログを出力しておき、
    - TX中断後、逆順に操作をやり直す (Undo)
    - システム障害後、コミット済みだがディスクに未反映の操作を再実行 (Redo)



# 分離性

---

- 並行して実行されたTX同士が互いに影響を与えないことを保証
- 分離性の目的は、並行性の問題をアプリケーションから隠すこと

## 例:ダーティ・リード(並行性の問題)

口座振り込みのトランザクション

1. Aさんの預金残高を確認
2. Aさんの口座から1万円を差し引く

1. Bさんの口座に1万円を振り込む

預金確認のトランザクション

1. Aさんの預金残高を確認
2. **Aさんの預金を出力**

**分離性がないと、1万円差し引かれた値が  
みえてしまう**

# どう実現するか

---

- 単純に逐次処理していくという方法も考えられるが、現実には性能を考慮する必要がある
- TX間でどの程度の影響を許すか(どういう並行性の問題を許すか)にはいくつかのレベルが考えられていて、実現方法もそのレベルによって異なる

# 分離性の種類

## 分離レベル (SQL-92)と並行性問題の関係

	ダーティ・リード	ファジー・リード	ファントム・リード	分離性
リード・アンコミテッド	許可	許可	許可	低
リード・コミテッド	抑止	許可	許可	
リピータブル・リード	抑止	抑止	許可	
シリアライザブル	抑止	抑止	抑止	高

<https://xtech.nikkei.com/it/article/COLUMN/20080123/291846/>

下の表ではSQL-92では指摘されなかったものも含まれている

Table 4. Isolation Types Characterized by Possible Anomalies Allowed.								
Isolation level	P0 Dirty Write	P1 Dirty Read	P4C Cursor Lost Update	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	A5A Read Skew	A5B Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

<https://arxiv.org/pdf/cs/0701157.pdf>

# ファジーリード

---

- **ダーティーリード**は、他のTXの**未コミット**な書き込みが読み取れる問題
  - ロールバックされると存在しないデータを読み込んだことになる
- **ファジーリード**は、読み出した**行**が他のTXにより更新/削除され、同じTXで再度同じ行を読み込んだときに、その行が更新/削除される問題

## 口座振り込みのトランザクション

1. Aさんの預金残高を確認
2. Aさんの口座から1万円を差し引く
3. Bさんの口座に1万円を振り込む

## 預金確認のトランザクション

1. Aさんの預金残高を確認
2. **Aさんの預金**を出力

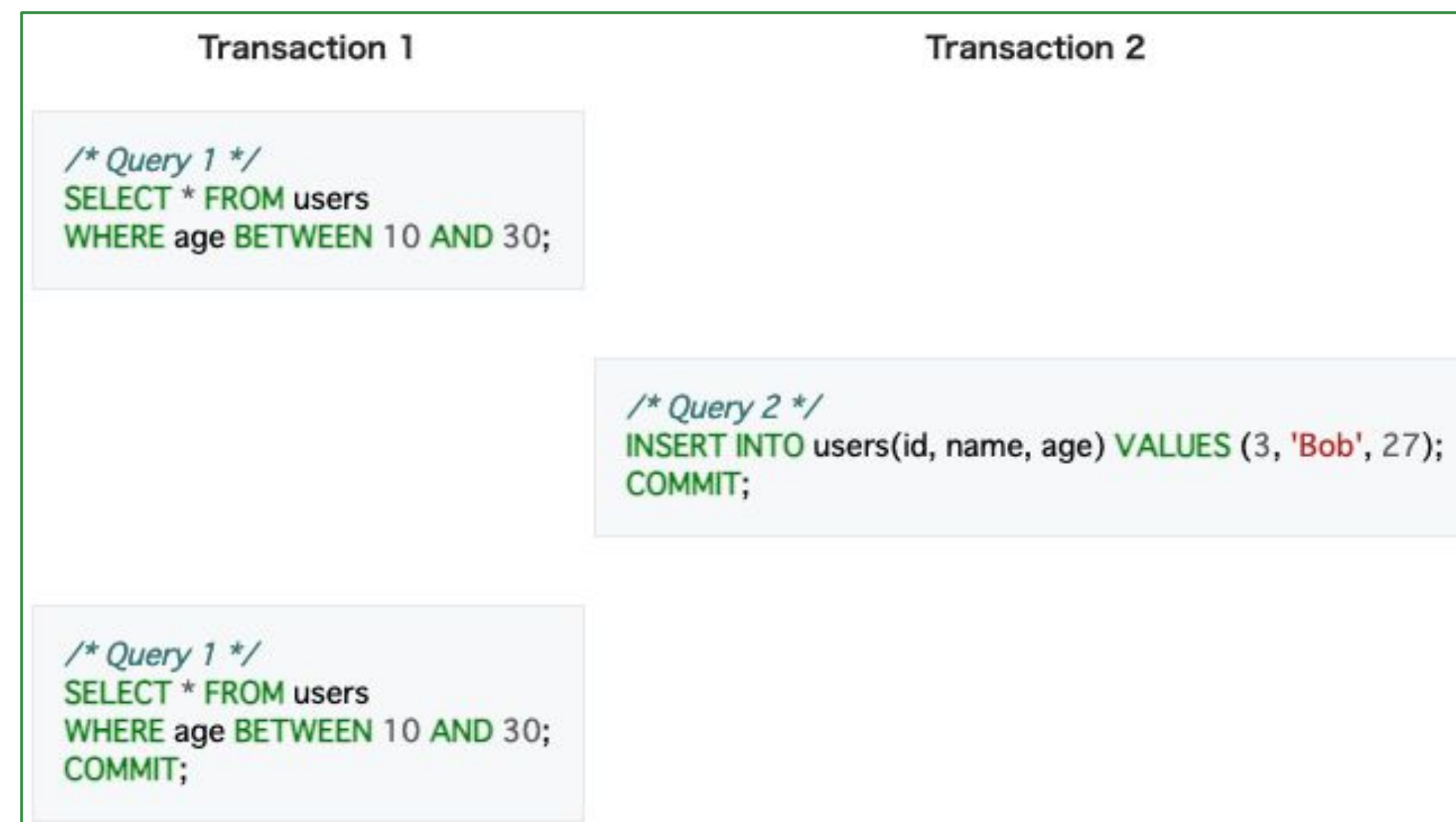
出力値が変わる

1. Aさんの預金座高を確認
2. **Aさんの預金**を出力



# ファントムリード

- 他のTXの書き込みによって、同じ検索条件で読んでいるのに、あったはずの行が消えたり、なかった行が現れたりする問題
  - ある**検索条件を満たす行の集合**を読み込んだ後に、他のTXがその検索条件に合うような行を追加/削除するときにかかる



# 分離レベル in MySQL

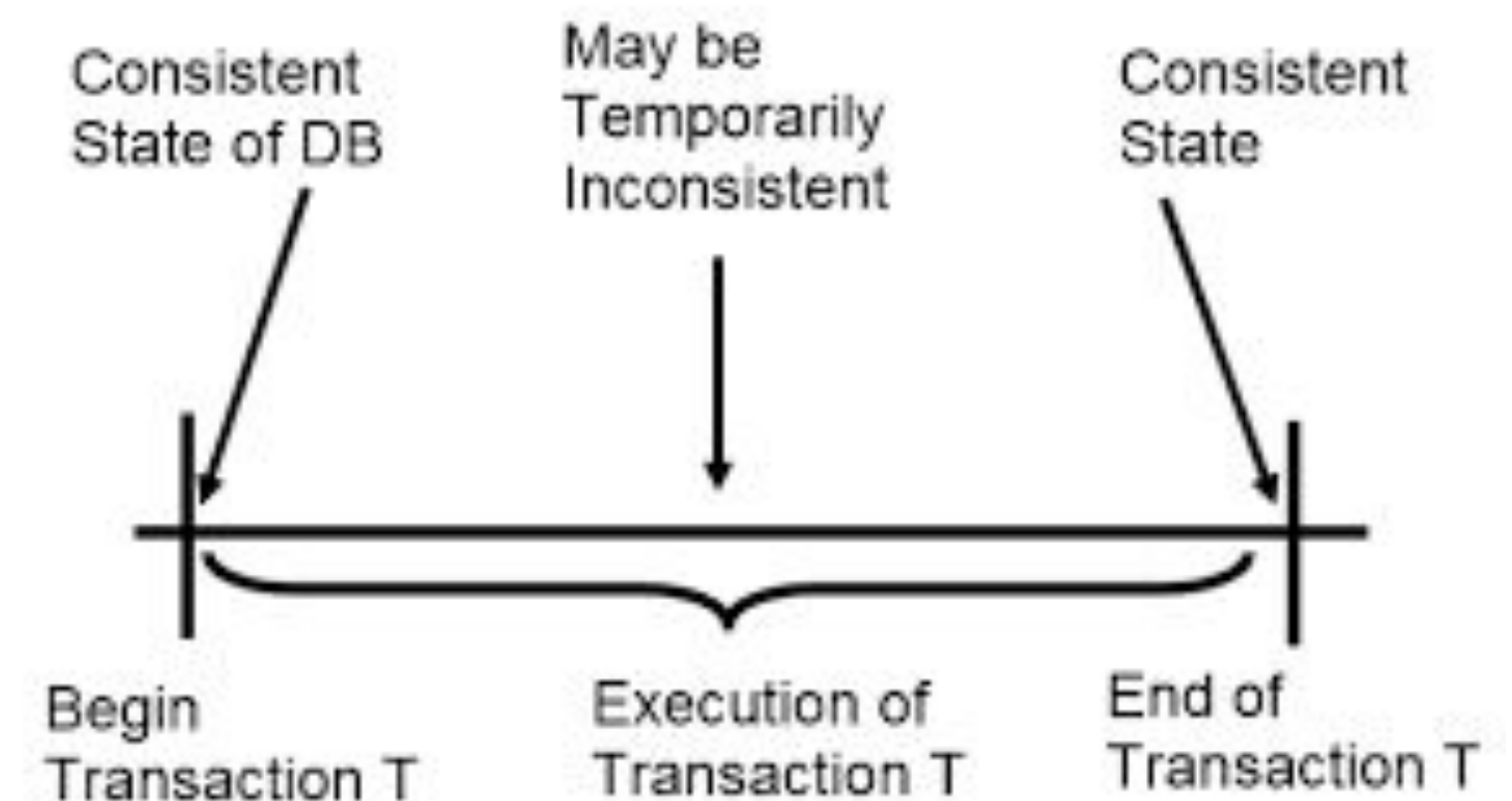
---

- MySQL 5.6 (InnoDB)
- サポートされている分離レベル
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ (デフォルト)
  - SERIALIZABLE
- 実現方法
  - MVCCとロック



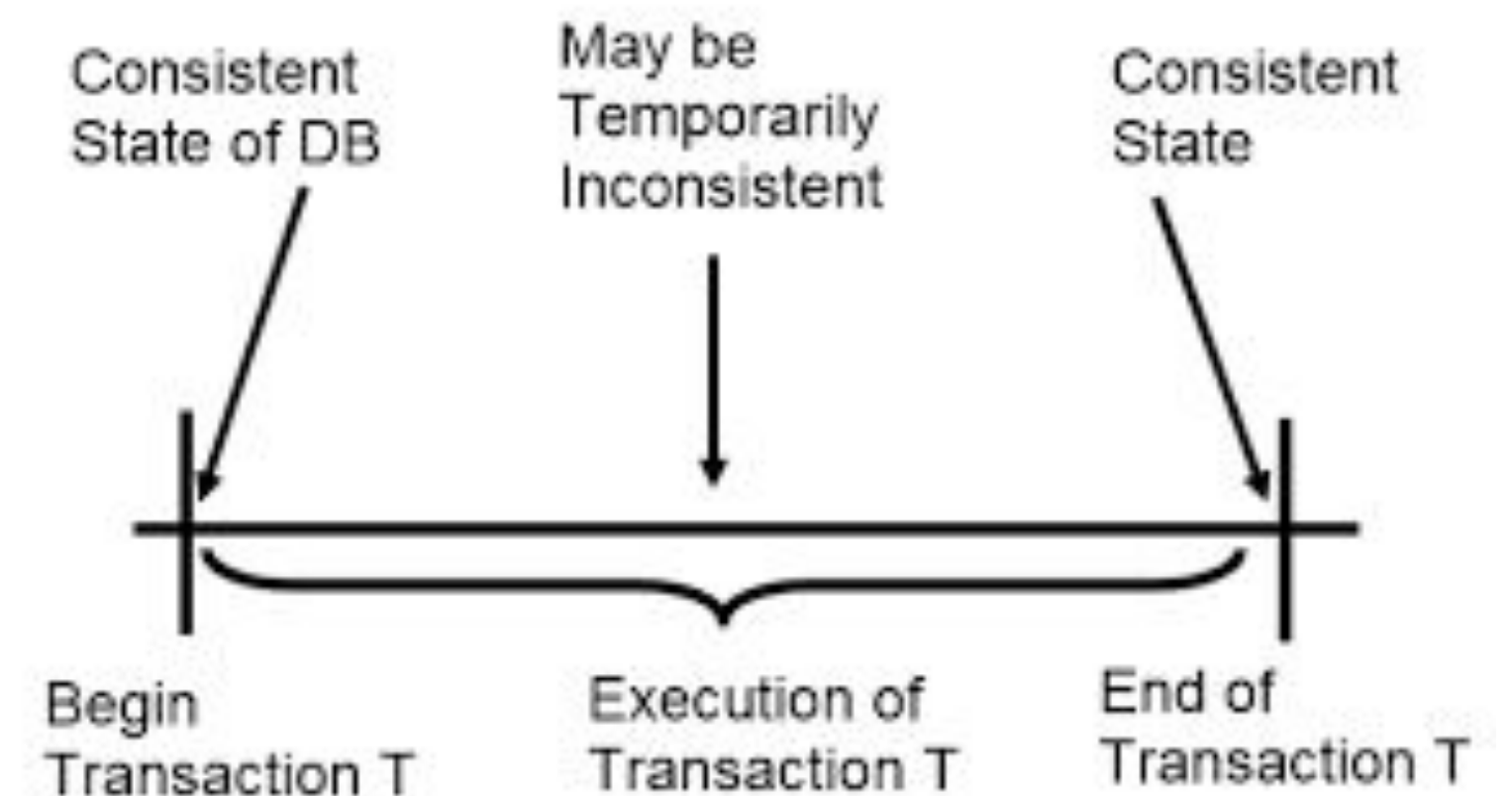
# 一貫性

- “TXを実行した前後ではデータの一貫性が損なわれない”
- DBの状態遷移から考えると
  - TX開始前に、DBが一貫性のある状態であれば、終了後は別の一貫性のある状態へと遷移する
  - データに変更はあるものの、一貫性が保たれている
- 一貫性を損なうような操作は処理されず、中断される(一貫性は保たれたまま)



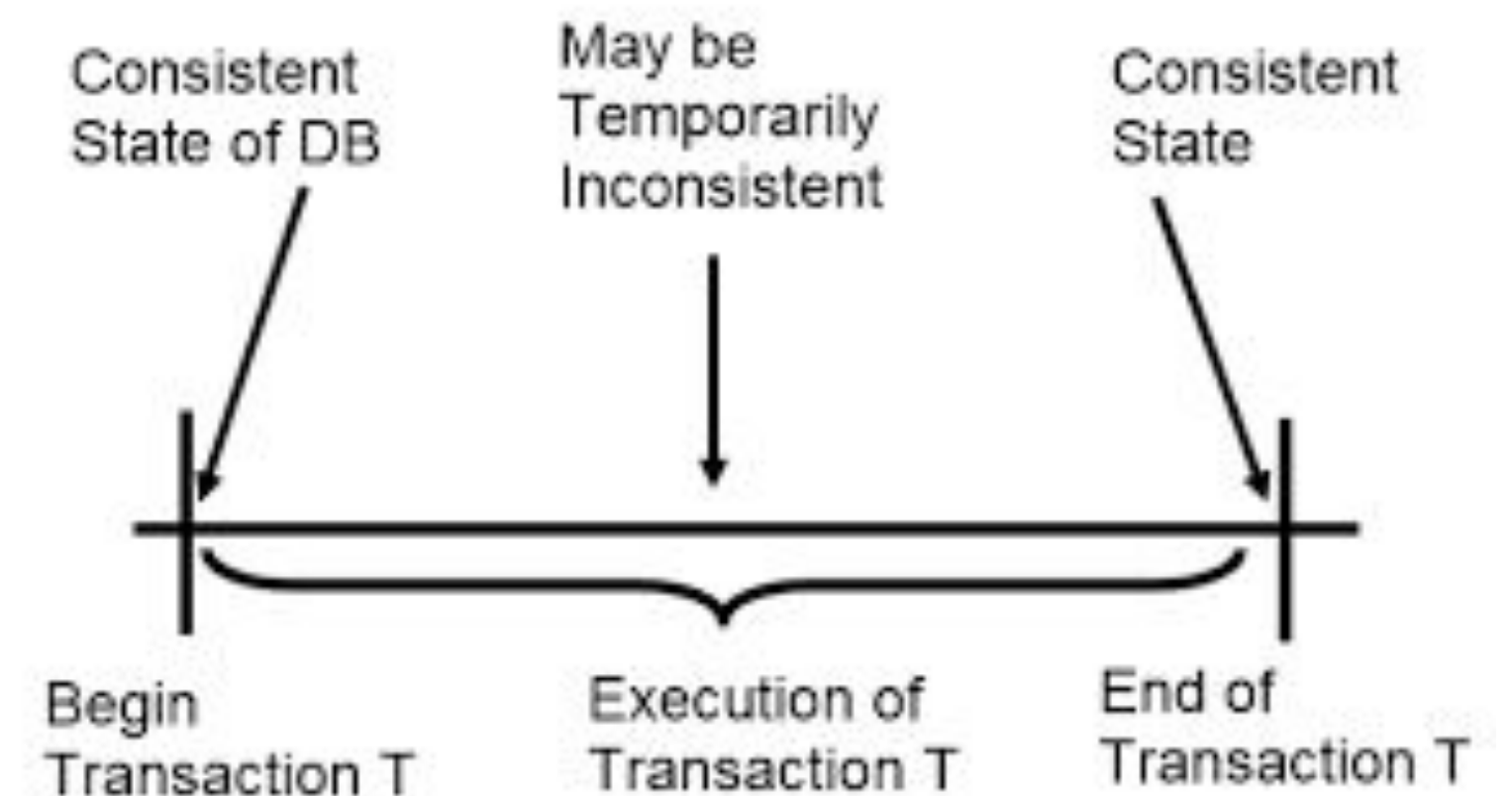
# 一貫性

- モチベーションから考える
  - 多くの場合、データについて常に真でなければならない何らかの言明(不変性)が存在する
    - 例: 口座振込において、引出額と振込額は同じでなければならない(“ビジネスルール”)
  - 一貫性は、TX前後でその不変性を常に満たすことを保証



# 一貫性

- ACIDの中で唯一のアプリケーションの特性
  - DBMSはどのような不変性があるのかわからないので保証しようがない
  - 一貫性を保つようにTXを適切に定義することはアプリケーションの責任
  - (ただし、例えば、外部キー制約などの一種の不変性はデータベースがチェック可能)



# トランザクションの要点

---

- TXの主な機能は同時実行制御とクラッシュリカバリ
- アプリケーションはTXを使うことでよりシンプルになる(DBMSに色々任せられる)
- ACID特性はTXが提供する安全性の保証のこと(分離性には種類がある)
- ACIDを簡単にまとめると

Atomicity: “all or nothing”

Consistency: “it looks correct to me”

Isolation: “as if alone”

Durability: “survive failures”

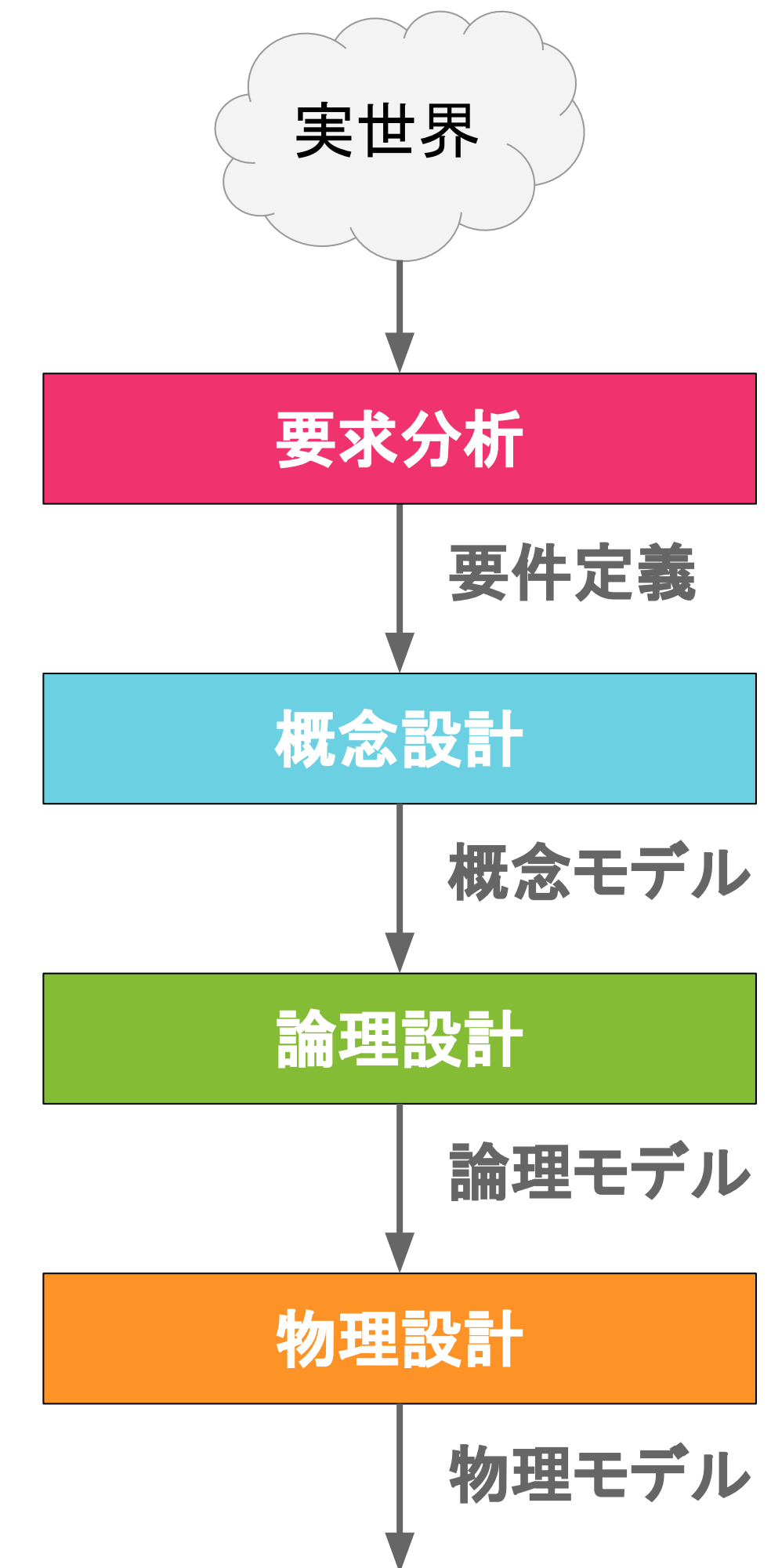
4

# データベース設計



# データベース設計手順

- 要求分析
  - データベースで管理したいデータやデータの使われ方等を整理
- 概念設計
  - 要件定義を元に、DB化の対象となる実世界をモデル化
  - 特定のDBMSのデータモデルには依存しない
  - ERモデルが主流
- 論理設計
  - 概念モデルをDBMSのデータモデルでスキーマに変換
  - スキーマの改善
- 物理設計
  - インデックス、ファイルフォーマットなどの性能チューニング
  - この後のステップにはセキュリティ設計などが含まれる





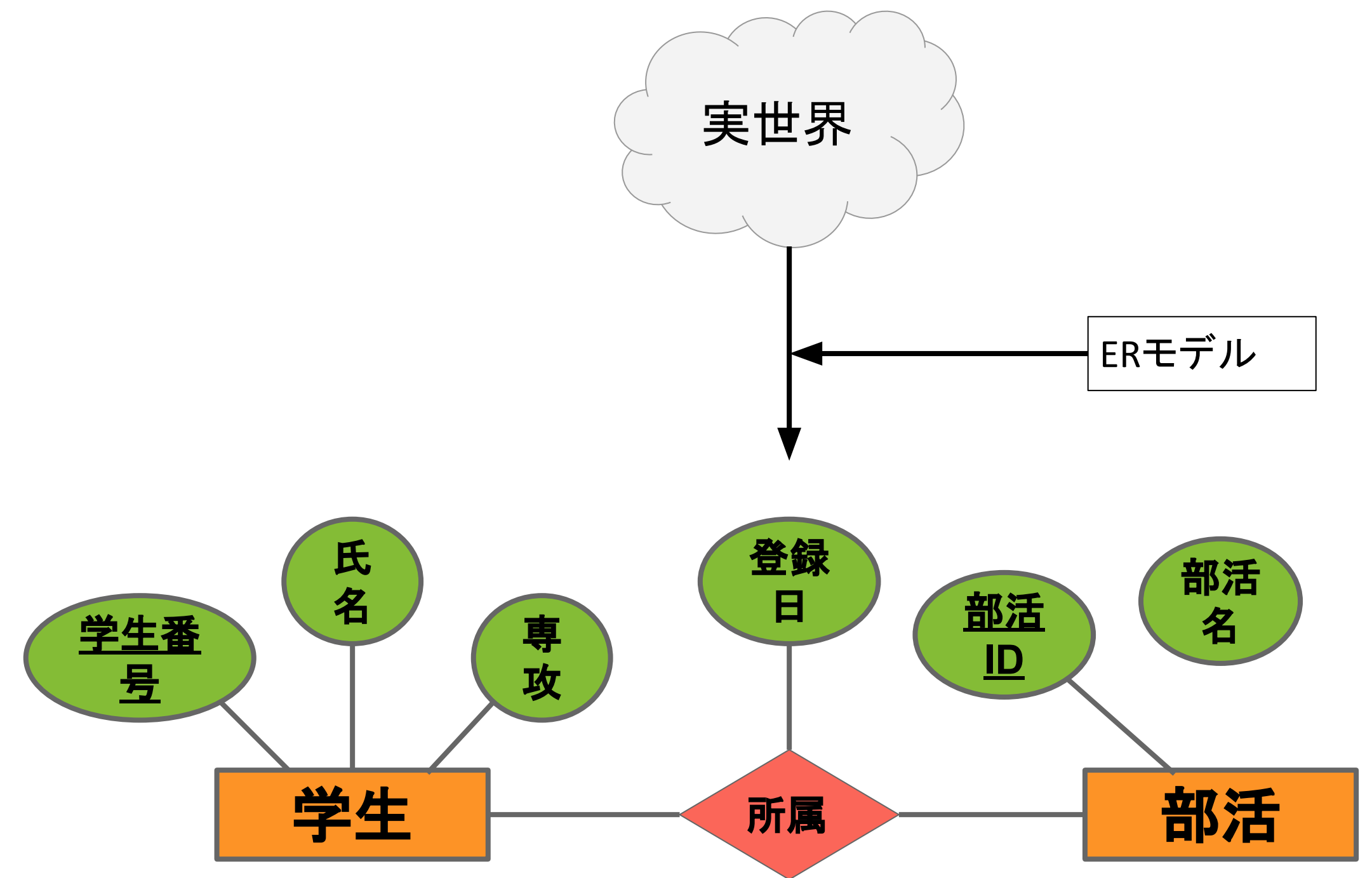
# 概念設計: ERモデル

## ERモデル

- 実体(エンティティ)や関連(リレーションシップ)を使って実世界を表現するモデル
- ERモデルで記述されたものがER図

以下のような質問に答えていく

- 何が実体？
- どの実体同士が関連してる？
- 実体や関連はどんな属性が必要？
- 満たすべき整合性制約(ビジネスルール)は何？



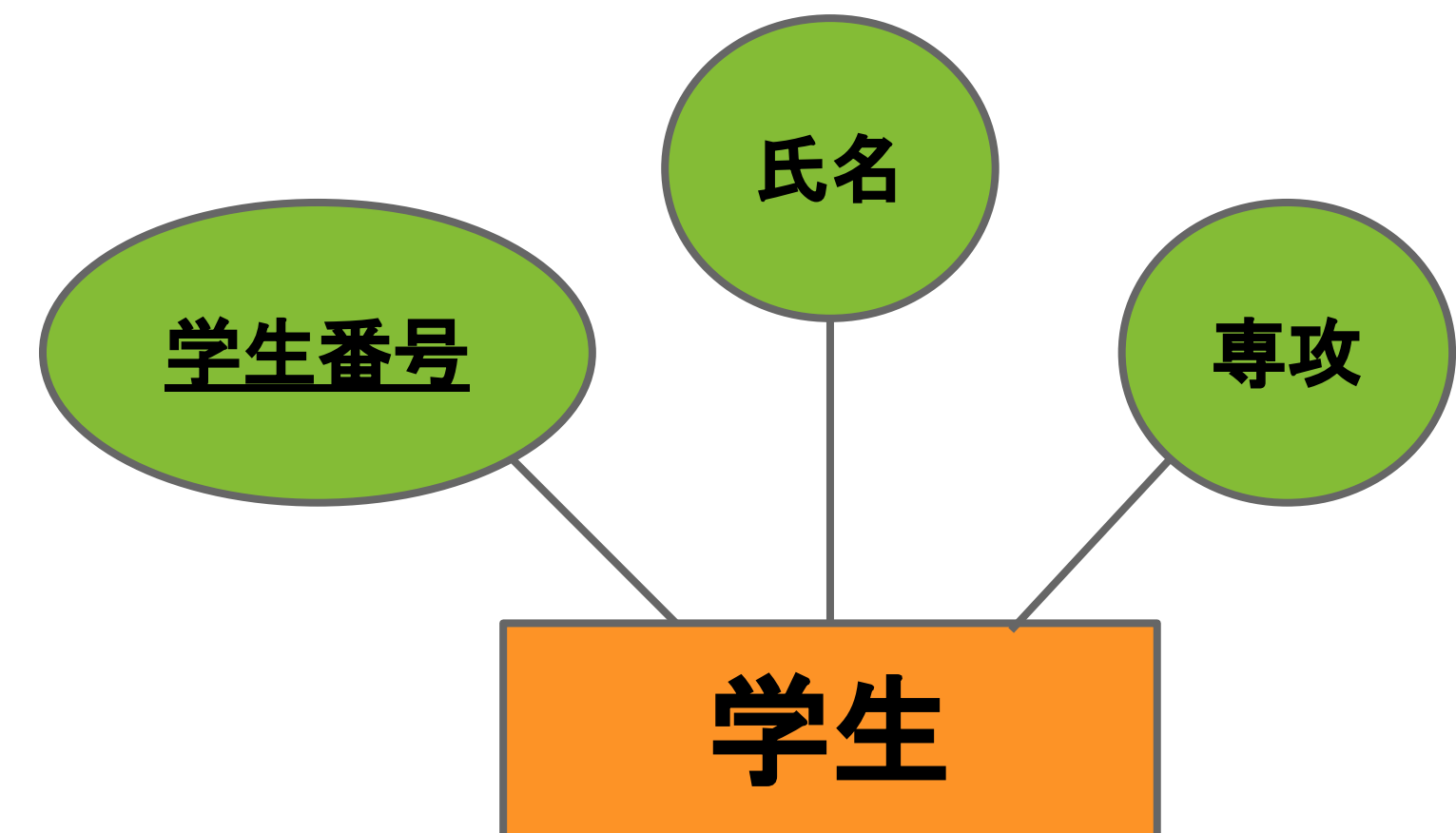
ER図 (Chen記法)



# エンティティ

---

- 実体 (entity)
  - DBで表現したい実世界の対象物(例: 山田、佐藤 ...)
  - 属性値の集まりから構成される(例: 学生番号、氏名 ...)
- 実体集合 (entity set) or 実体型 (entity type)
  - 同一種類の実体の集まり(例: 学生、部活)
  - ERモデルは集合単位で実世界を表現！
  - 主キーを1つ持つ (アンダースコア)



# キー用語

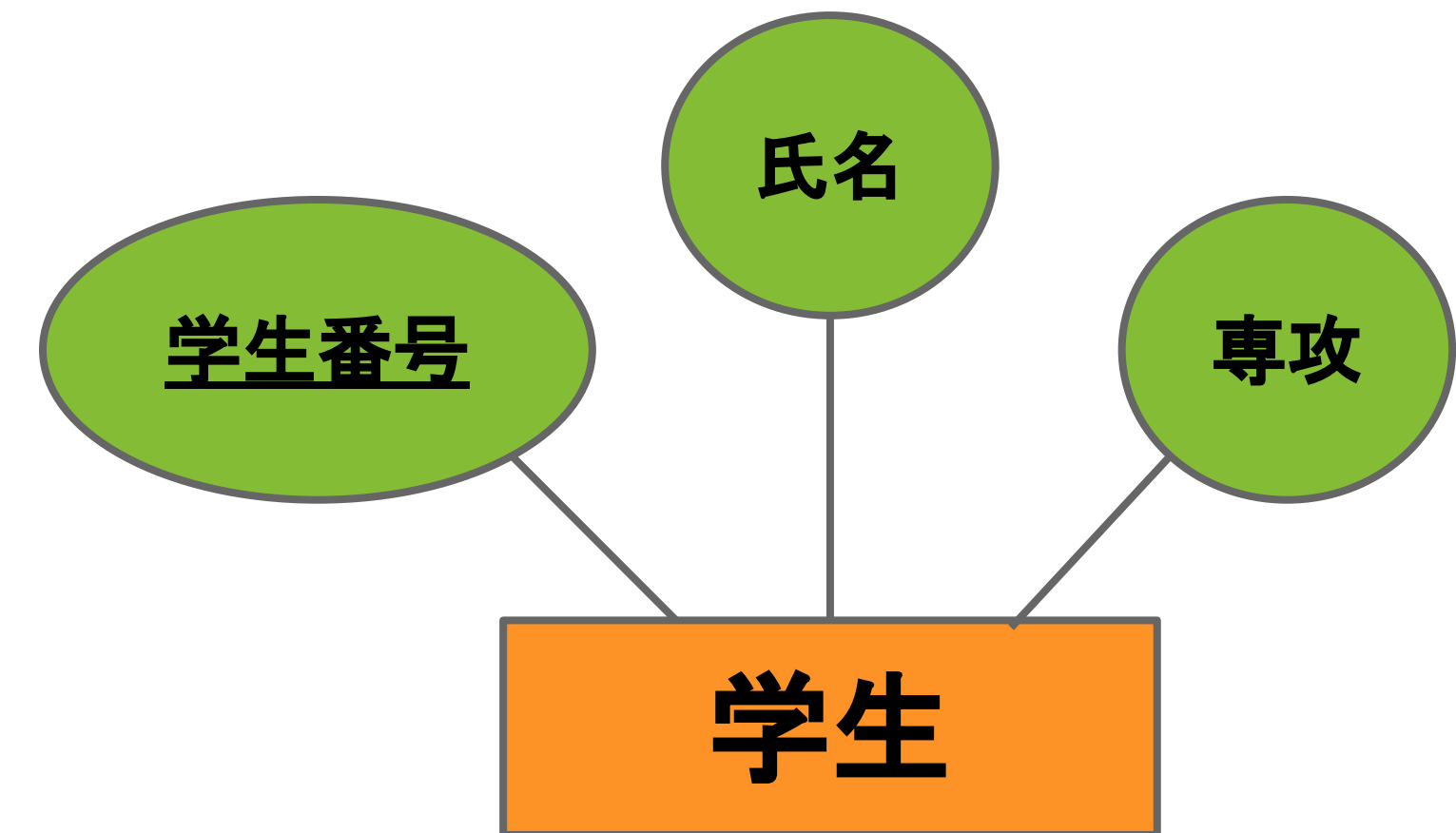
---

- キー

- 実体集合の個々の実体を一意に識別できる属性もしくは属性の組合せで極小なもの
  - 例: 学生番号, 学生の氏名と住所
- 複数存在しうるので、**候補キー**とも呼ばれる

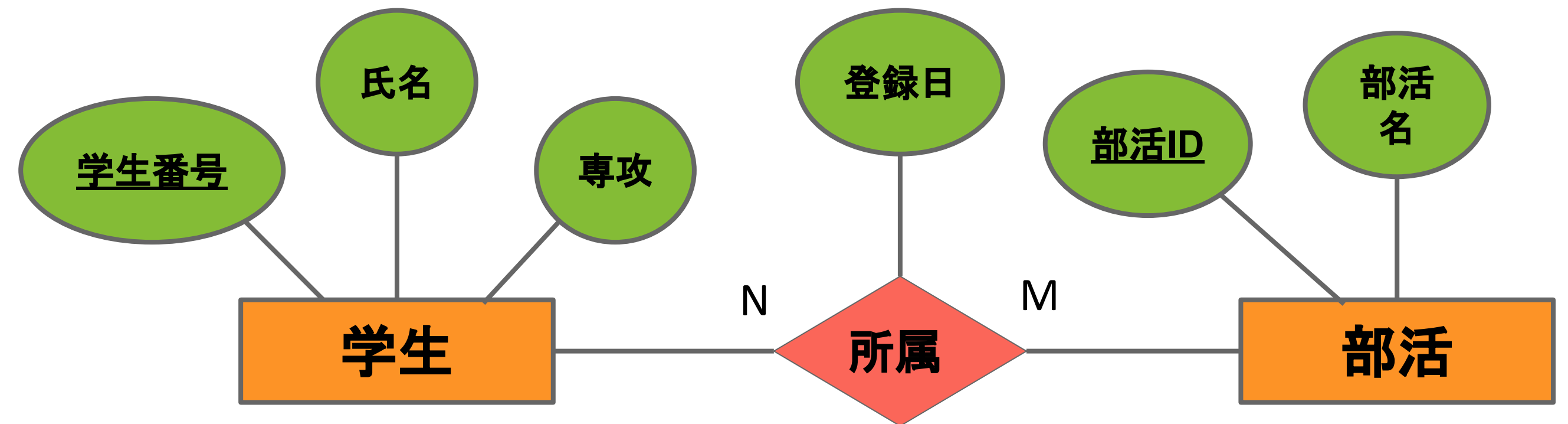
- 主キー

- キーの内、システムが使うのに最も便利なもの
- 実体は主キーを1つ持つ(キー制約)



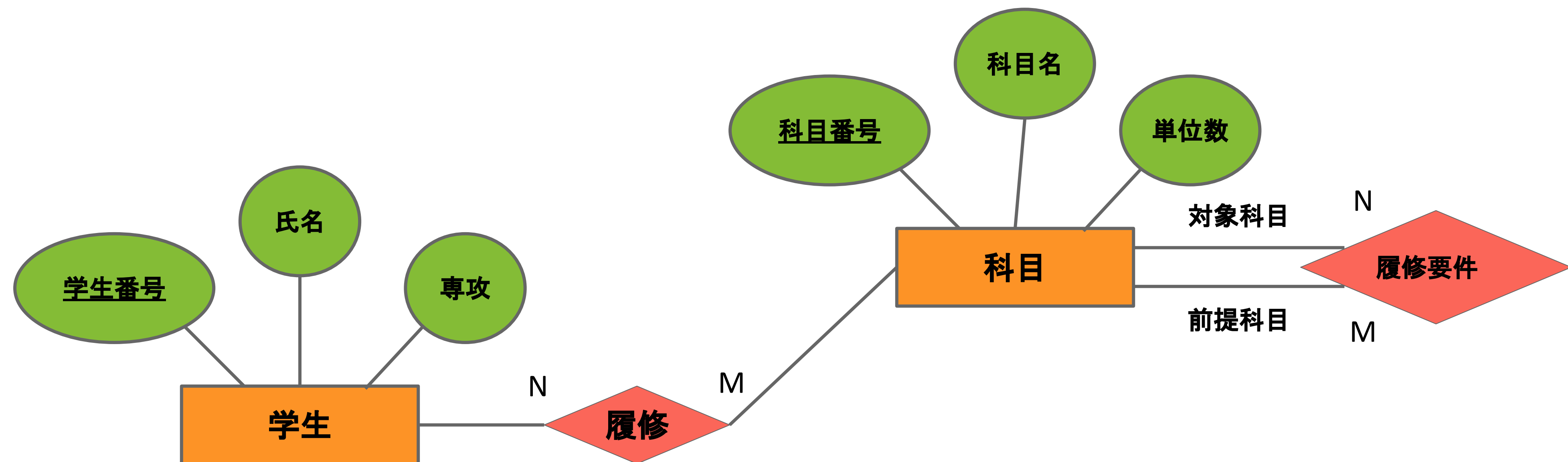
# 関連

- 関連 (relationship)
  - 実体間の相互関係
- 関連集合 or 関連型
  - 実体集合同様、総体として捉える
    - 例: 「学生は部活に所属する」という具合に
  - 関連集合にも属性を付与可能



# 複数の関連

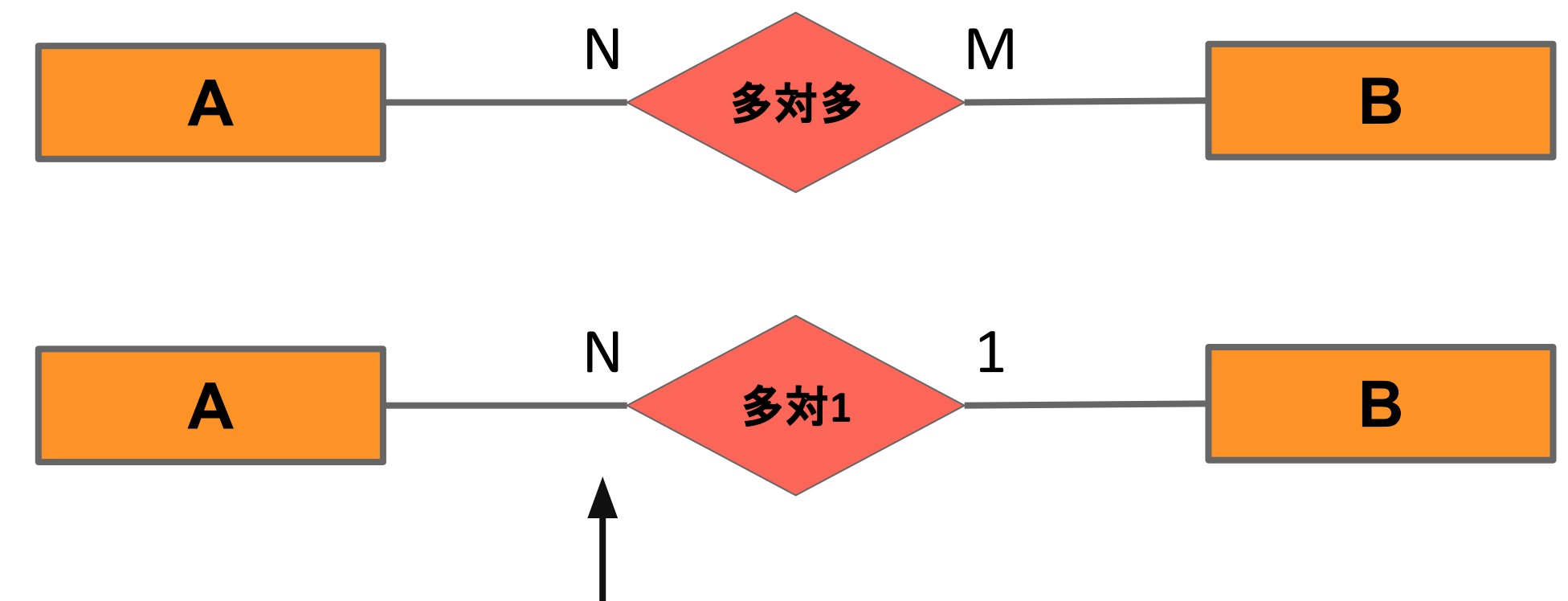
- 実体集合は複数の関連集合に参加できる
- ER図では、同一の実体集合に対する関連集合も記述可能
  - 役割(role)を明記



# カーディナリティ

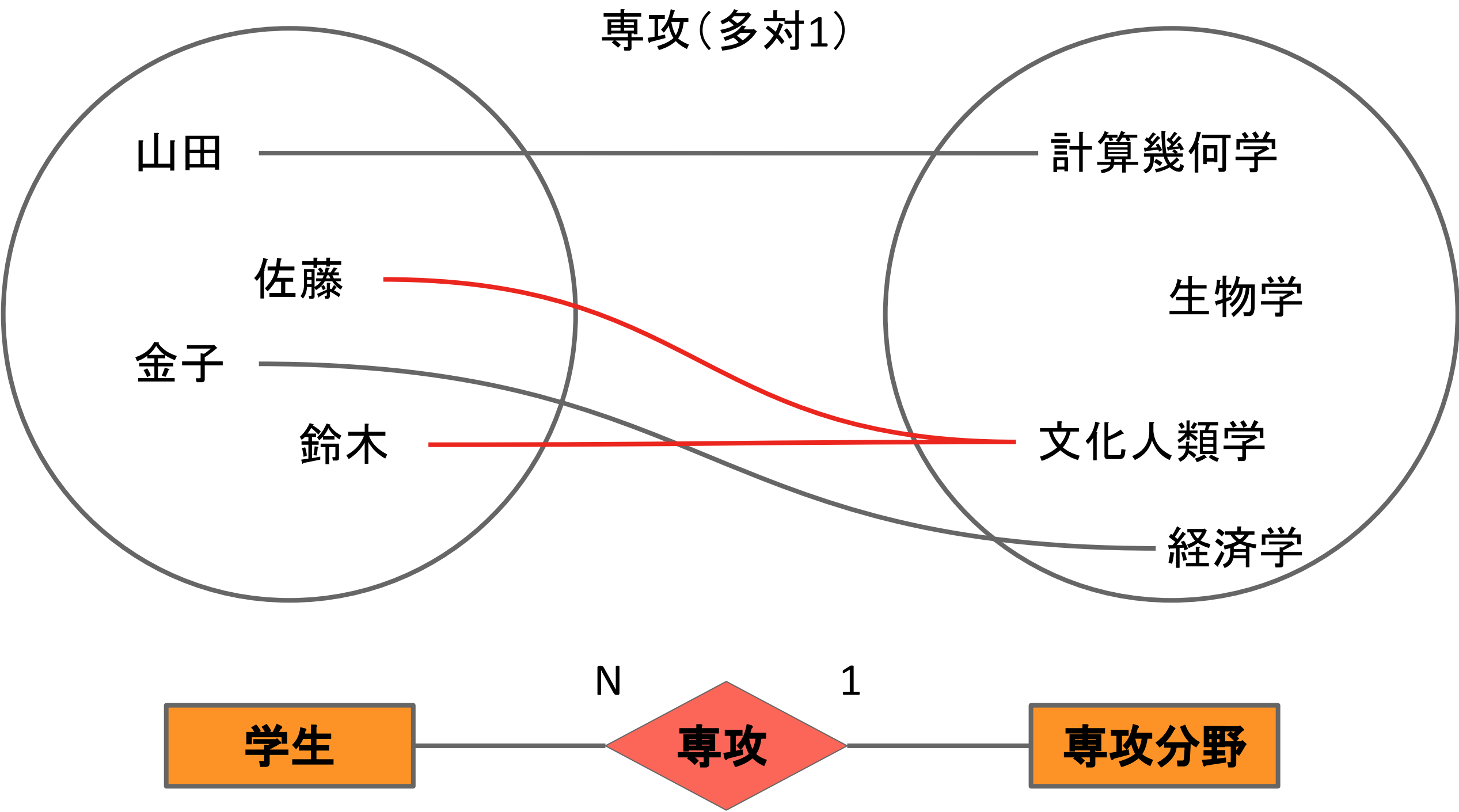
- カーディナリティとは
  - 関連集合における実体間の量的な対応関係を指す記号(1,N,M)
  - **関連集合に参加できる実体の最大数**を表す
  - 一種の制約
- 対応関係による関連集合の分類
  - 多対多関連集合 (many-to-many)
  - 1対多関連集合 (1-to-many)
  - 多対1関連集合 (many-to-1)
  - 1対1関連集合 (1-to-1)

表記法の例

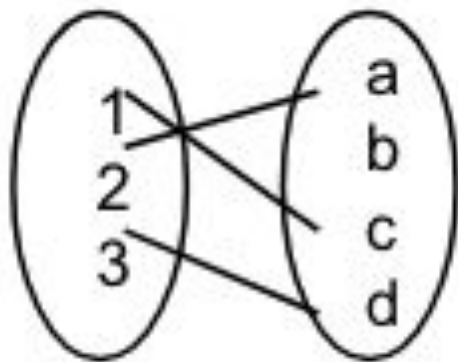


↑  
Bの実体がこの関連集合に参加できる最大の数

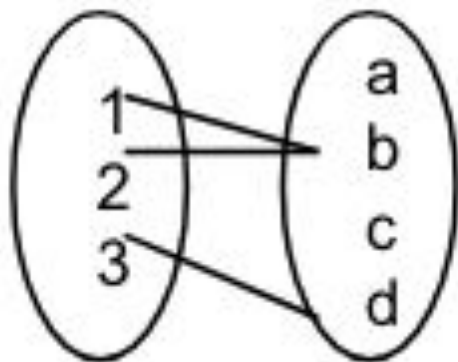
# 対応関係のイメージ



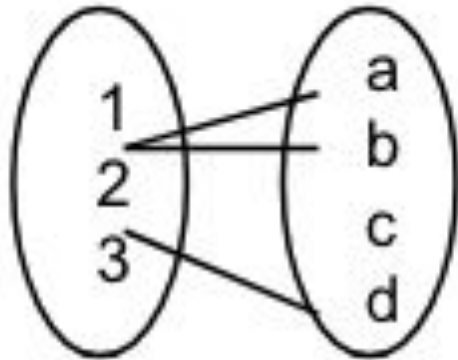
One-to-one:



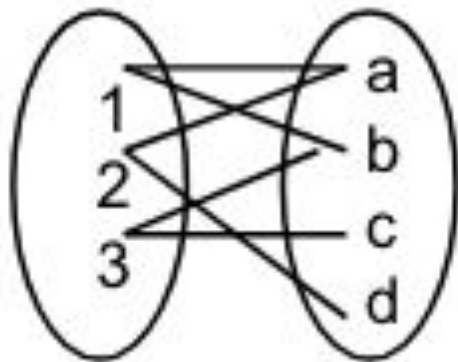
Many-to-one:



One-to-many:



Many-to-many:



# 各対応関係に当てはまる表現

- 多対多

- 例: 各学生は**複数**の部活に所属できて、各部活は**複数**の学生を部員として登録できる



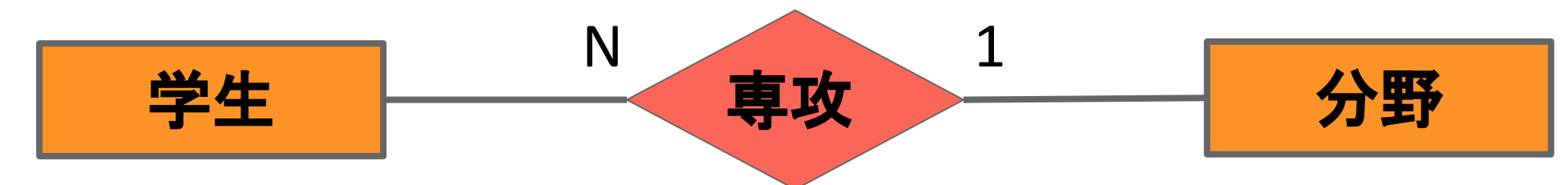
- 1対多

- 例: 各学生は**複数**の部活の部長を務めることができるが、各部活の部長は**1人**まで



- 多対1

- 例: 各学生は**1分野**まで専攻分野を選択できるが、1つの分野は**複数**の学生が専攻している



- 1対1

- 例: 各学生が住める部屋は**1部屋**まで、かつ各部屋に住める学生は**1人**まで





# カーディナリティとキー制約

---

- 「**最大1人まで**」のような表現には**キー制約**が隠れている
  - 例: 各部活の部長は**1人まで**
    - 部長関連集合に参加する部活を、部活の主キーで一意に特定すること  
ことで実現
- i.e., 1つのキー制約で**1対多関連**を実現できる
  - 1対1関連は両サイドからキー制約を課すことで実現できる



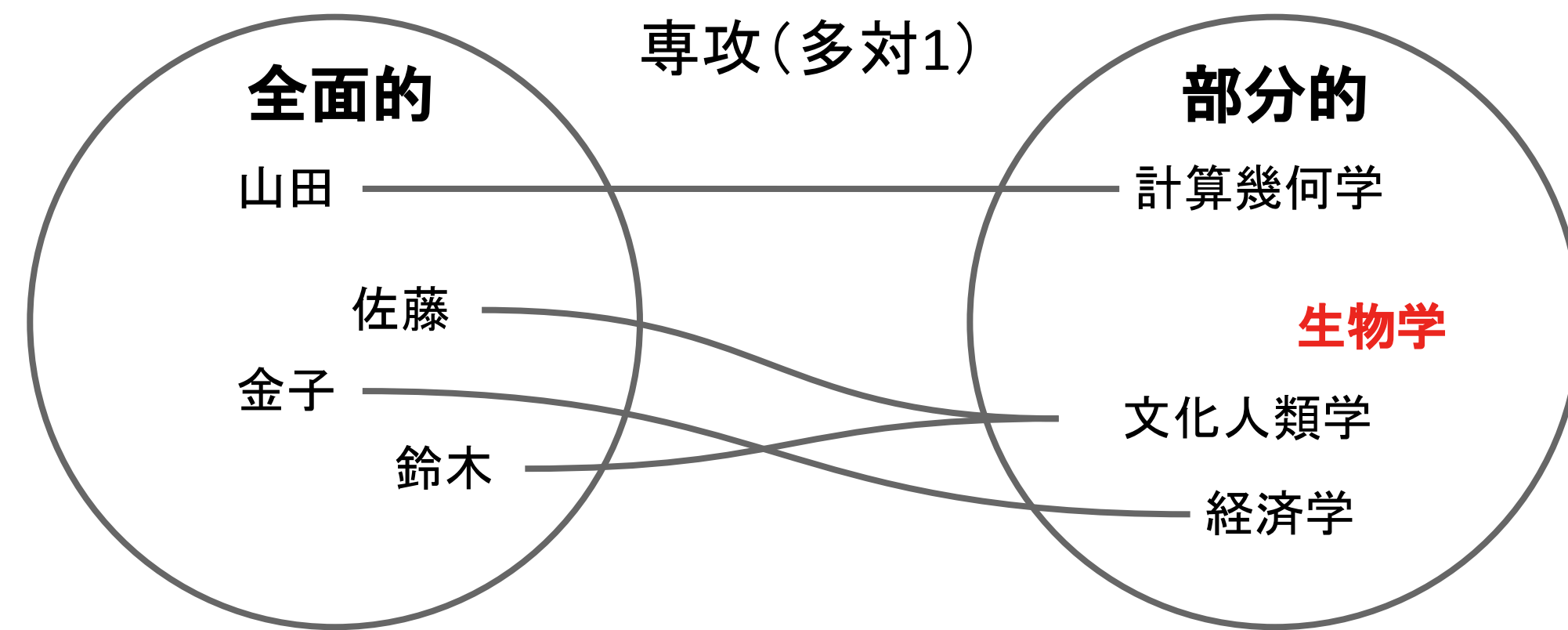
# 参加制約

---

- 参加制約(participation constraint)
  - 関連に参加しない実体を許すかどうか
  - 制約を課すことで「**最低1人**」のような表現が可能
- 参加制約の種類
  - **全面的** (total): 全ての実体は関連への参加が**義務** (mandatory)
  - **部分的** (partial): 関連へ参加は**任意** (optional)
- 全面的参加制約の例
  - 学生は**最低1つ**の専攻分野を選択しなければならない
  - 部活には**最低1人**の部長が必要

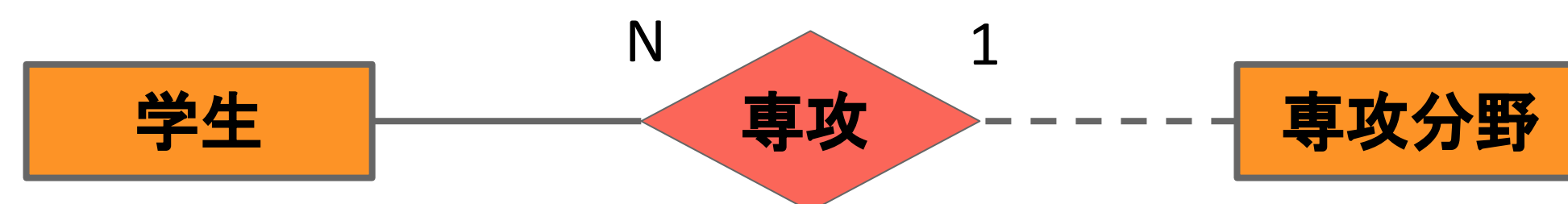
# 参加制約のイメージと表記法

- 参加制約のイメージ



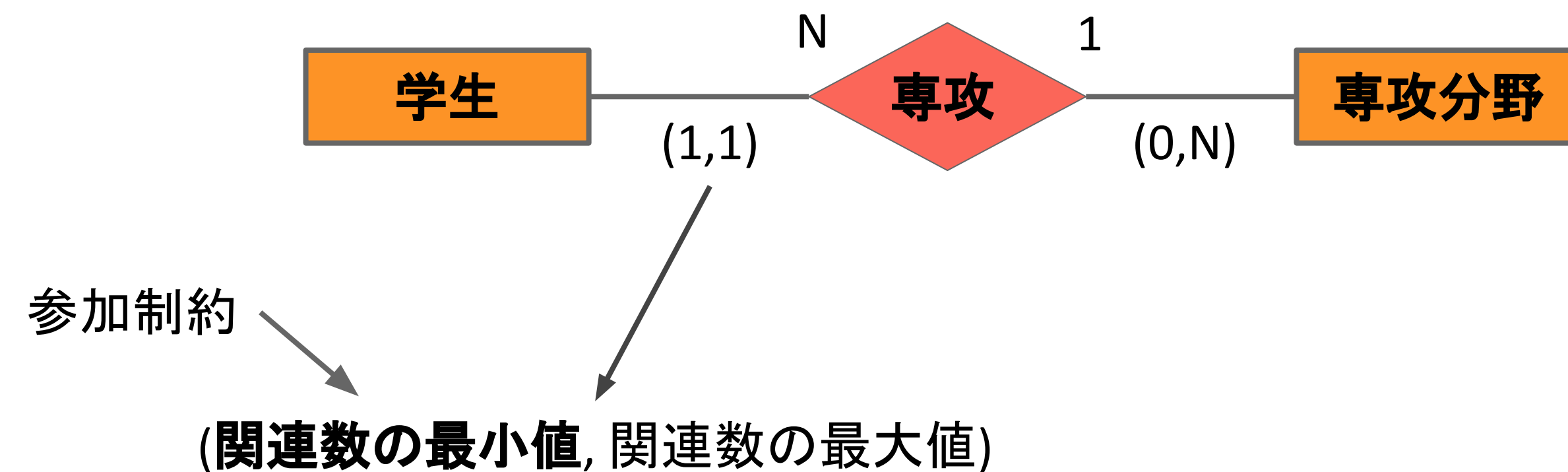
- 表記法

- 参加が全面的/義務なら実線
- 参加が部分的/任意なら破線



# Chen Notation

- 対応関係と参加制約の両方を明記する表記法

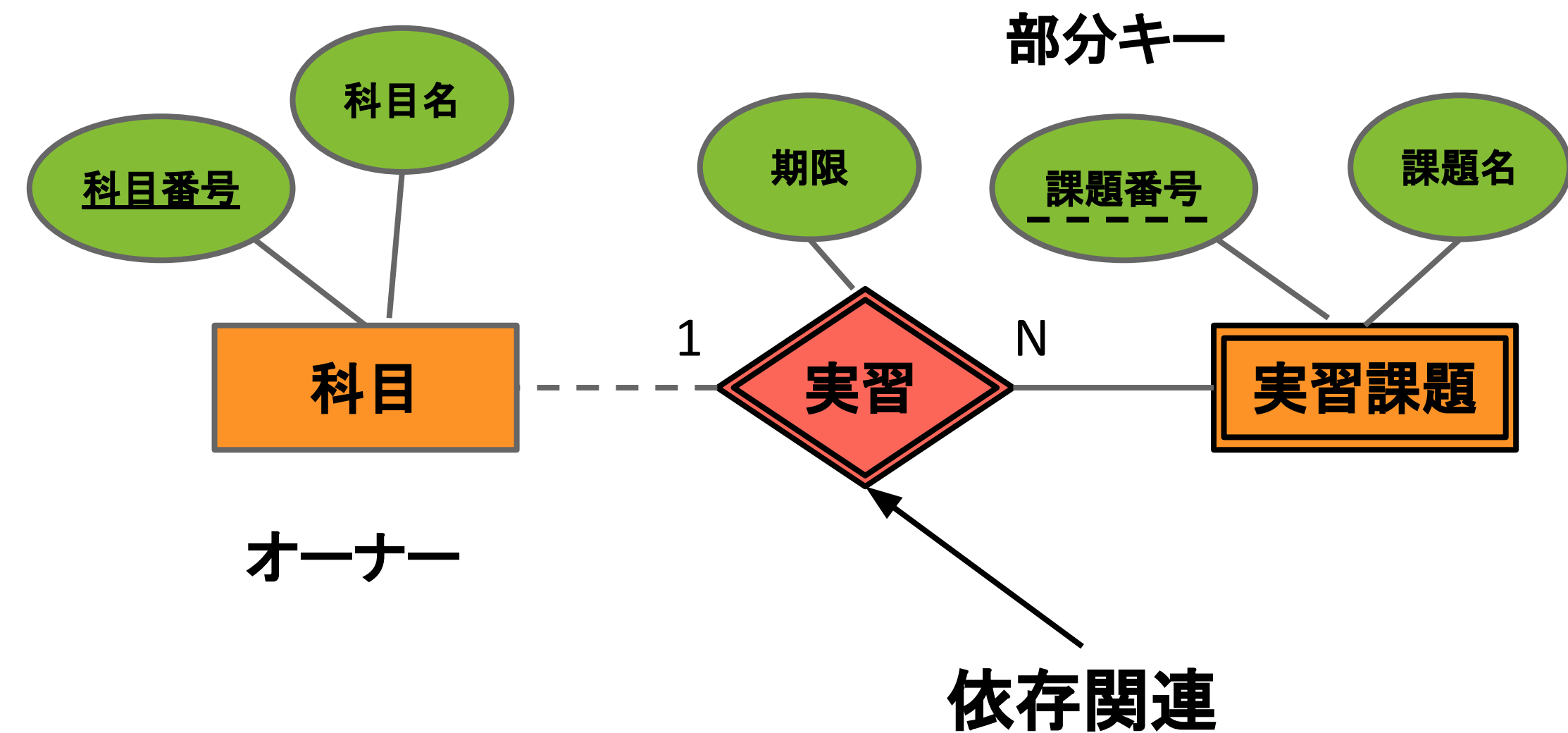


- 例: 専攻集合に個々の学生は最小何回、最大何回まで含まれるか

専攻集合{  
(山田、計算機科学),  
(佐藤、文化人類学),  
(金子、経済学),  
(鈴木、文化人類学)  
}

# 弱実体集合

- 弱実体集合 (weak entity set)
  - 別の実体集合(オーナー)に依存した存在
    - オーナーなしでは存在できない
  - オーナーのPK+自分が持つ部分キーで一意に識別できる
- 依存関連集合
  - オーナーと共に参加必須な1対多関連集合
    - 弱実体集合の参加は義務
- 例えば
  - 物理の実習1は「物理-1」、数学の実習1は「数学-1」のような具合で一意に特定できる



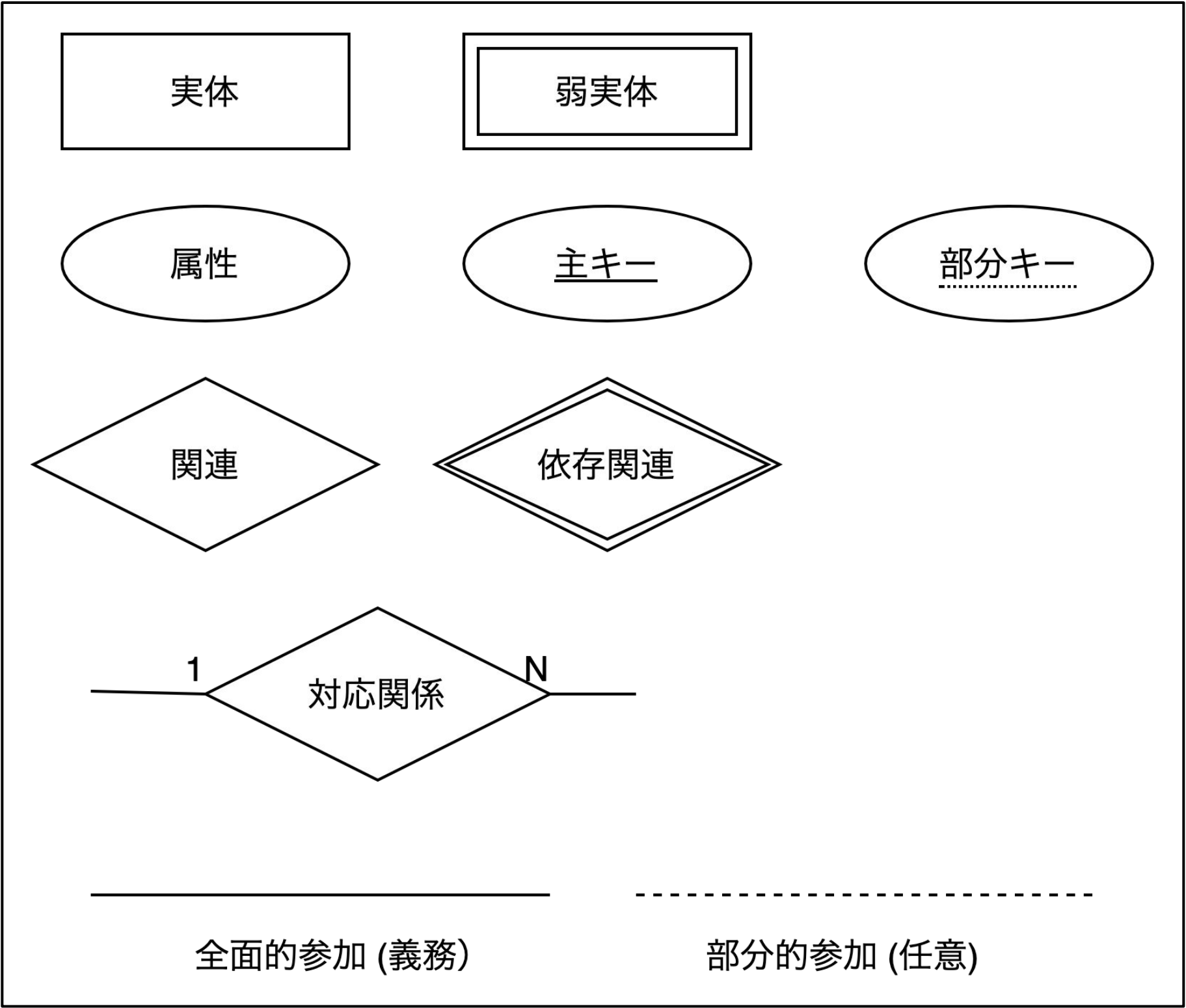
# ER図を書いてみよう

Q. スポーツチームとプレイヤー情報をデータベースで管理したい

以下の要件をもとにER図を書いてみよう! (書き方は自由、[drawio](#) 推奨)

- チームは一意的なチーム名と彼らがプレーする専用のスタジアムを持つ
- コーチは氏名を持つ
- プレイヤーはプレイヤーID、氏名、スコアを持つ
- データベースでは、どのプレイヤーがどのチームのどのポジションでプレーするかに加えて、誰がチームのキャプテンで、誰がチームのコーチかというデータも管理したい
- チームには複数のプレイヤーがいて、その中の1人は必ずキャプテン
- 基本的には、プレイヤーは最大1チームでプレーしているが、一時的に無所属になるプレイヤーもいる
- チームにはコーチが必ず必要だが、複数のコーチがいてもかまわない
- コーチの識別は、どのチームでコーチングするかで決まる

ER図 チートシート





# 補足: 様々な表記法

*Chen Notation*

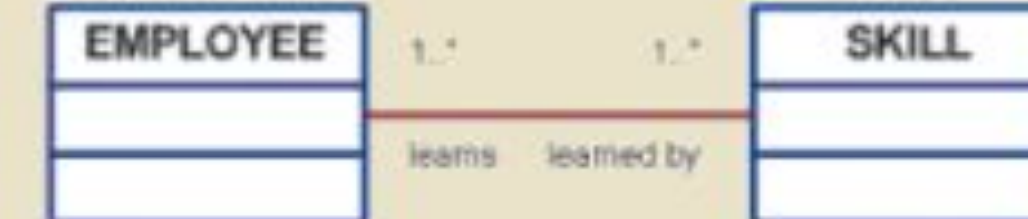
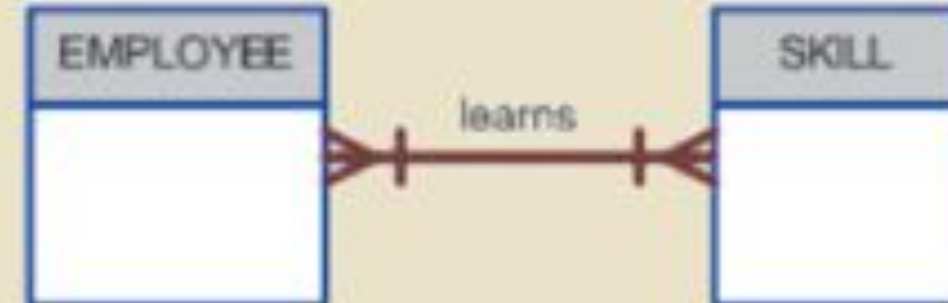
*Crow's Foot Notation*

*UML Class  
Diagram Notation*

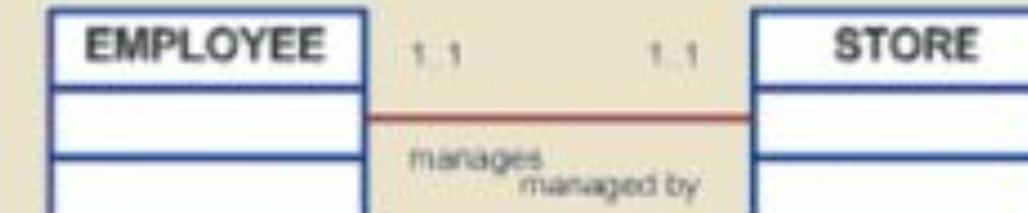
A One-to-Many (1:M) Relationship: a PAINTER can paint many PAINTINGs; each PAINTING is painted by one PAINTER.



A Many-to-Many (M:N) Relationship: an EMPLOYEE can learn many SKILLs; each SKILL can be learned by many EMPLOYEEs.



A One-to-One (1:1) Relationship: an EMPLOYEE manages one STORE; each STORE is managed by one EMPLOYEE.



# 論理設計

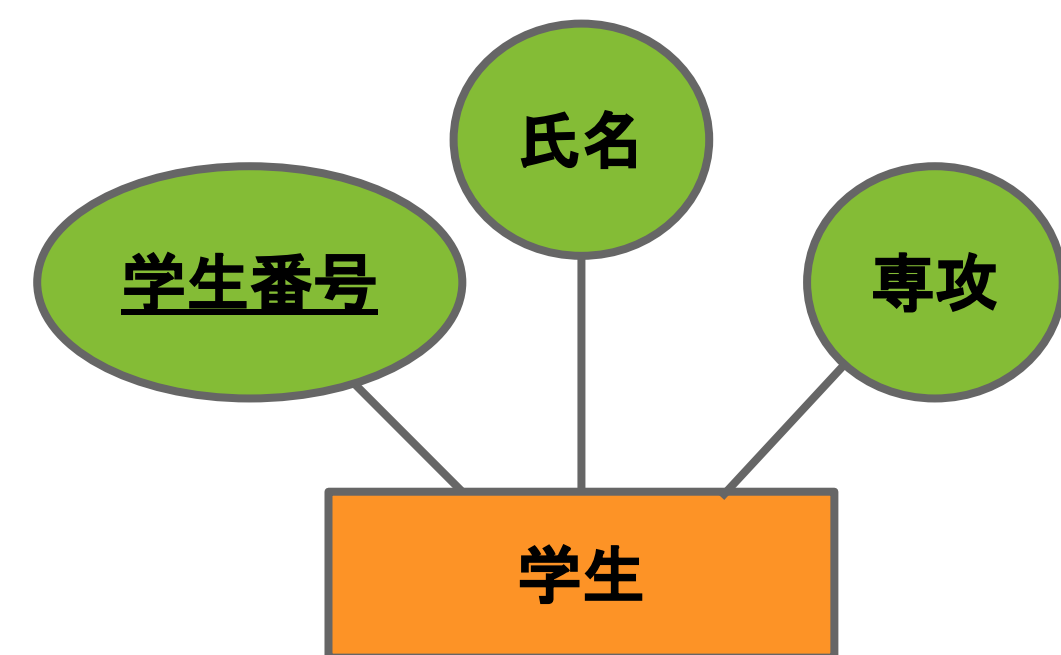
---

## 手順

- ER図からリレーショナルモデルを使ってスキーマへ変換
- スキーマの正規化

# 実体集合の変換

- 実体集合からテーブルへの変換

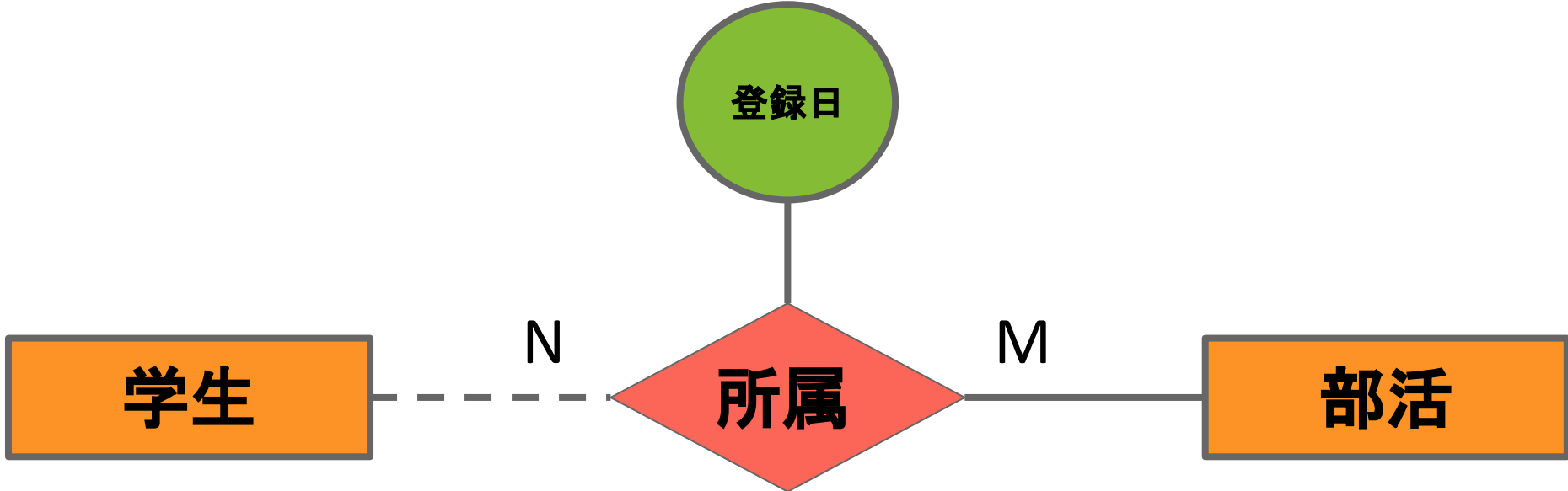


```
CREATE TABLE student(  
  sid INTEGER,  
  name CHAR(20),  
  major CHAR(20),  
  PRIMARY KEY (sid));
```

<u>sid</u>	name	major
1	佐藤	文化人類学
2	山田	計算機科学
3	金子	経済学

# 多対多関連集合の変換

- 例えば、学生は**複数**の部活に登録できるし、部活には**複数**の学生が参加できる
- 多対多関連集合はテーブルに変換
- テーブルに含めるもの
  - 関連集合の全ての属性を追加
  - 関連集合に参加する各実体集合のキーを**外部キー**として設定
  - 外部キーのセットを**複合主キー**として設定

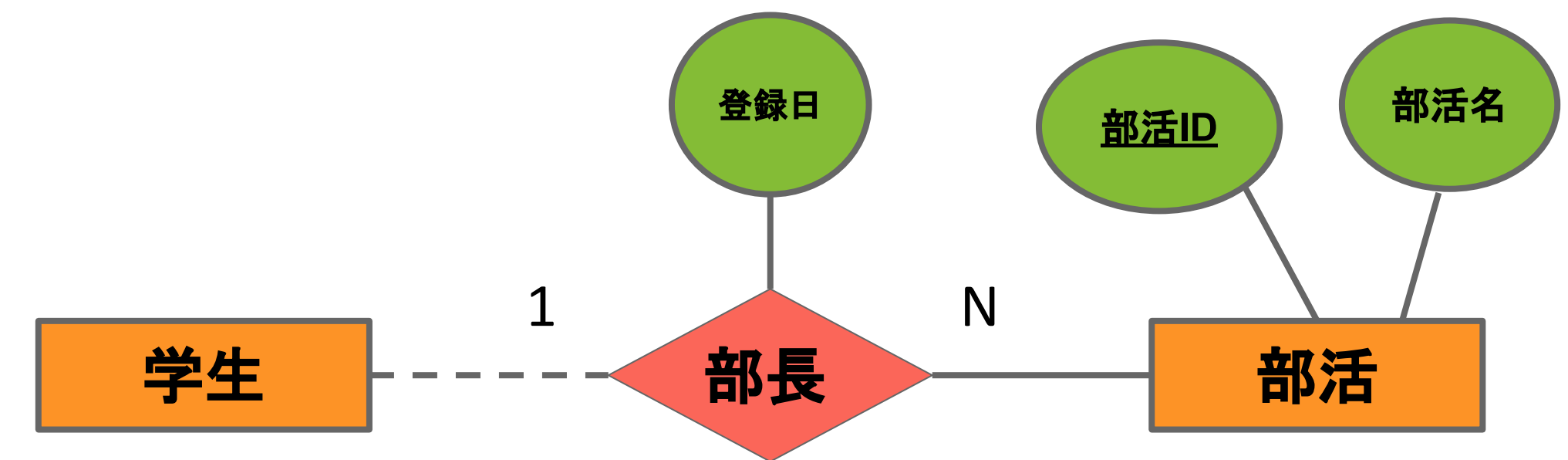


```
CREATE TABLE club_member (  
  sid INTEGER,  
  cid INTEGER,  
  joined_at DATE,  
  PRIMARY KEY (cid, sid),  
  FOREIGN KEY (sid) REFERENCES student(sid),  
  FOREIGN KEY (cid) REFERENCES club(cid));
```

<u>sid</u>	<u>cid</u>	joined_at
1	1	2021/4/15
1	2	2021/4/15
2	3	2021/5/1

# 1対多関連集合の変換

- 例: 学生は複数の部長を兼任できるが、各部活の部長は**最大1人**まで
- 2通りのやり方
  - 関連集合をテーブルに変換
    - **主キー**は部活のキーにし、それ以外は多対多と同じように作る
  - 実体集合のテーブルに関連集合を統合
    - 部活テーブルに部長を統合
    - 関連の属性の追加 + 学生を参照する**外部キー**を設定



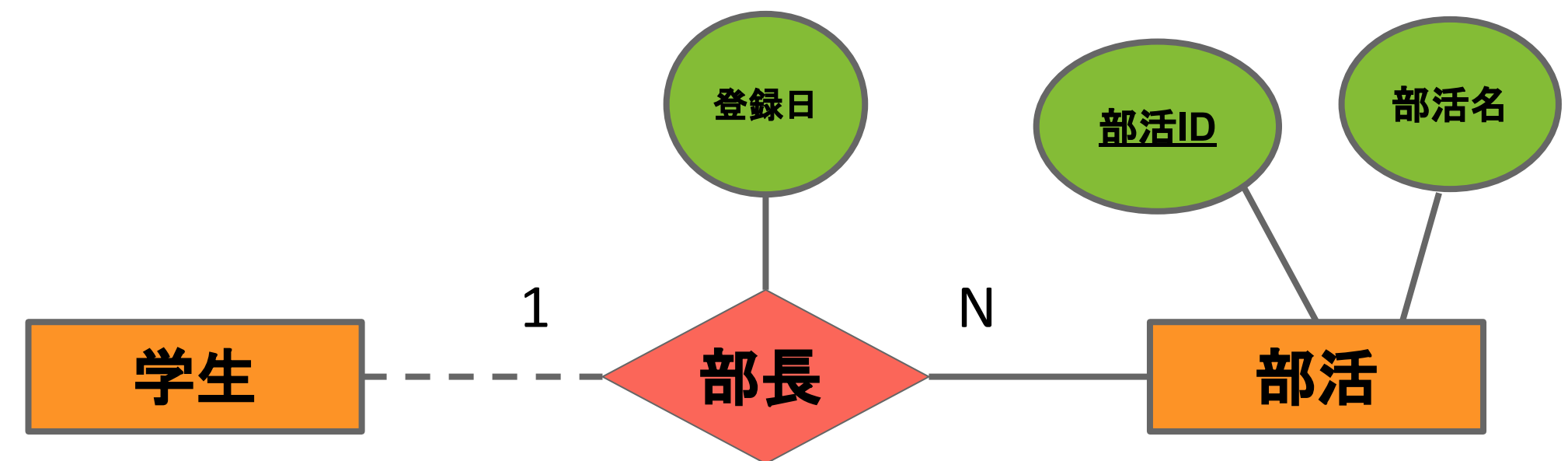
```
CREATE TABLE club_director (  
  sid INTEGER,  
  cid INTEGER,  
  joined_at DATE,  
  PRIMARY KEY (cid),  
  FOREIGN KEY (sid) REFERENCES student,  
  FOREIGN KEY (cid) REFERENCES club);
```

```
CREATE TABLE club (  
  cid INTEGER,  
  name CHAR(20),  
  director_id INTEGER,  
  director_since DATE,  
  PRIMARY KEY (cid),  
  FOREIGN KEY (director_id) REFERENCES student(sid));
```



# 全面的参加制約の変換

- 例: 各部活には必ず**最低1人**の部長がいる
- **NOT NULL制約**やCHECK制約を利用



部長と部活を統合するやり方で作ったテーブル

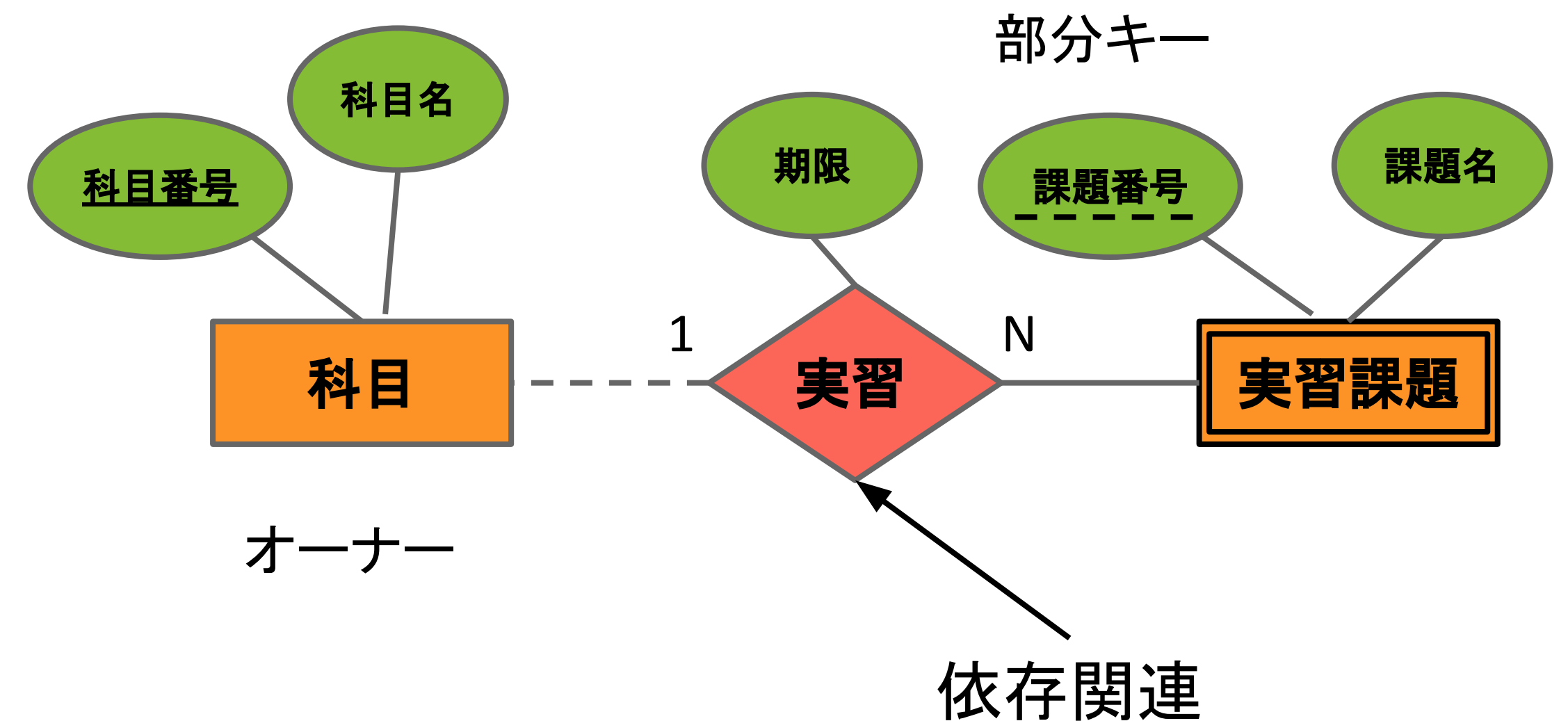
```
CREATE TABLE club (  
  cid INTEGER,  
  name CHAR(20),  
  director_id INTEGER NOT NULL,  
  director_since DATE,  
  PRIMARY KEY (cid),  
  FOREIGN KEY (director_id) REFERENCES student);
```



# 弱実体集合の変換

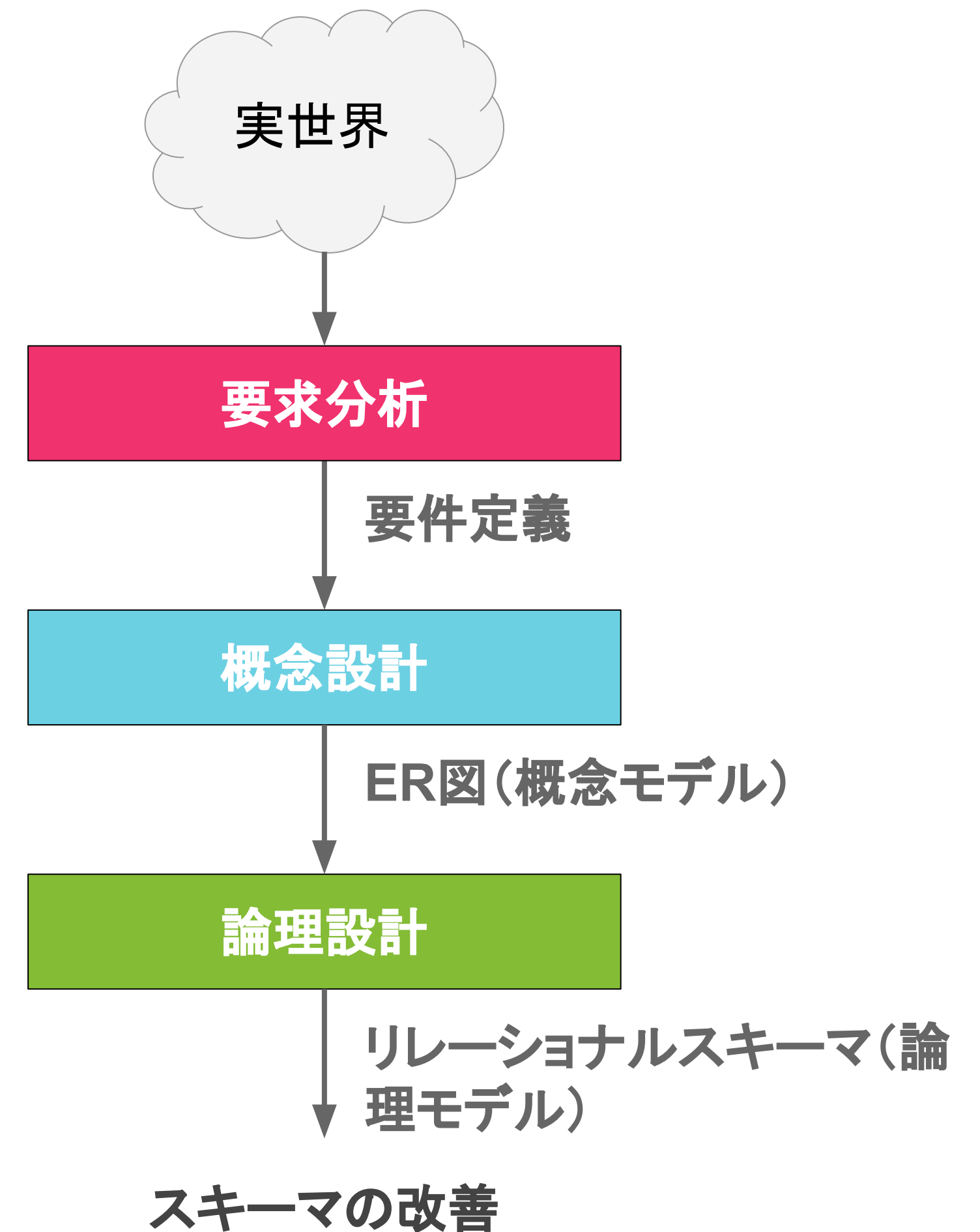
- 弱実体集合と依存関連集合を単一のテーブルに変換
  - 主キーは、オーナーの主キー＋部分キー
  - オーナーの削除時に、全ての弱実体集合を削除するために ON DELETE CASCADE

```
CREATE TABLE exercise (  
  eid INTEGER,  
  name CHAR(20),  
  deadline DATE,  
  PRIMARY KEY (eid, subject_id),  
  FOREIGN KEY (subject_id) REFERENCES subject  
  ON DELETE CASCADE);
```



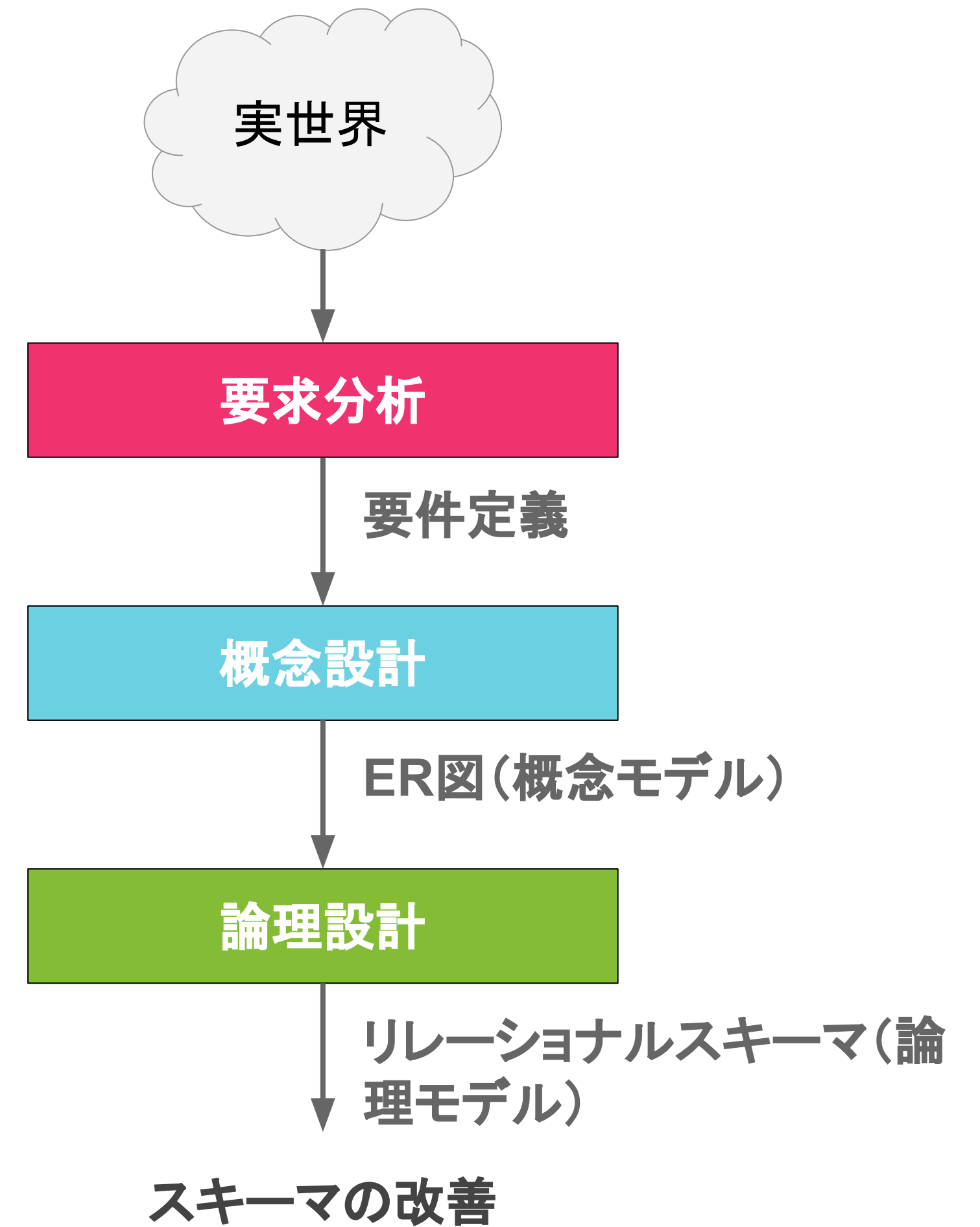
# ここまでの要点

- 概念設計
  - 要件定義を元に、管理したい対象の実世界をモデリング
- ERモデル
  - 構造: 実体、関連、属性、その他(弱実体)
  - 制約: キー制約、参加制約
    - 外部キー制約は関連の定義に暗黙的に存在
- 論理設計
  - 概念モデルをDBのデータモデルでスキーマに変換
  - 次は、スキーマの改善(その前に、なぜするのか)



# ERモデルの問題

- ERモデルはシンプルだけど、**表現できない制約**がまだまだある
  - 特に、関数従属性など
- ER図は**主観的**
  - 与えられたシナリオに対して様々な図が書けてしまう
- 次の正規化は、数学的な視点でERモデルが表現しきれなかった制約と向き合っていくプロセス



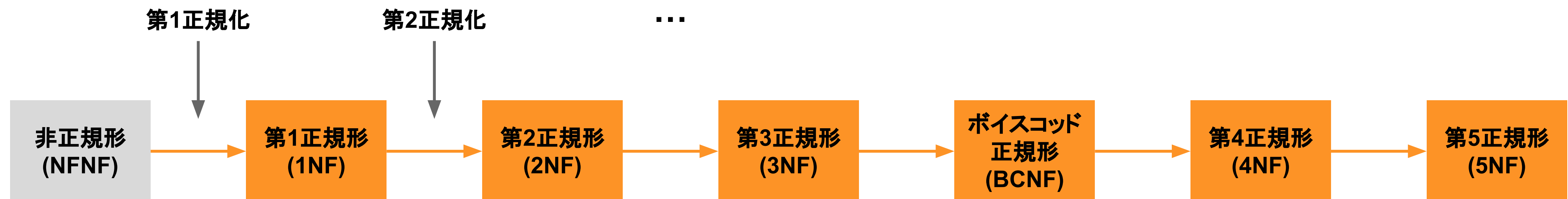
# 正規化の目的とは

---

- 目的:リレーションから**データの冗長性**を排除すること
- 冗長性:必要最低限のものに加えて、事実やデータの重複がある状態
- 冗長性の問題
  - スペースの無駄遣い
  - **更新時の異常**
    - 異常:データに論理的な不整合がある状態(事実が食い違った状態)
- 要は、データの冗長性を排除して、データ不整合(更新時異常)を未然に防ぎたい！

# 正規形 (Normal Form, NF)

- 正規化とは、リレーションからデータの冗長性を排除していく一連のプロセス
- 複数の進行段階
  - 各段階でその段階特有の冗長性を排除していくようなイメージ
  - 各ステップ完了後に得られる形式が正規形 (NF)
    - 例: 第1正規化されたリレーションは、第一正規形として定義された条件を満たす
  - 段階が上がるとデータの”純度”が増す
    - i.e., 前の段階の条件も満たす
    - 例: BCNFのリレーションは1NFから3NFまで全て満たす



# 第一正規形

- 1NFの条件
  - 全ての属性が”アトミック”(単一値)であること
    - 属性値はそのドメイン中の要素の1つであること
    - i.e., リレーションであること
      - NULLや重複があっても正規化はできる
- 1NFを満たさない例

属性値に複数の値が含まれる

学籍番号	氏名	履修科目
1	佐藤	Linux, データベース
2	山田	Linux, Go言語
3	金子	Go言語, kubernetes

複数の属性が概念的に同じ(同じ事実を表せる)

学籍番号	氏名	履修科目1	履修科目2
1	佐藤	Linux	データベース
2	山田	Go言語	Linux
3	金子	kubernetes	Go言語

補足: 上記の例は、SQLアンチパターン本で、それぞれジェイウォーク、マルチ  
カラムアトリビュートとして紹介されている

# 第一正規形

- 正規化の方法
  - ”繰り返し項目”を複数の行に分割
  - この例の場合、二通りのやり方が考えられる

同一テーブル内に展開

学籍番号	氏名	履修科目
1	佐藤	Linux
1	佐藤	データベース
2	山田	Go言語
..	..	..

※ 主キーが変わる

別テーブルに分解

学籍番号	履修科目
1	Linux
1	データベース
2	Go言語
..	..



# 第二正規形

---

- 2NFの条件
  - 1NFであること
  - 全ての非キー属性がキーに完全関数従属していること
    - i.e., すべての**部分関数従属性が排除**された状態

# 関数従属性

---

- 関数従属性はリレーションに存在しうる一種の制約
- $X \rightarrow Y$  と記述できる
  - 意味: **Xの値が決まれば、Yの値が一つに定まる**
  - 読み方: "YがXに関数従属している"、"XがYを関数的に決定する"

X	Y	Z
2	4	1
2	?	2
3	4	1

# 関数従属性

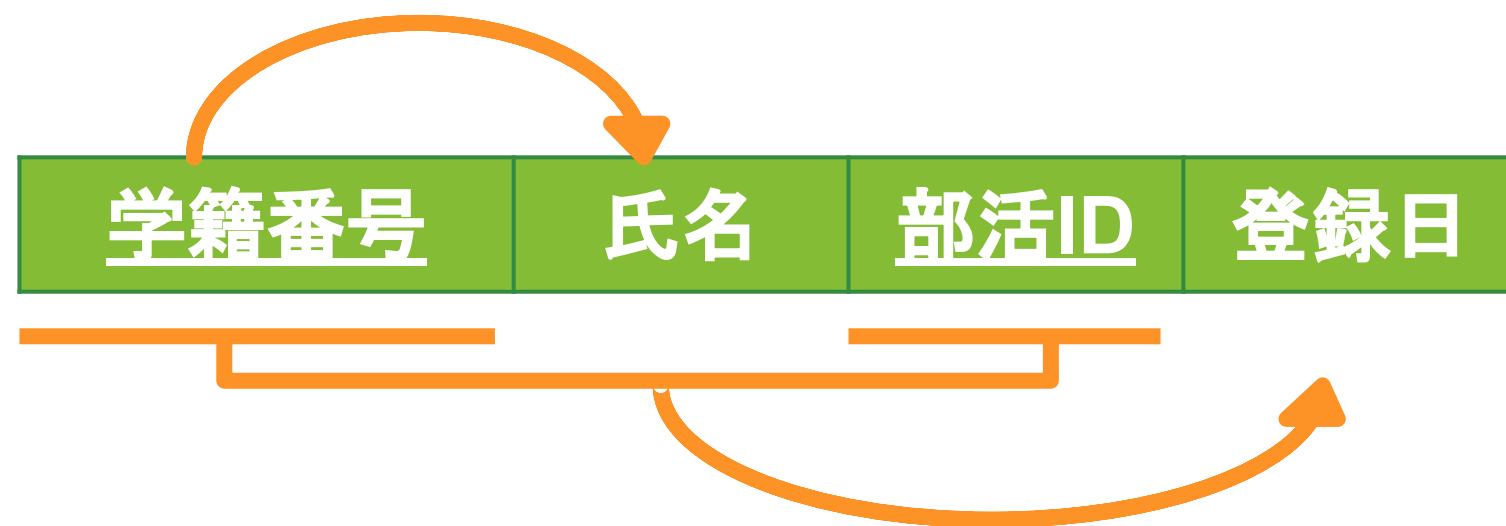
---

- Xが複数列の場合
  - 例:  $\{A, B\} \rightarrow C$
  - "Cは{A,B}に関数従属している"
  - Cの値はAとBの値の組み合わせで、1つに定まる
- Yが複数列の場合
  - 例: 主キー
  - $PK \rightarrow \{A, B, \dots, \}$
  - AはPKに..., BはPKに..., 関数従属している
  - (すべての属性が主キーに関数従属している)

# 関数従属性は何が嬉しいのか

---

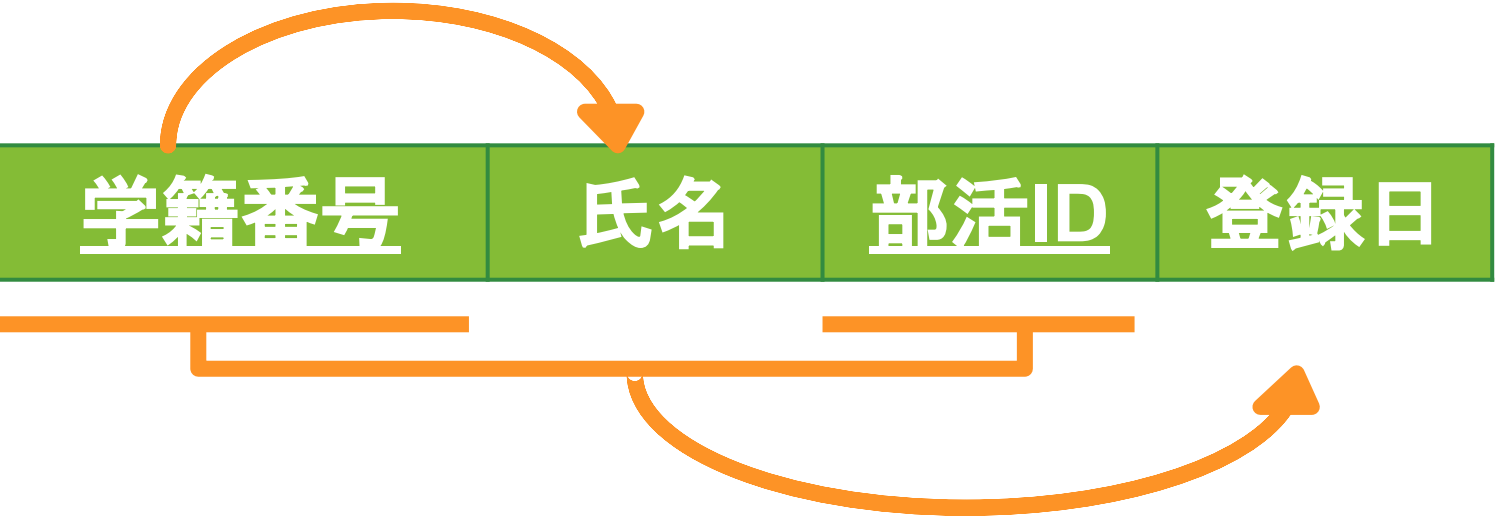
- 冗長性を発見する手助けをしてくれる
- 例: 以下のテーブルには二つの関数従属が存在する
  - {学籍番号, 部活ID} -> 登録日
  - 学籍番号 -> 氏名



Q. この2つの関数従属を見た上で、冗長性の観点から言えるこのテーブルの問題とは何か

# 関数従属性は何が嬉しいのか

- 冗長性を発見する手助けをしてくれる
- 例: 以下のテーブルには二つの関数従属が存在する
  - {学籍番号, 部活ID} -> 登録日
  - 学籍番号 -> 氏名



Q. この2つの関数従属を見た上で、冗長性の観点から言えるこのテーブルの問題とは何か

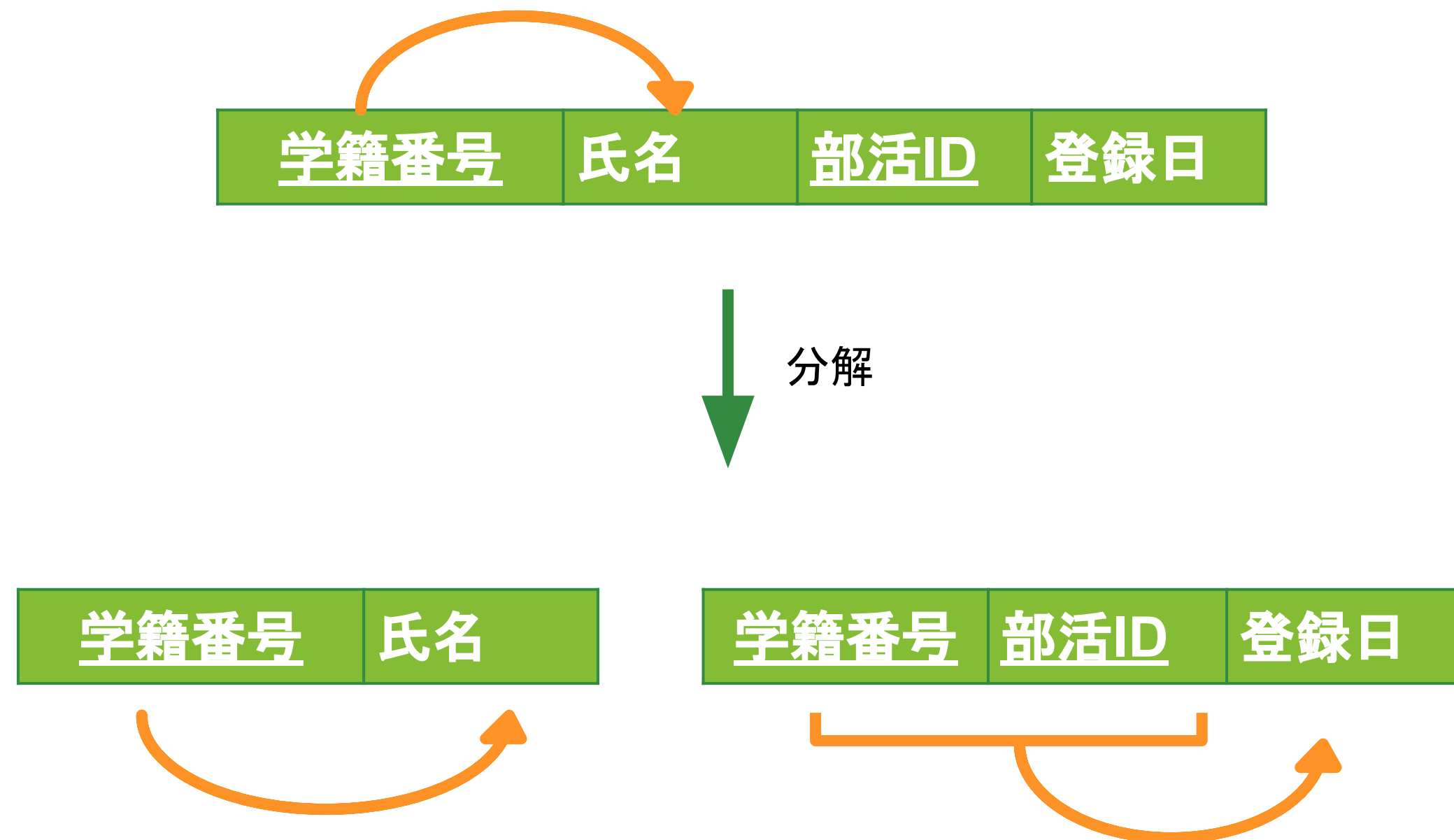
学籍番号	氏名	部活ID	登録日
S1111	佐藤	1	4/12
S1111	佐藤	2	5/12
S2222	山田	3	5/12

A. 学籍番号と氏名のペアが複数存在できてしまう！学籍番号と部活IDのペアはキー制約によって一度しか格納されないので問題なし！

# テーブルの分解

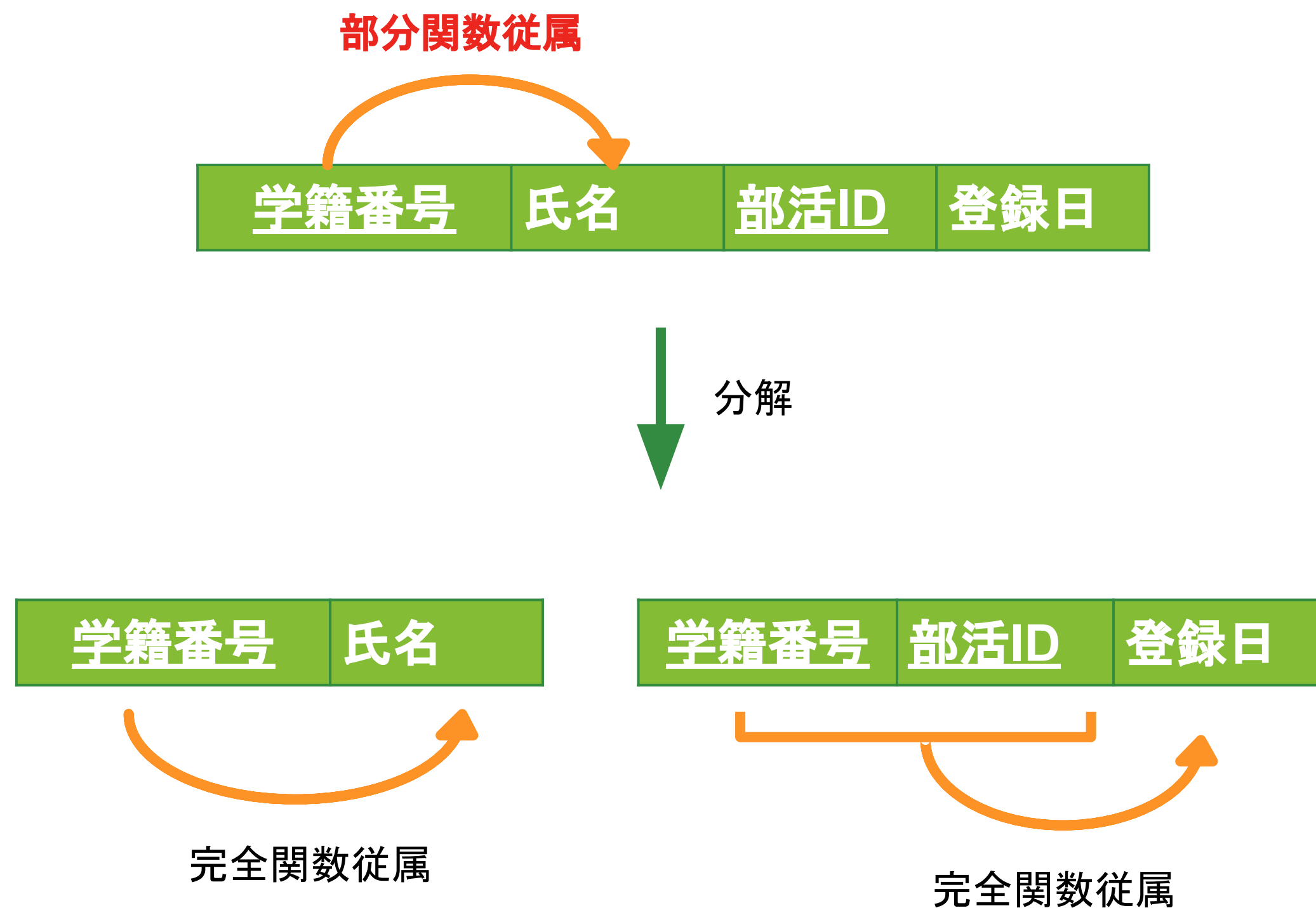
---

- この冗長性もテーブルを分解することで解決できる！



# テーブルの分解

- この冗長性もテーブルを分解することで解決できる！



実は今排除したのが**部分関数従属性**！

-> 分解されたテーブル達は2NF



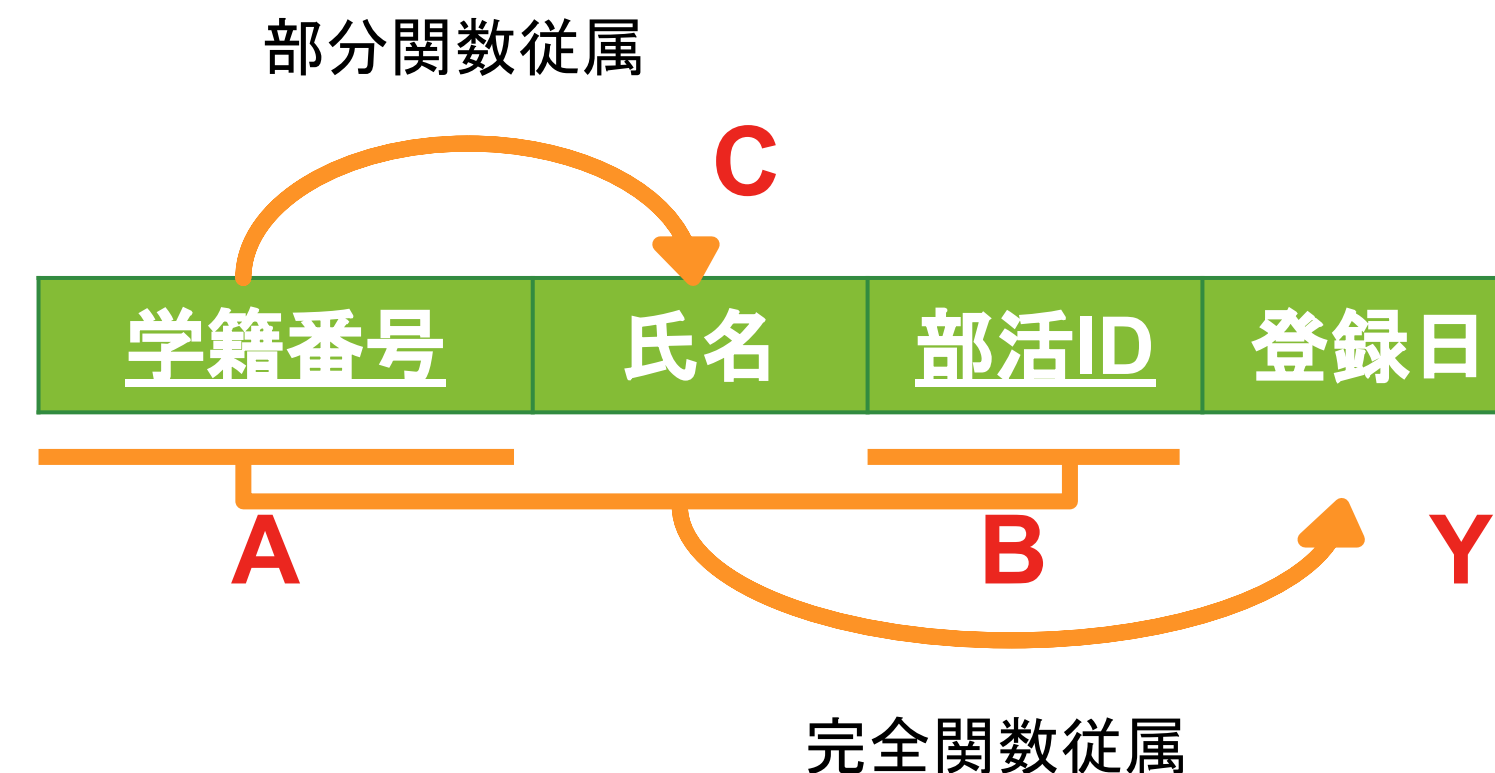
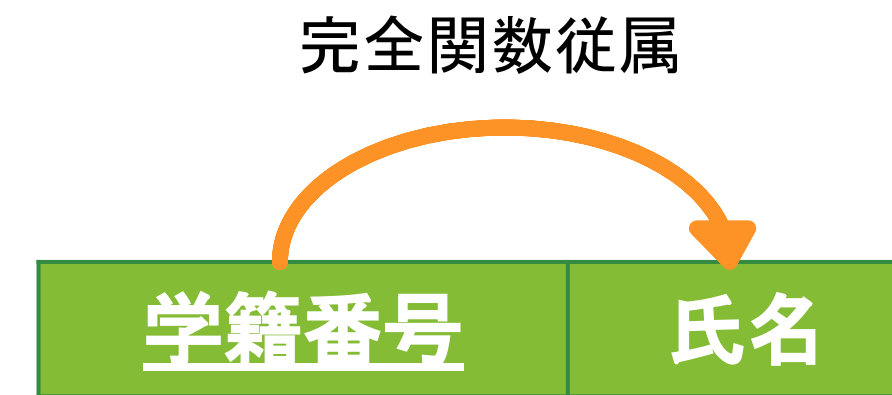
# 関数従属性の種類

## 完全関数従属性

- $X \rightarrow Y$ の時
  - $Y$ は $X$ に完全関数従属している
- $\{A, B\} \rightarrow Y$ の時
  - $A \rightarrow Y$ も $B \rightarrow Y$ も成り立たない時、 $Y$ は $\{A, B\}$ に完全関数従属している

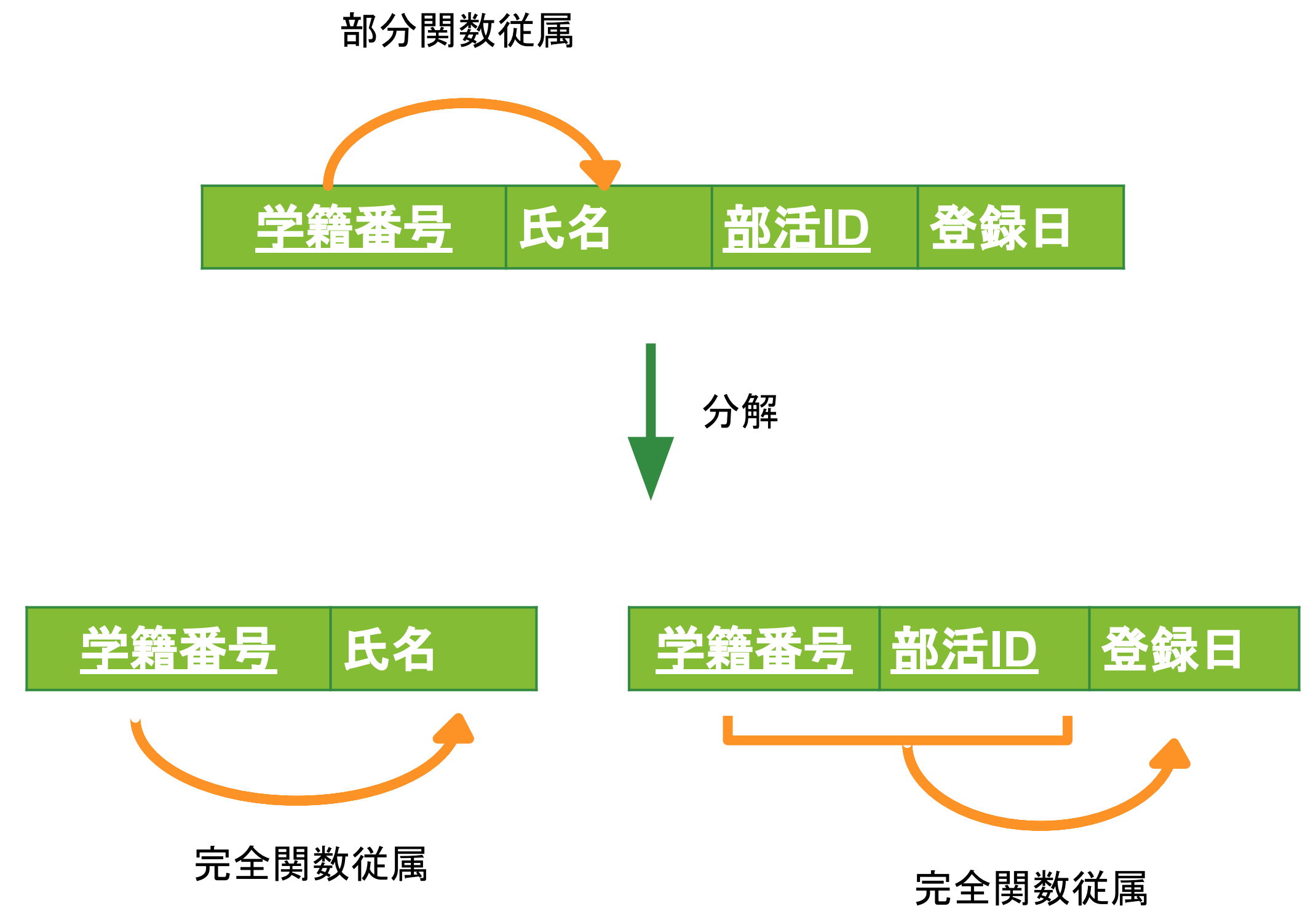
## 部分関数従属性

- $X \rightarrow Y$ の時
  - 起こり得ない
- $\{A, B\} \rightarrow Y$ が与えられた時
  - もし $A \rightarrow C$ か $B \rightarrow C$ のどちらかが成り立つなら、 $C$ は $\{A, B\}$ に部分関数従属している



# 第二正規形

- 2NFの条件
  - 1NFであること
  - 全ての非キー属性がキーに完全関数従属していること
    - i.e., 全ての**部分関数従属性が排除**された状態
    - (全てのキーが単一ならずで2NF)
- 正規化の方法
  - キーの一部によって一意に決まる非キー属性を別表に移す



# 第三正規形

---

- 3NFの条件
  - 2NFであること
  - 全ての非キー属性がキーに**推移関数従属**していないこと

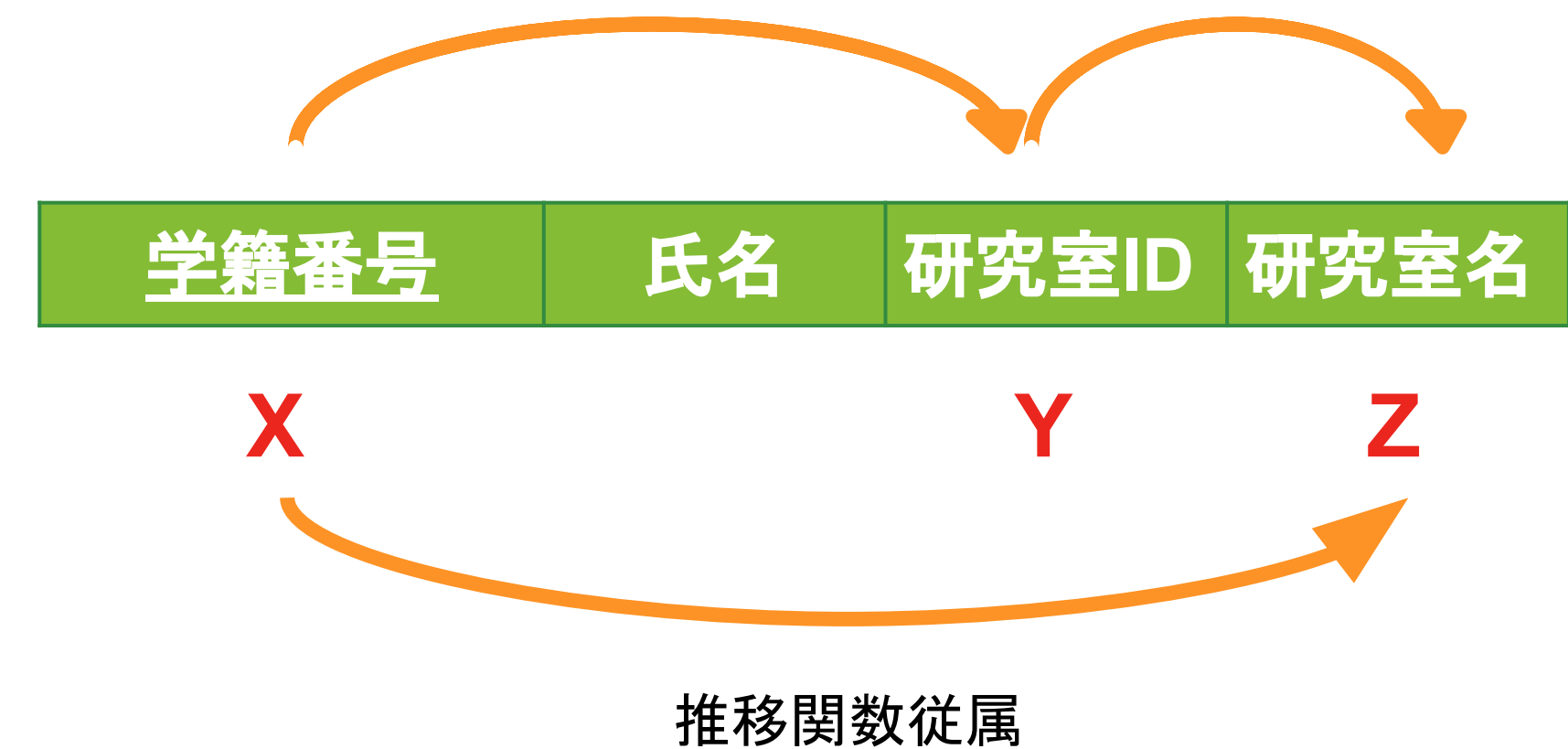
# 推移関数従属性

- やや形式的には

- キーXと非キーYが与えられ

$X \rightarrow Y$ かつ $Y \rightarrow Z$ なら、 $X \rightarrow Z$ が成り立つ時

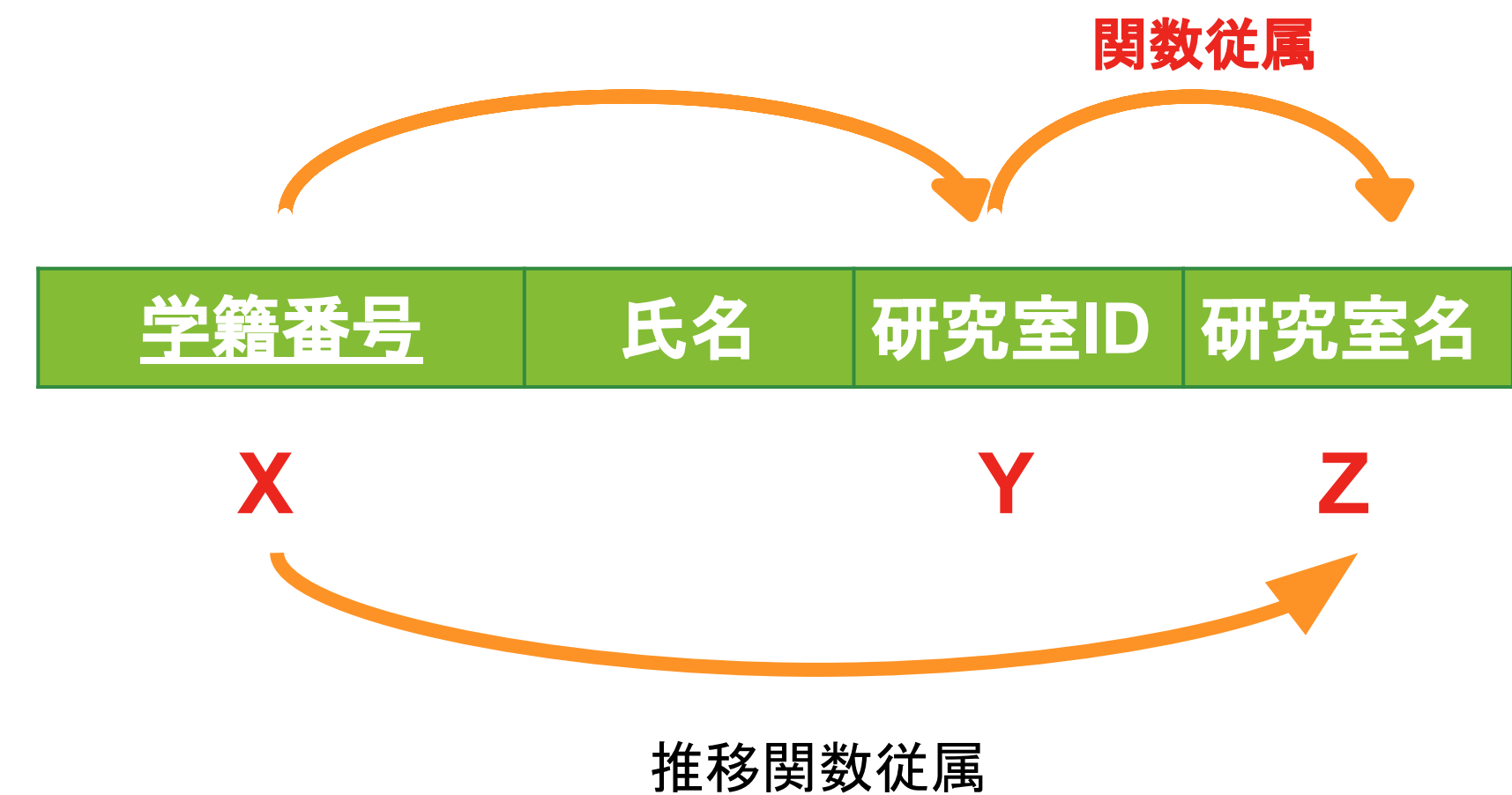
ZはXに推移関数従属していると表現



前提: 学生が参加できる研究室は**最大1つ**まで

# 推移関数従属性

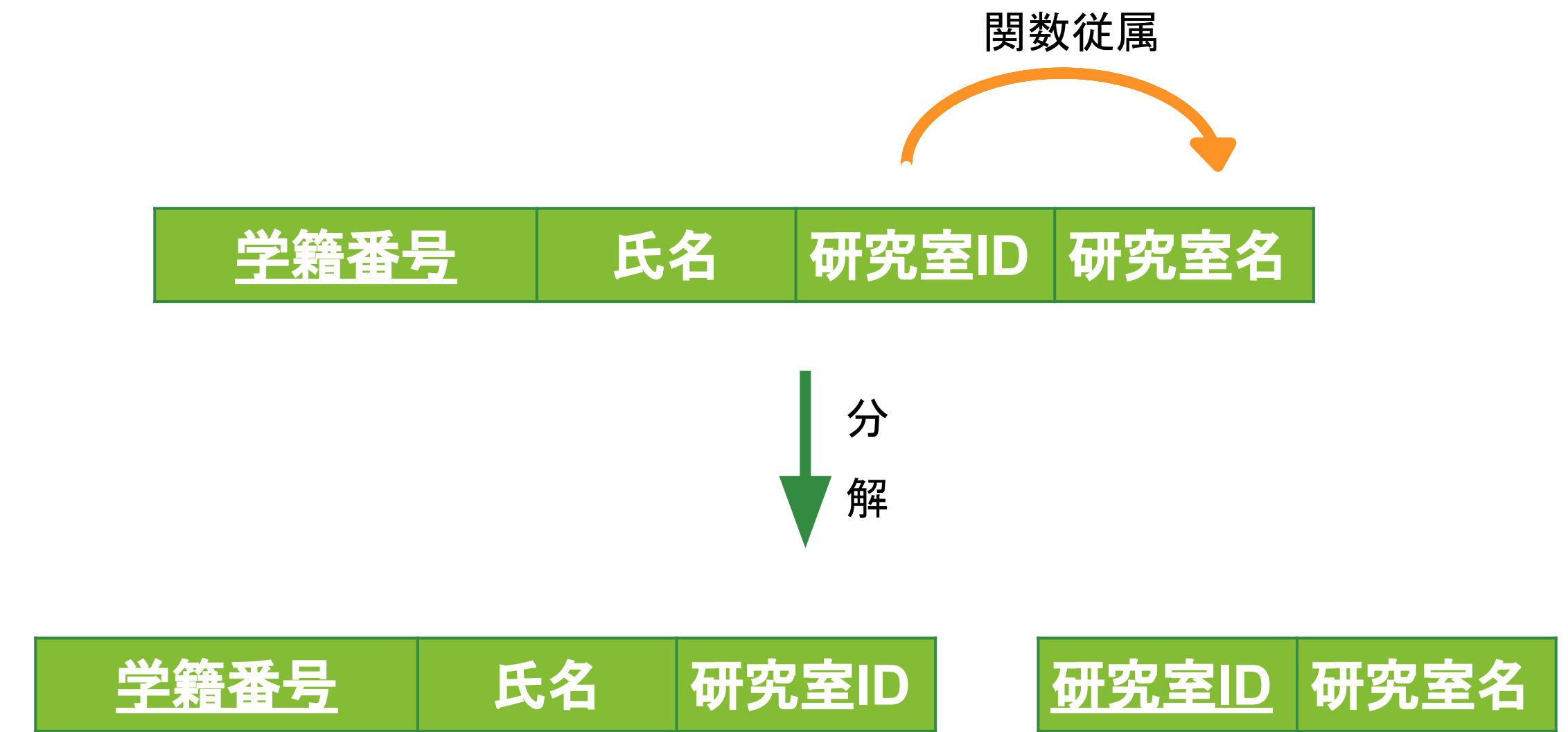
- やや形式的には
  - キーXと非キーYが与えられ  
 $X \rightarrow Y$ かつ $Y \rightarrow Z$ なら、 $X \rightarrow Z$ が成り立つ時  
ZはXに推移関数従属していると表現
- 要するに3NFでは、**非キー属性間の関数従属**を排除したい



前提: 学生が参加できる研究室は**最大1つ**まで

# 第三正規形

- 3NFの条件
  - 2NFであること
  - 全ての非キー属性がキーに**推移関数従属**していないこと
- 正規化の方法
  - 非キー属性によって一意に決まる非キー属性を別表に移す



# ボイスコード正規形

---

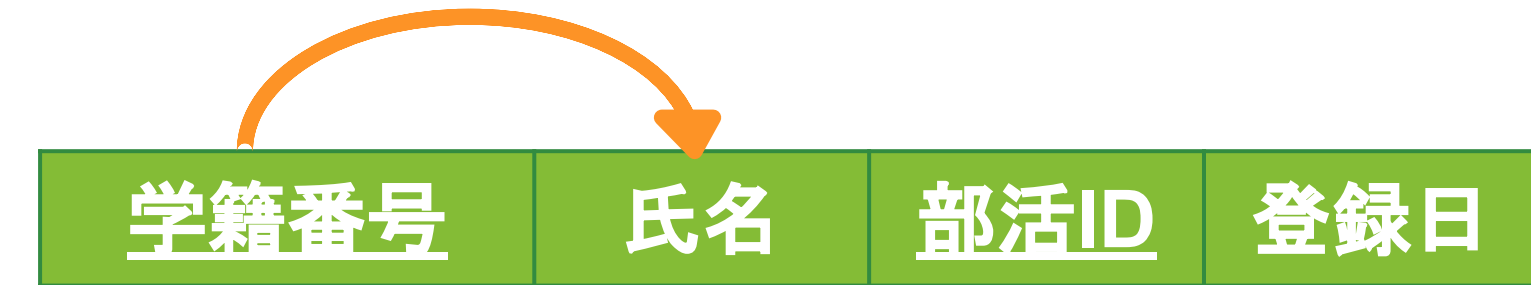
- BCNFの条件
  - 3NFであること
  - (自明ではない)関数従属性がすべて取り除かれた状態
- 自明ではないの説明は省略
- つまり、3NFに残る(自明ではない)関数従属性を取り除けば良い



# 3NFに残る関数従属性

- 今まで排除してきた(自明ではない)関数従属性
  - キーの一部 → 非キー属性
  - 非キー属性 → 非キー属性

2NFで排除: キーの一部 → 非キー属性



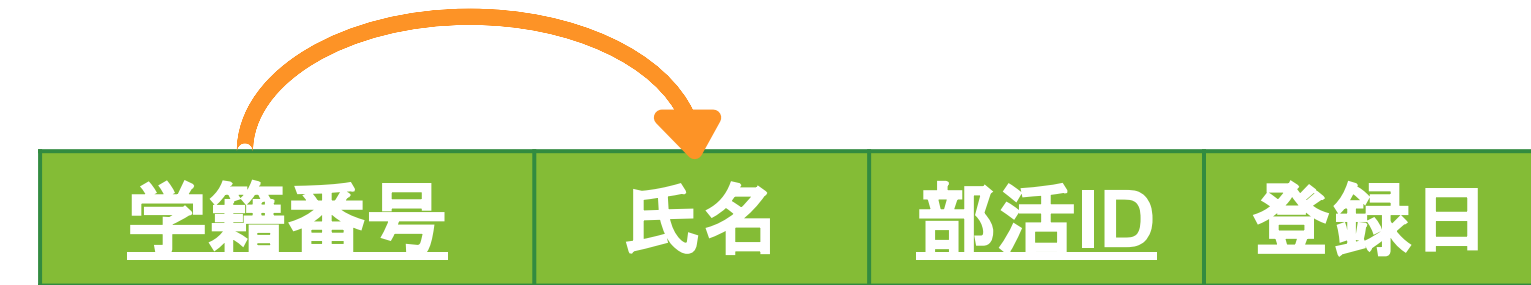
3NFで排除: 非キー属性 → 非キー属性



# 3NFに残る関数従属性

- 今まで排除してきた(自明ではない)関数従属性
  - キーの一部 → 非キー属性
  - 非キー属性 → 非キー属性
- 残るは
  - **非キー属性 → キーの一部**
  - **キーの一部 → キーの一部**

2NFで排除: キーの一部 → 非キー属性



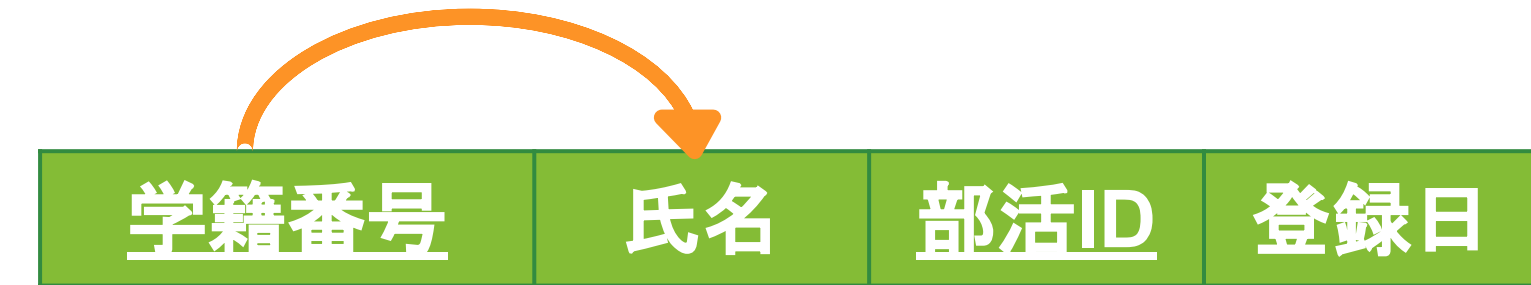
3NFで排除: 非キー属性 → 非キー属性



# 3NFに残る関数従属性

- 今まで排除してきた(自明ではない)関数従属性
  - キーの一部 → 非キー属性
  - 非キー属性 → 非キー属性
- 残るは
  - 非キー属性 → キーの一部
  - ~~キーの一部 → キーの一部~~

2NFで排除: キーの一部 → 非キー属性



3NFで排除: 非キー属性 → 非キー属性



# ボイスコッド正規形

- 正規化する方法
  - 非キー属性→キーの一部を排除すること
  - この関数従属関係を別表に移すこと

BNCFで排除: 非キー属性 → キーの一部



学籍番号	学科	研究室名
S1111	データベース	RDB
S1111	言語	Go言語
S2222	データベース	NoSQL

↓  
分解

学籍番号	学科
S1111	データベース
S1111	言語
S2222	データベース


学科	研究室名
データベース	RDB
言語	Go言語
データベース	NoSQL

前提: 学生は複数の研究室に参加できる

# ボイスコッド正規形

- 正規化する方法
  - 非キー属性→キーの一部を排除すること
  - この関数従属関係を別表に移すこと
- 注意点
  - 3NF -> BCNFへの分割では、関数従属性が保存されないことがある
    - 何かしらの情報が欠損する可能性がある

BCNFで排除: 非キー属性 → キーの一部



学籍番号	学科	研究室名
S1111	データベース	RDB
S1111	言語	Go言語
S2222	データベース	NoSQL

分解  
↓

学籍番号	学科
S1111	データベース
S1111	言語
S2222	データベース

学科	研究室名
データベース	RDB
言語	Go言語
データベース	NoSQL

前提: 学生は複数の研究室に参加できる

# ボイスコッド正規形


- 正規化する方法

- 非キー属性→キーの一部を排除すること
- この関数従属関係を別表に移すこと

- 注意点

- 3NF -> BCNFへの分割では、関数従属性が保存されないことがある
  - 何かしらの情報が欠損する可能性がある

BNCFで排除: 非キー属性 → キーの一部



学籍番号	学科	研究室名
S1111	データベース	RDB
S1111	言語	Go言語
S2222	データベース	NoSQL

分解  
↓

学籍番号	学科
S1111	データベース
S1111	言語
S2222	データベース

学科	研究室名
データベース	RDB
言語	Go言語
データベース	NoSQL

元々合った情報が欠損している

Q. 何が消えただろうか



# 情報欠損と確認方法

- この例で消えた情報(関数従属性)

- 分解前

- A. {学籍番号, 学科} -> 研究室名

- 研究室名 -> 学科

- 分解後

- 研究室名 -> 学科

- 情報欠損の確認方法

- 分解後のテーブルを結合したときに復元できない関数従属があるかどうか

BNCFで排除: 非キー属性 → キーの一部



学籍番号	学科	研究室名
S1111	データベース	RDB
S1111	言語	Go言語
S2222	データベース	NoSQL

↓ 分解

学籍番号	学科
S1111	データベース
S1111	言語
S2222	データベース

学科	研究室名
データベース	RDB
言語	Go言語
データベース	NoSQL

元々合った情報が欠損している

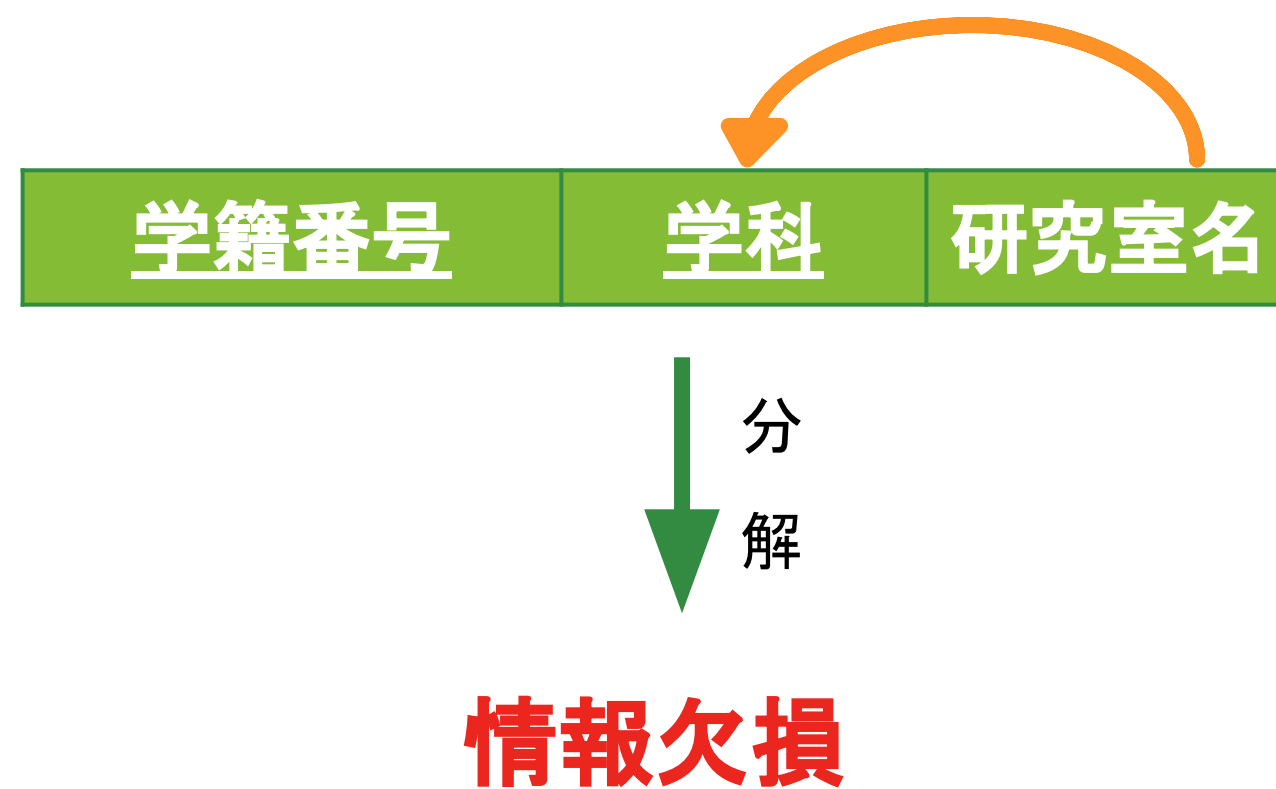
Q. 何が消えただろうか

# どうすればよいか

- そもそも、3NFだけどBCNFではない状態は、**複数の属性からなるキーが複数存在**する場合にのみ起きる可能性がある
- まずは、別のキーがないか考える

今までのキー: {学籍番号, 学科} → 研究室名

別のキー: {学籍番号, 研究室名} → 学科

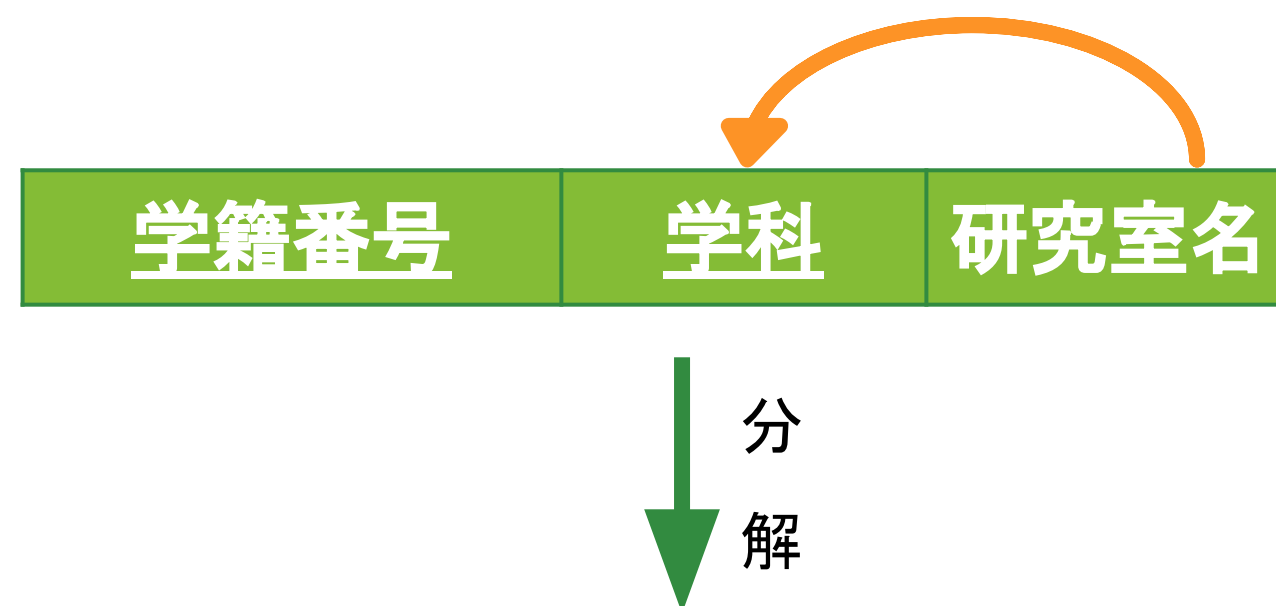


# どうすればよいか

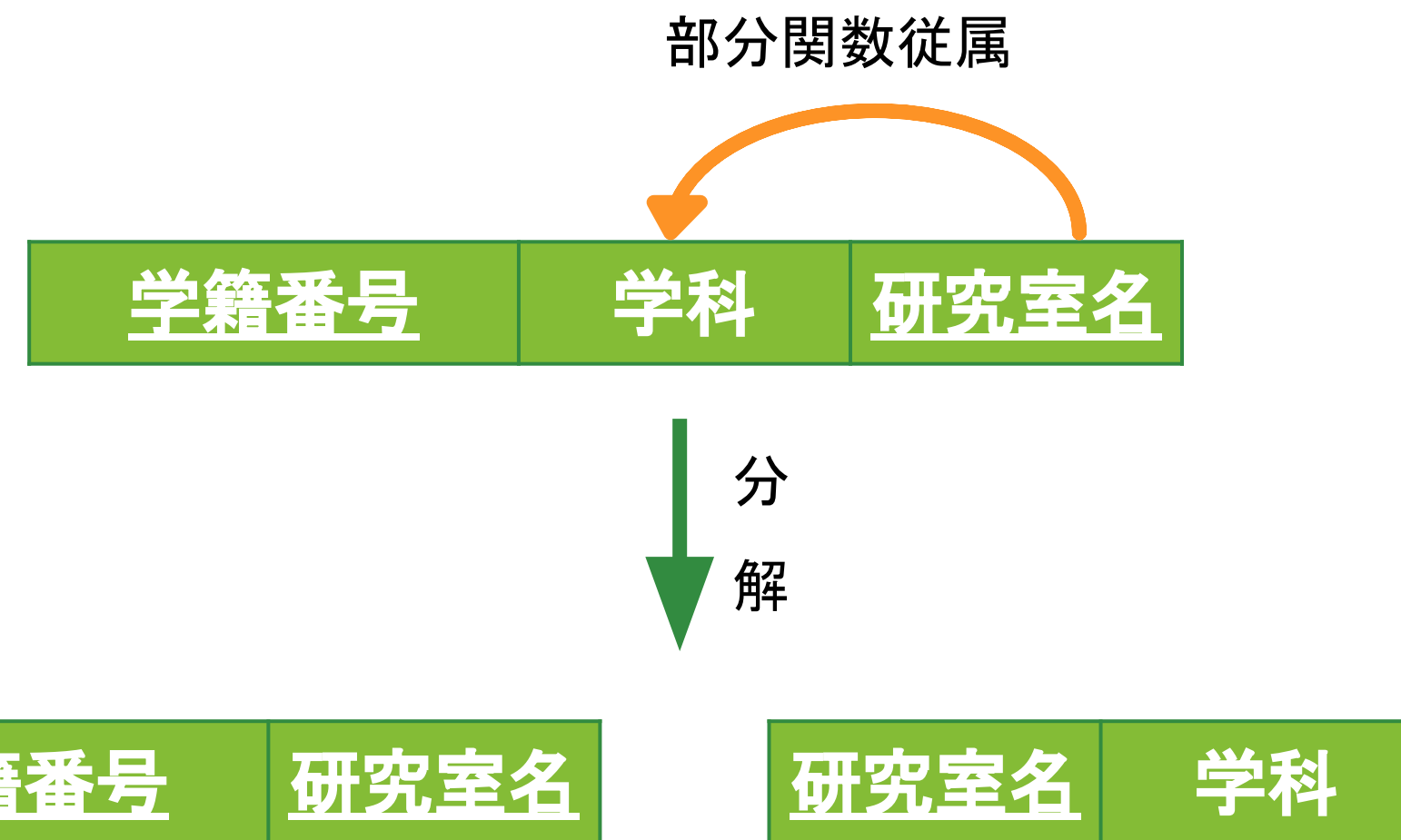
- そもそも、3NFだけどBCNFではない状態は、**複数の属性からなるキーが複数存在**する場合にのみ起きる可能性がある
- まずは、別のキーがないか考える

今までのキー: {学籍番号, 学科} → 研究室名

別のキー: {学籍番号, 研究室名} → 学科



情報欠損



情報欠損なし(情報無損失分解)

# ボイスコッド正規形の要点

---

- 正規化する方法
  - **非キー属性→キーの一部を排除すること**
  - この関数従属関係を別表に移すこと
- 注意点
  - 3NF -> BCNFへの分割では、**関数従属性が保存されないことがある**
    - 何かしらの情報が欠損する可能性がある
  - **なので、別のキーがあるか確認してみよう！**

# ボイスコッド正規形の要点

---

- 正規化する方法
  - **非キー属性→キーの一部を排除すること**
  - この関数従属関係を別表に移すこと
- 注意点
  - 3NF -> BCNFへの分割では、**関数従属性が保存されないことがある**
    - 何かしらの情報が欠損する可能性がある
  - **なので、別のキーがあるか確認してみよう！**
  - **ただし、必ずしも全ての関数従属性を保存できるような(情報無損失分解できるような)分解方法があるとは限らない.....**

# BCNF以降

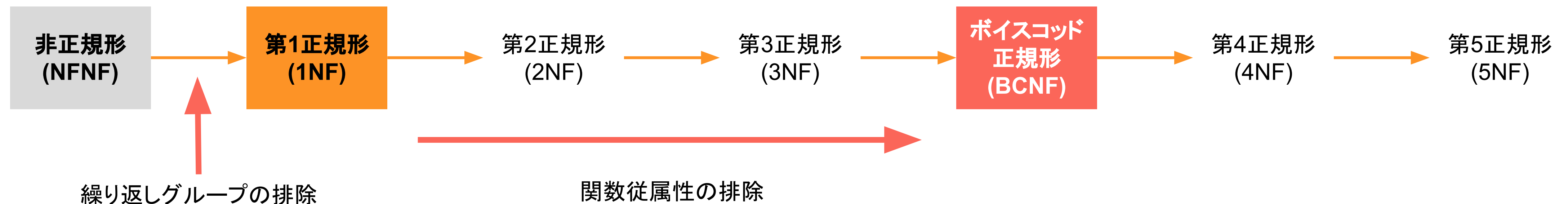
---

- BCNFまで到達すると
  - **関数従属性**を利用した分解はこれ以上できなくなる
- 4NF以降は
  - **結合従属性**を排除する戦い
  - BCNFを満たすと、自動的に5NFを満たす場合が多く、実践的にはBCNFまで理解していれば十分



# スキーマ改善の要点

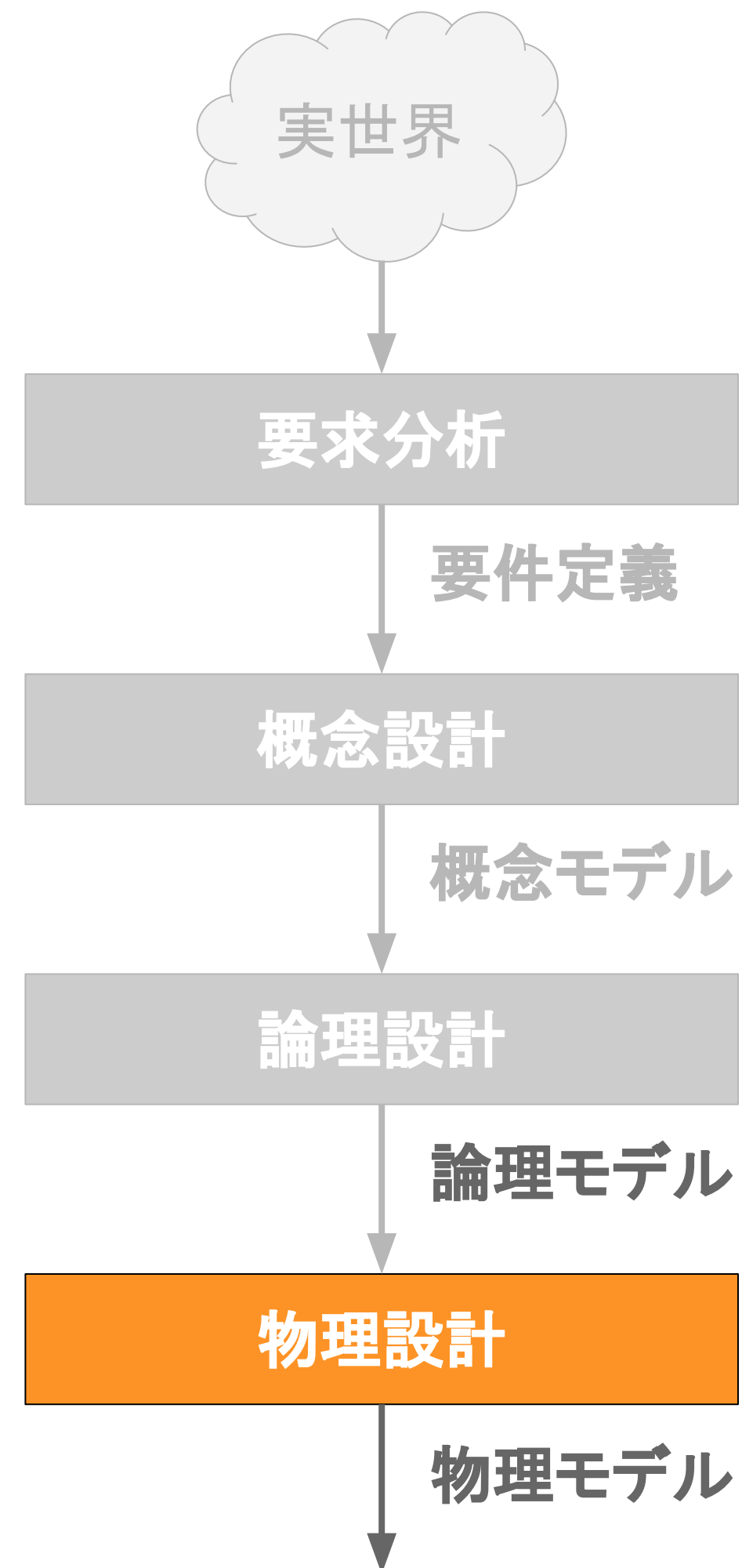
- 正規化の目的は、冗長性を排除し、更新時異常を起こりにくくすること
- 様々な正規形を見たが、とりあえず、BCNFまでが大事
- とりあえず、BCNFまで分解してみよう！
  - ある属性値が関数従属性によって推測できるなら、それはBCNFではない
- 3NFまでは必ず情報無損失な分解方法が存在するが、BCNFは違う
- 正規化はヒューリスティック!





# 残るは物理設計

---



5

# インデックス



# インデックスの概要

---

- インデックスとは、検索を高速化するために使われるもの
- インデックスの設計は物理設計の一部
  - 性能チューニングにおける最も主流な方法
  - アプリケーションの変更なしで設計可能
  - クエリの結果からも独立

# データアクセス方法

---

- データアクセス方法
  - DBMSがSQLで指定されたデータにアクセスする方法
  - アクセス方法の選択はDBMSが自動で行う
- 主に2種類
  - シーケンシャルスキャン(フルスキャン)
    - 対象テーブルの全データにアクセスして、条件を満たす行を1行ずつチェック
  - インデックススキャン
    - インデックスを利用したアクセス方法
    - 本の索引のように、条件に合う行の場所を特定してから、テーブルにアクセス
- 要するに、インデックスの目的は**フルスキャンを避ける**こととも言える

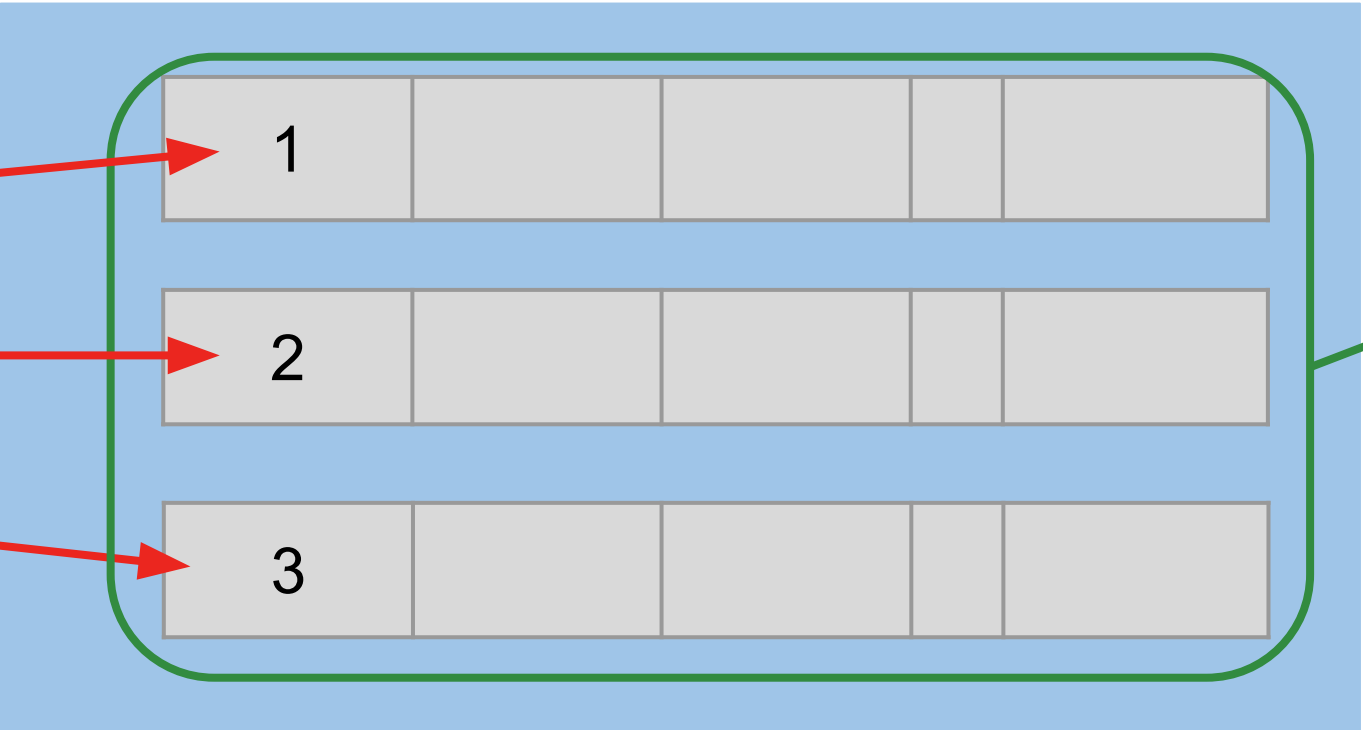
# インデックスとは

- インデックスとは
  - レコードのフィールド値とそのレコードの格納番地の対応表
    - i.e., (フィールド値、格納番地へのポインタ)のタプルの集合
  - レコードの探索機能を提供するデータ構造
  - 原理的には、バイナリファイル
- 一般に、1テーブルに複数のインデックスが定義可能、複合インデックスも貼れる

インデックス

フィールド値	格納番地
1	
2	
3	

データベース

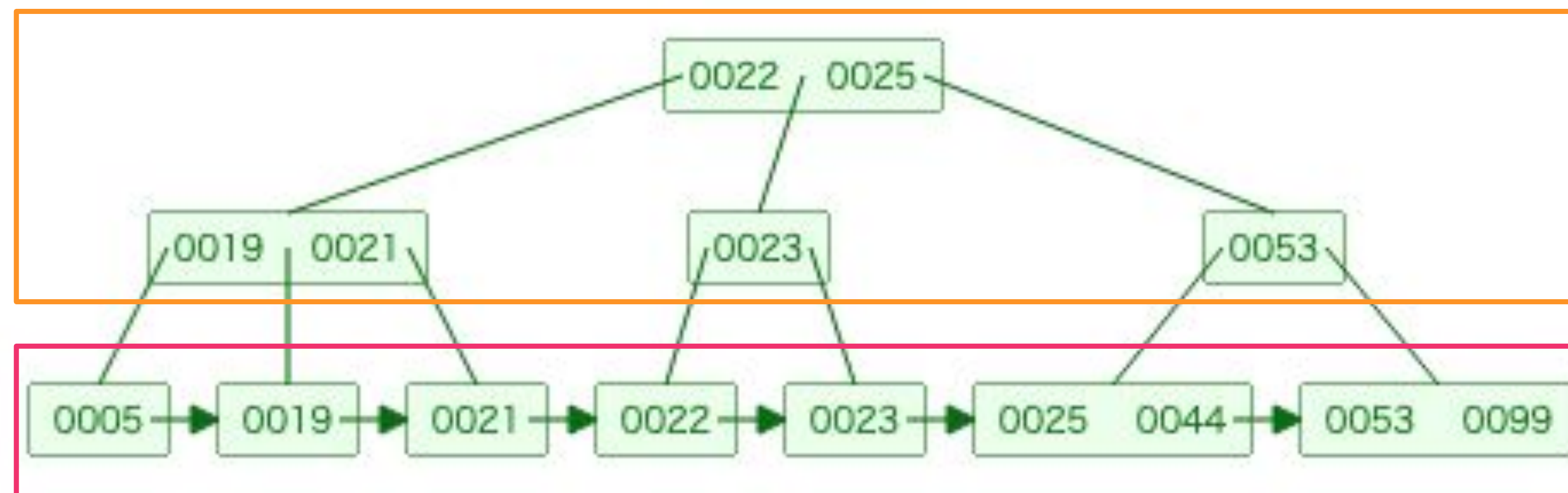


テーブル

<u>sid</u>	name	major
1	佐藤	文化人類学
2	山田	計算機科学
3	金子	経済学

# B+Tree

- RDBの代表的なインデックス
- 多分岐の平衡木(バランス木)
- データは常にソート済み
- 二段構成
  - データ部: データを格納したリーフノードから構成される
  - 索引部: ノンリーフノードから構成され、データ部への経路としての役割を担う

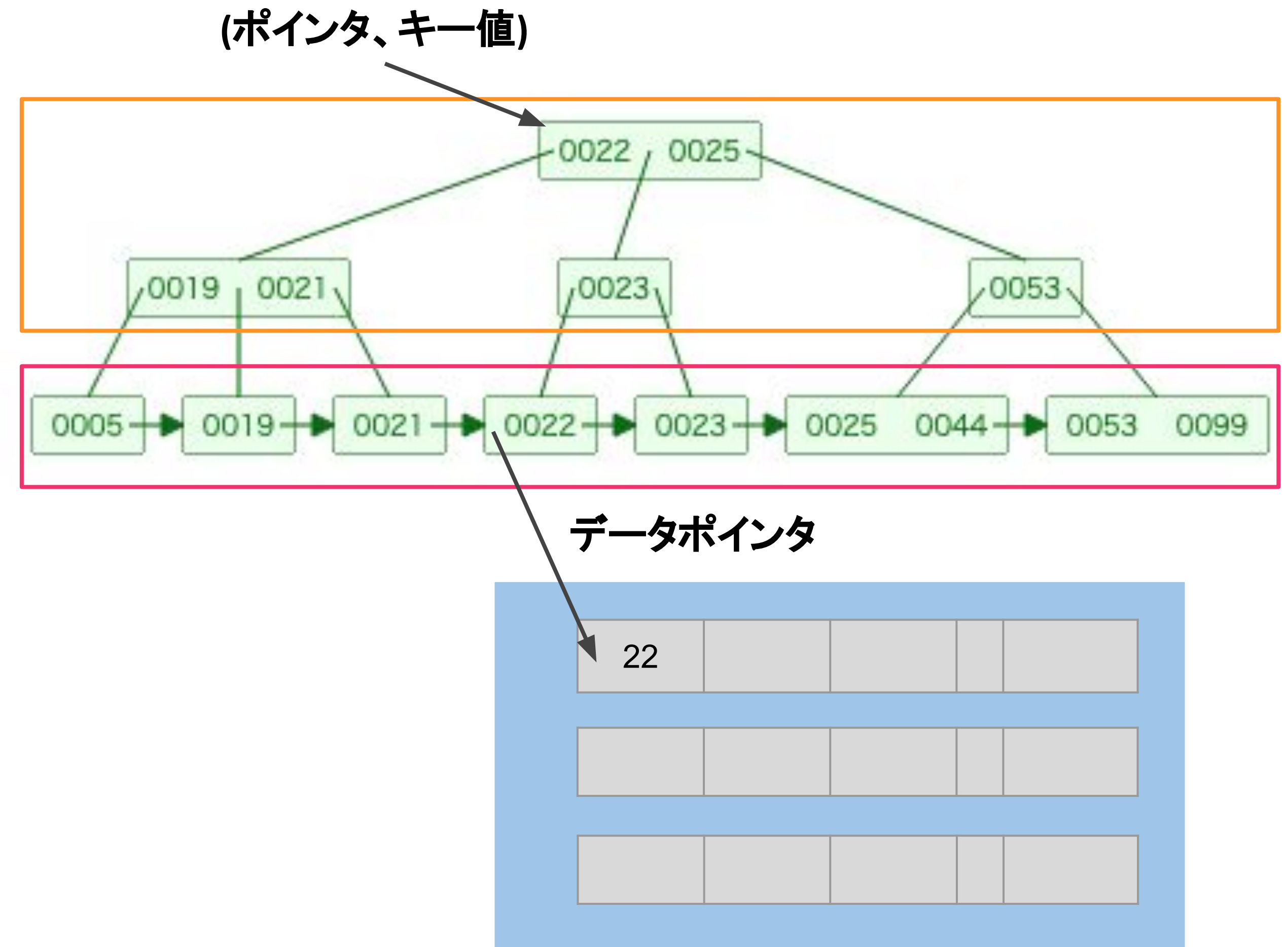


索引部

データ部

# B+Tree 構成

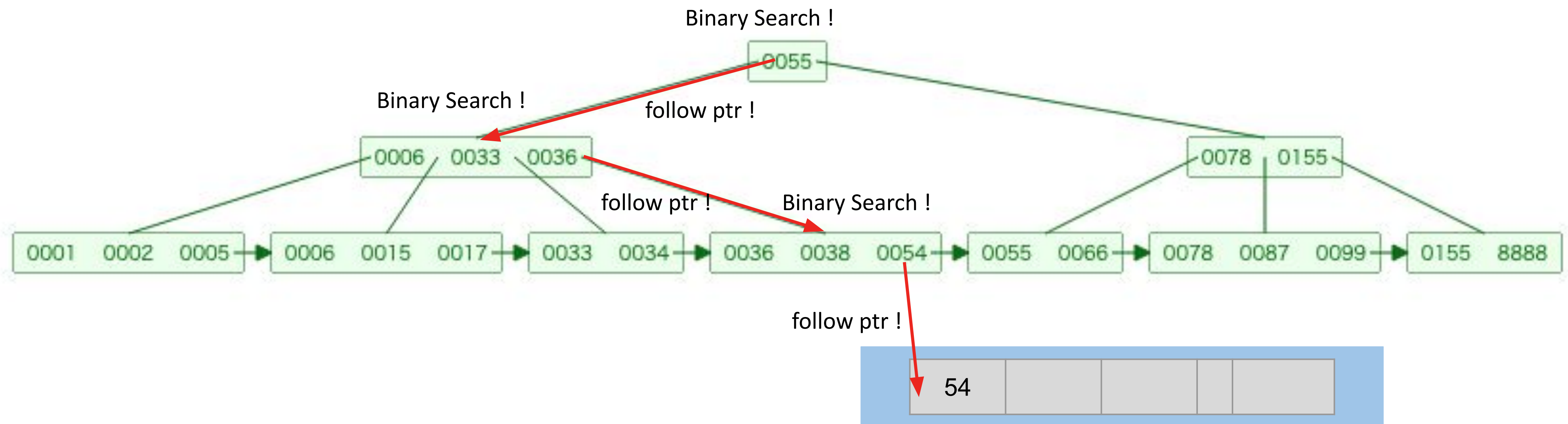
- 索引部
  - ルートノードと中間ノードから構成される
  - 各ノードは(ポインタ、キー値)の配列
- データ部
  - リーフノードから構成される
  - リーフノードは(データポインタ、キー値)の配列
  - さらに、隣同士はポインタによって結合





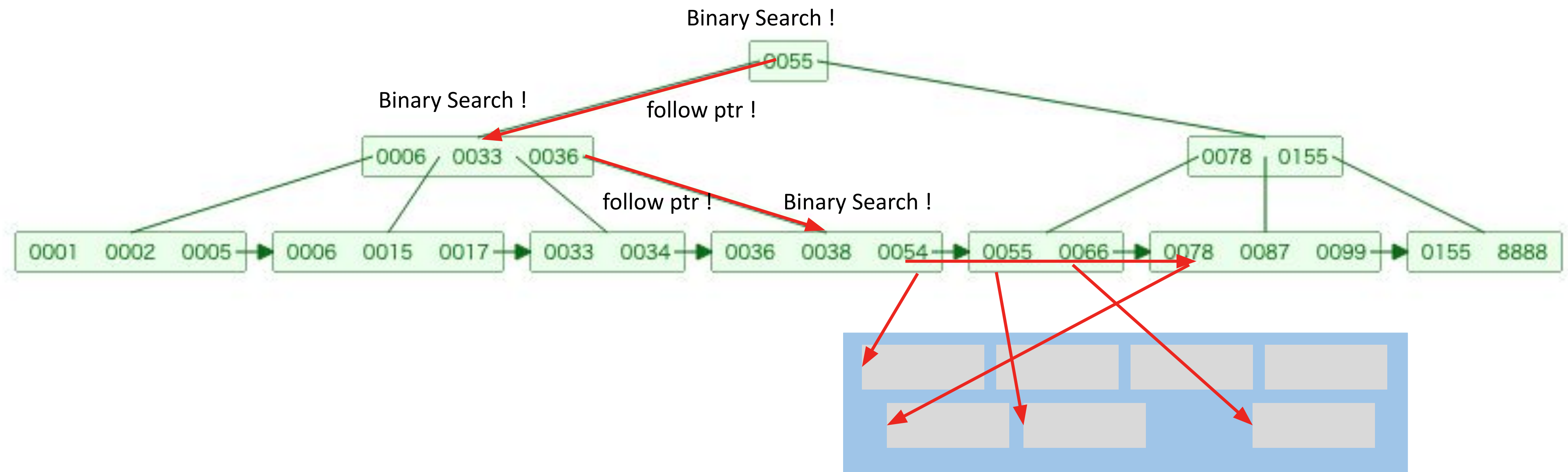
# B+Tree 等価比較

SELECT ... WHERE k = 54



# B+Tree 範圍檢索

SELECT ... WHERE BETWEEN 50 AND 80



# B+Treeで遊んでみよう

---

B+ Tree Visualization

# 他にもたくさん

---

- 代表的なインデックスの種類
  - ハッシュインデックス
  - ビットマップインデックス
  - 全文検索インデックス
  - 空間インデックス
- どのインデックスが利用できるかは製品次第

# インデックスの定義

---

- 実は、標準SQLにはインデックスについての規定はない
- 主キー制約やユニーク制約を定義すると、自動でインデックスが貼られる
- CREATE INDEX構文で明示的にも定義もできる

```
CREATE INDEX newIndex ON table (aaa, bbbb);
```

# インデックスの設計ポイント

---

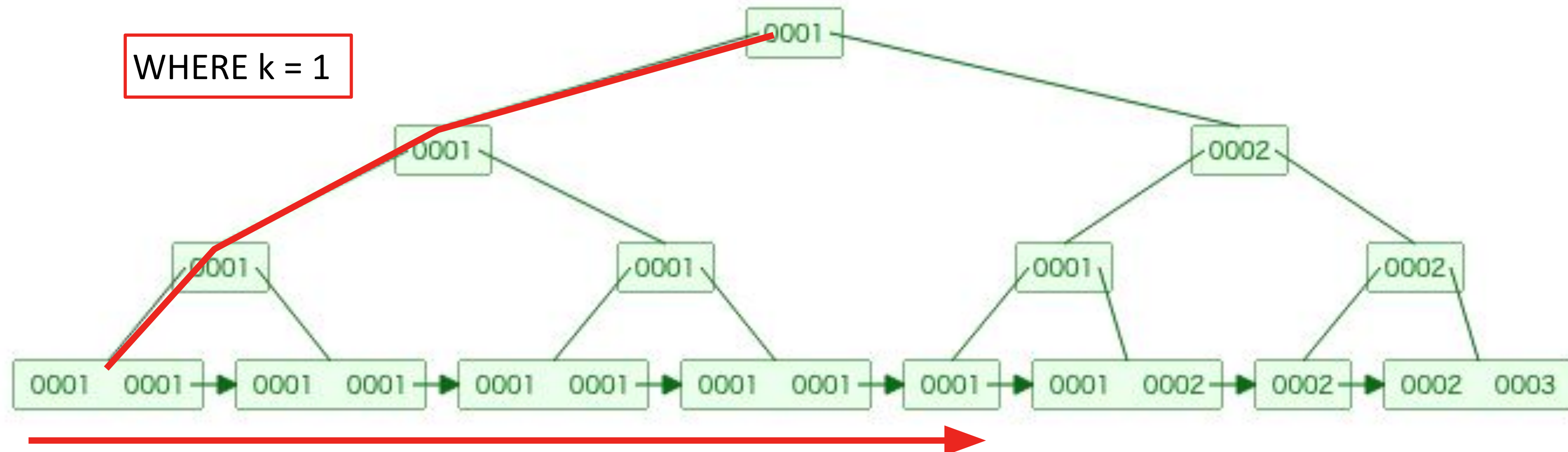
- 必要なインデックスだけを貼る
  - 更新性能、スペース、キャッシュ効率に関わるので
- **選択率が低くなるように貼る**
  - 選択率＝検索時に母集合からどれだけ絞り込めるか
- **カーディナリティの高い列を選ぶ**
  - カーディナリティ＝値のばらつき具合、ユニークな値の数



# 低いカーディナリティ

- カーディナリティが低いと選択率が悪くなる
  - トラバースのコストも馬鹿にならない
- 以下の例だと、半分にも絞り込めていない

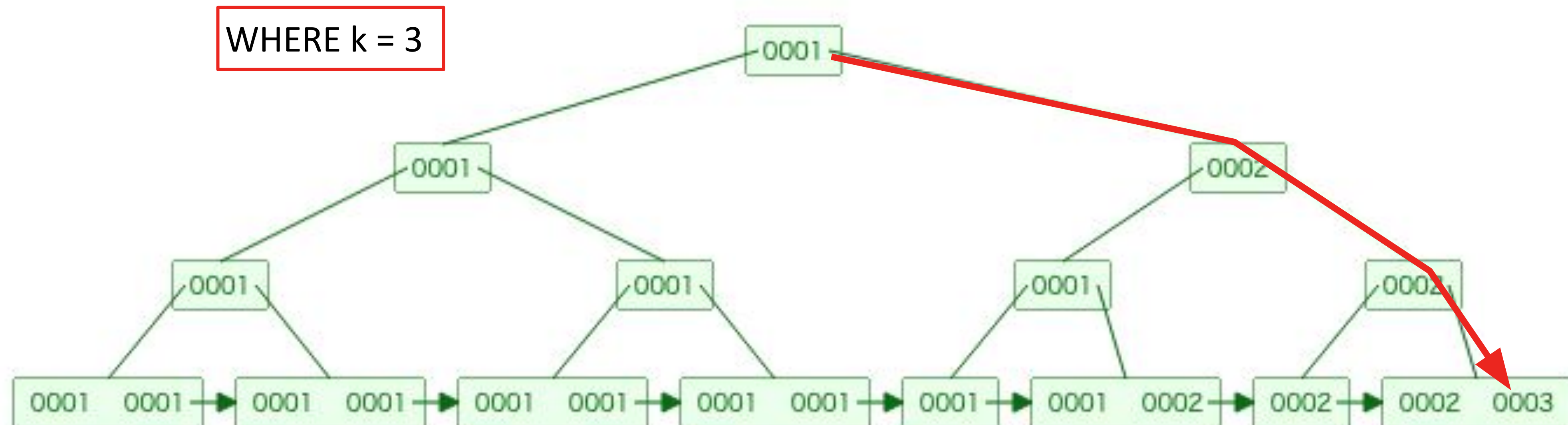
例: カーディナリティ = 3





# 例外もある

- データの分布に偏りがあれば、検索条件によって効果が大幅に変わる



# カーディナリティ in MySQL

```
SHOW index FROM student;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
student	0	PRIMARY	1	sid	A	90	NULL	NULL		BTREE
student	1	major	1	major	A	3	NULL	NULL	YES	BTREE

MySQL Workbench

# それでも効かない時1: 構文チェック

---

- **インデックスが使えない検索条件**になっていないか確認
  - インデックス列を使っていない
  - 中間一致、後方一致のLIKE述語
  - インデックス列で演算
    - NG: where age \* 10 > 100
    - OK: where age > 100/10
  - インデックス列に対して関数を使用
  - 否定形の利用

# それでも効かない時2: インデックスオンリースキャン

---

- インデックスを利用したもう一つのデータアクセス方法

例: 検索条件が存在しない (通常はフルスキャン)

```
SELECT cid, joined_at  
FROM club_member
```

# それでも効かない時2: インデックスオンリースキャン

- インデックスを利用したもう一つのデータアクセス方法

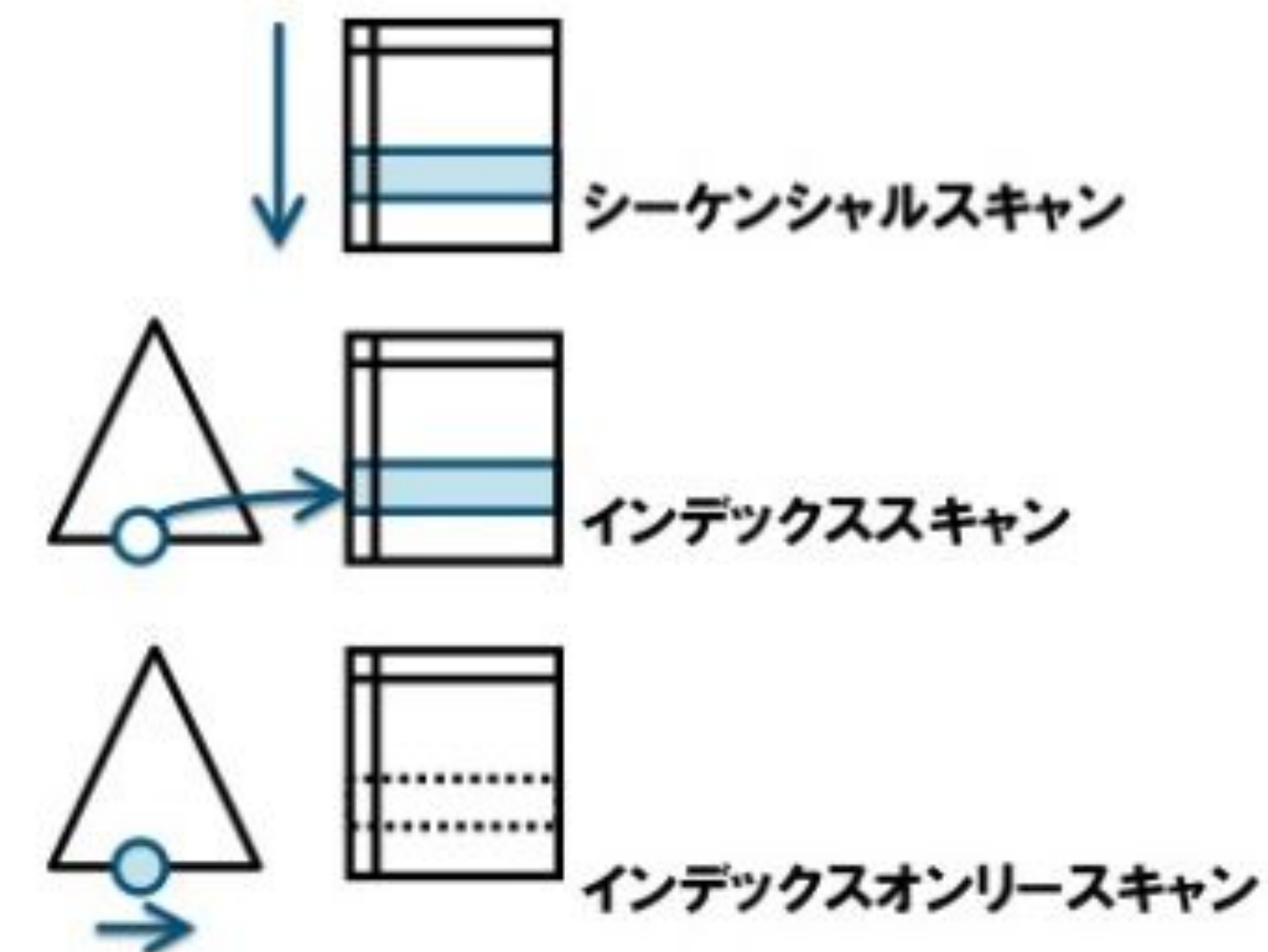
例: 検索条件が存在しない (通常はフルスキャン)

```
SELECT cid, joined_at  
FROM club_member
```

以下のように2列をカバーするインデックスを貼ると..

```
CREATE INDEX coveringIndex ON club_member (cid, joined_at);
```

- この2列へのアクセスは、テーブルではなく**インデックスをスキャン**するだけで済む



# それでも効かない時2: インデックスオンリースキャン

---

- 以下もインデックスオンリースキャンの対象

```
SELECT joined_at  
FROM club_member
```

```
SELECT cid, joined_at  
FROM club_member  
WHERE joined_at >= "2021/5/1"
```

# それでも効かない時2: インデックスオンリースキャン

---

- 採用時の注意点
  - 列数やサイズの上限
  - 性能はサイズに比例
  - 更新オーバーヘッド
  - SELECT句に新しい列を加えると使えない



# それでも効かない時3: SQLチューニング

---

- 手順
  - 遅いクエリを特定
  - **実行計画を確認**
    - **クエリ評価エンジン**が作成するデータアクセスプラン
  - チューニング
    - クエリ、テーブル、インデックスの見直し

6

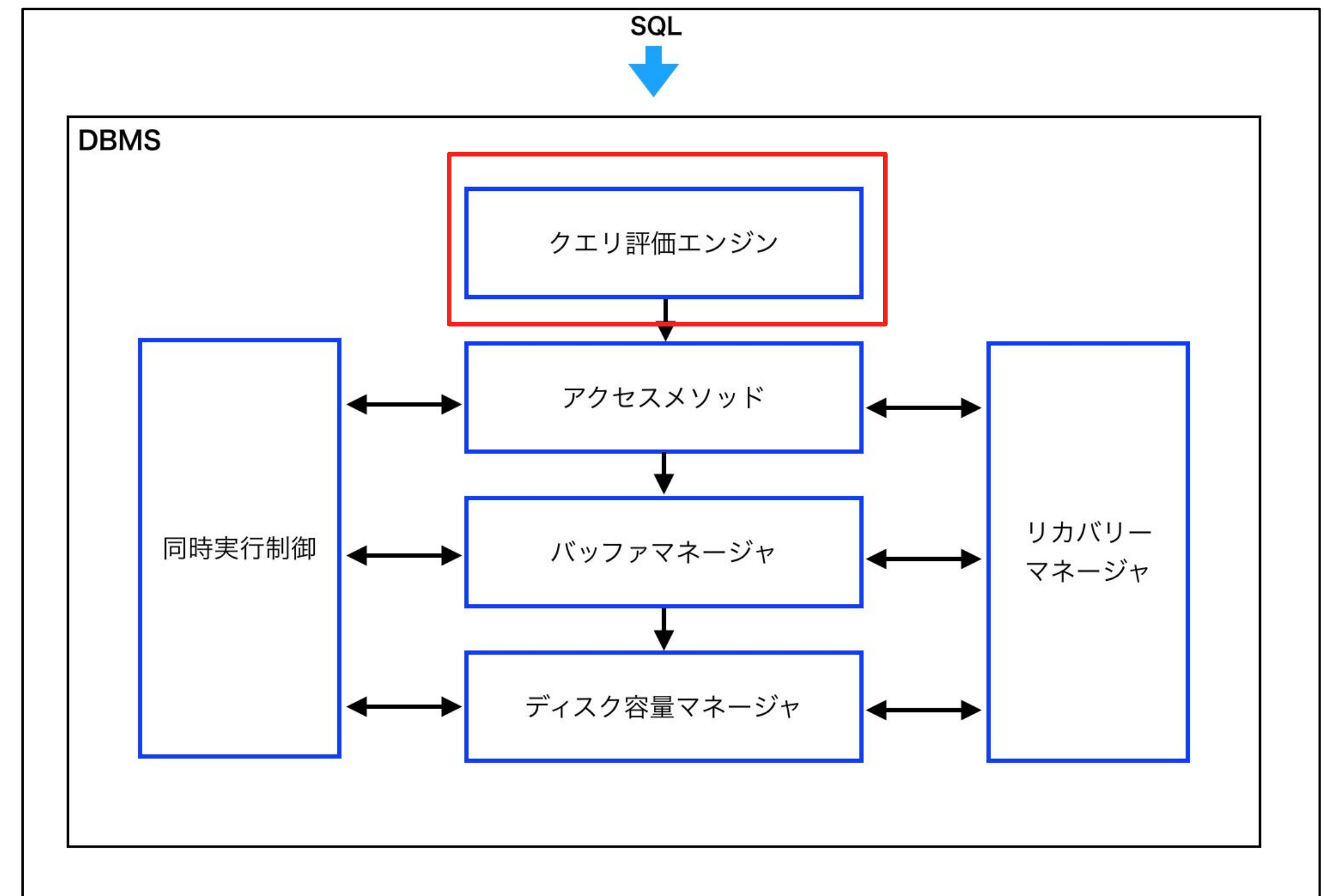
# 実行計画



# クエリ評価エンジンとは

- 役割は、クエリを解釈しどのような手順でデータにアクセスするかの計画を立て、それを実行すること
- **実行計画＝データアクセスの手順**
- DBMSのアーキテクチャだと上層に位置

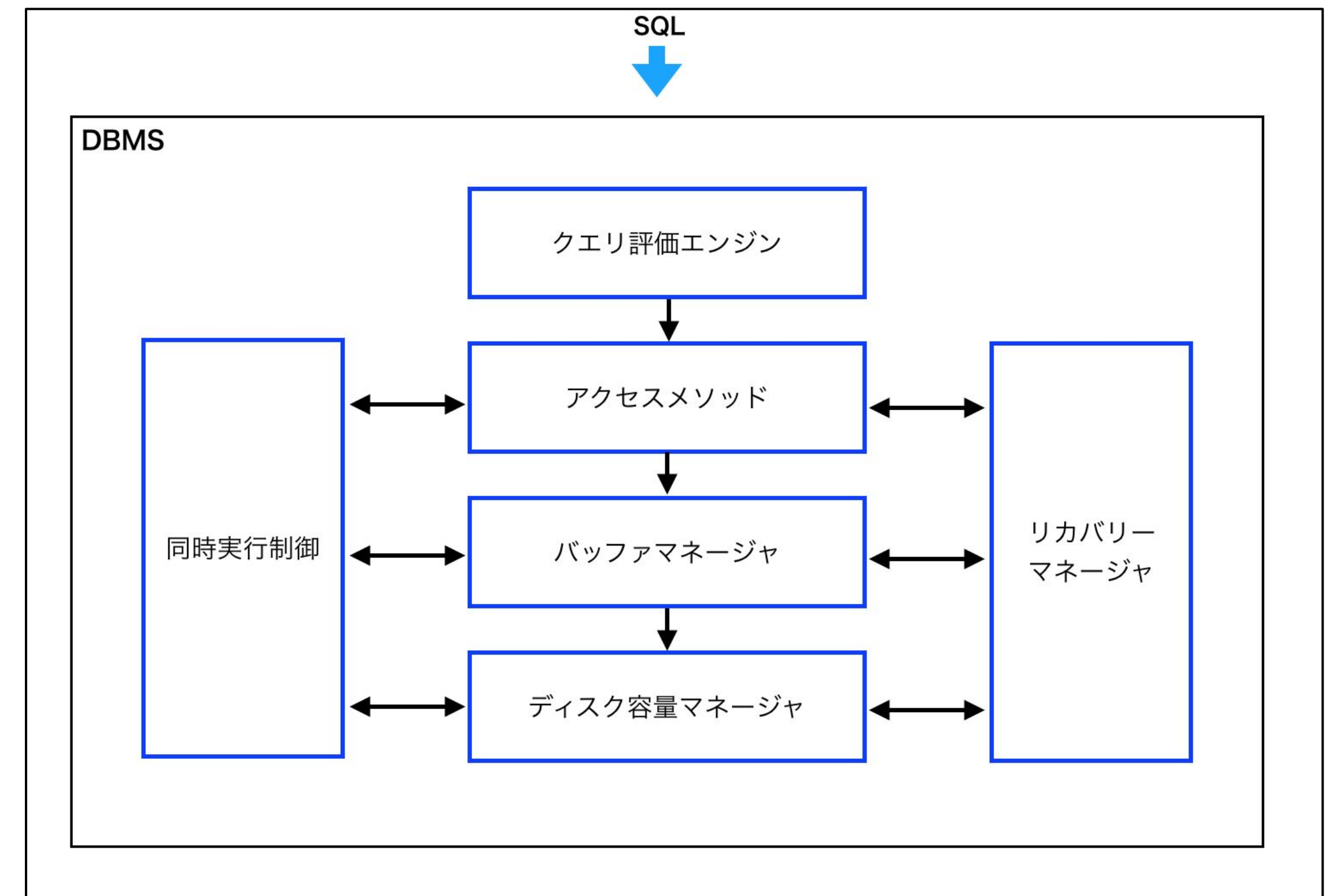
# DBMSのアーキテクチャ



# DBMSのモジュール

- クエリ評価エンジン
  - これからもう少しみていく
- アクセスメソッド
  - データアクセスの手段(API)を提供するのが仕事
- バッファマネージャ
  - メモリとディスクの間のデータの往来を管理
- ディスク容量マネージャ
  - ディスクやファイルシステムを抽象化するレイヤー
- 同時実行制御とリカバリマネージャ
  - 階層を横断する機能

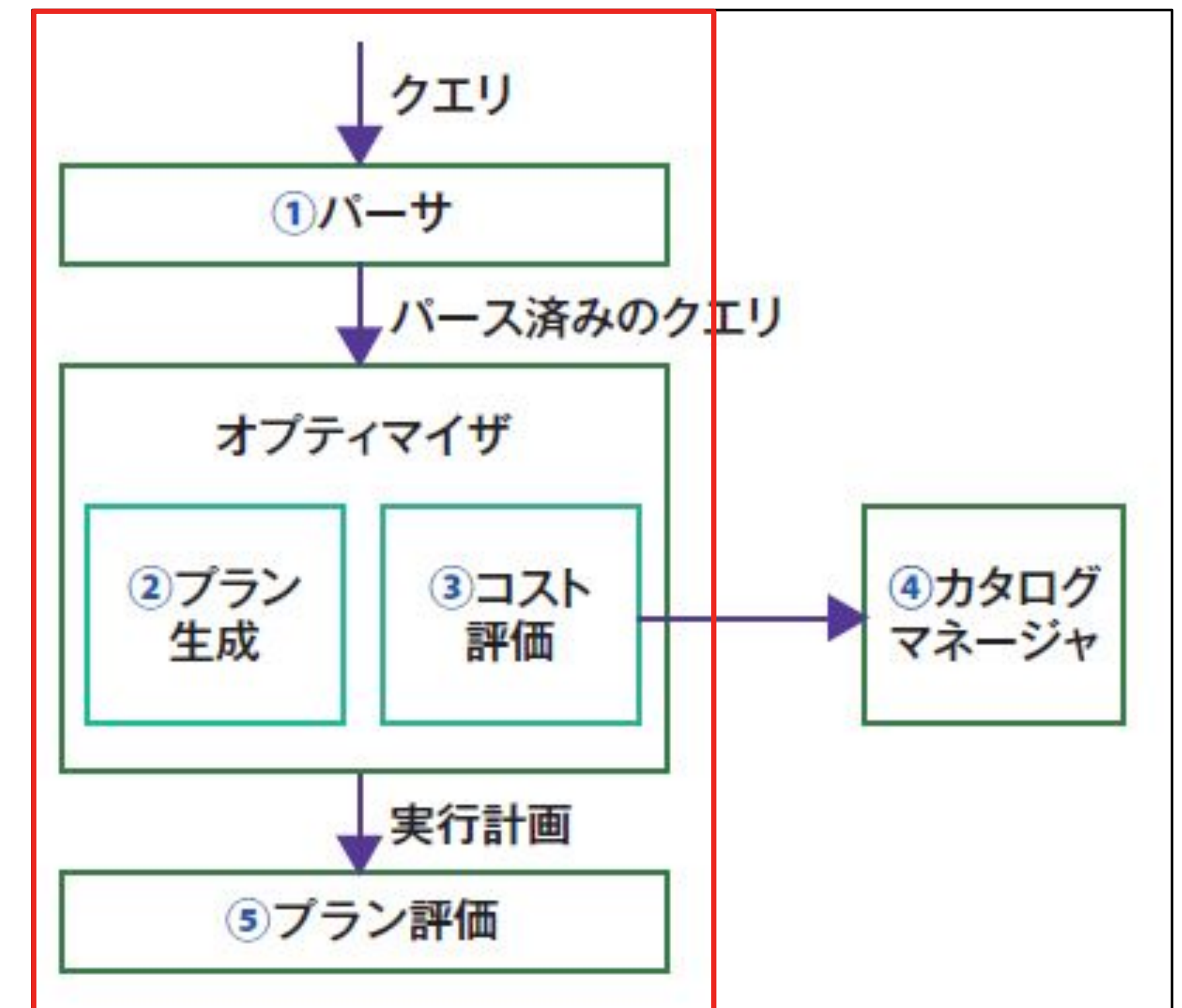
## DBMSのアーキテクチャ



# クエリが処理される流れ

- パーサ
  - クエリ(SELECT文)を構文解析
- オプティマイザ
  - テーブルやインデックスの統計情報を元に複数の実行計画を作成
  - 各プランのコストを算出し、最も低コストなものに絞る
- プランエグゼキューター(プラン評価)
  - 実行計画を手続き型のコードに変換して実際にデータアクセスを実行する

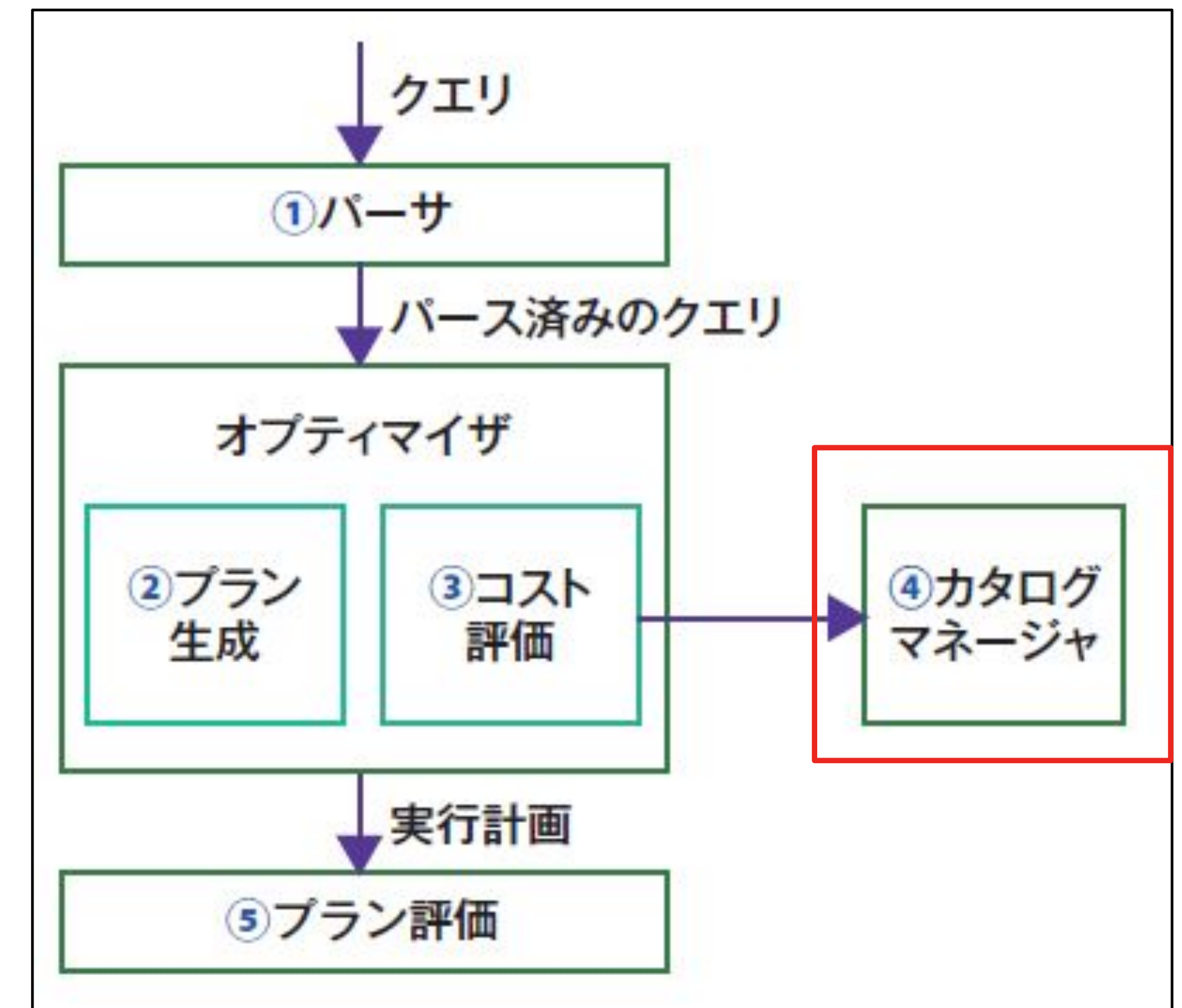
クエリが処理される流れ



# カタログマネージャー

- 役割は、メタデータを集めたテーブル群(カタログ)を管理すること
- カタログには、例えば、データ量、インデックスの有無、データのカーディナリティなど、様々な統計情報が格納されている

クエリが処理される流れ





# カタログ (統計情報) in MySQL

---

## テーブル統計

```
SELECT * FROM mysql.innodb_table_stats
WHERE table_name = 'student';
```

database_name	table_name	last_update	n_rows	clustered_index_size	sum_of_other_index_sizes
testdb	student	2021-05-10 02:31:17	72	1	1

## インデックス統計

```
SELECT * FROM mysql.innodb_index_stats
WHERE table_name = 'student';
```

database_name	table_name	index_name	last_update	stat_name	stat_value	sample_size	stat_description
testdb	student	PRIMARY	2021-05-10 23:48:14	n_diff_pfx01	90	1	sid
testdb	student	PRIMARY	2021-05-10 23:48:14	n_leaf_pages	1	NULL	Number of leaf pages in the index
testdb	student	PRIMARY	2021-05-10 23:48:14	size	1	NULL	Number of pages in the index
testdb	student	major_age	2021-05-10 23:48:14	n_diff_pfx01	3	1	major



# 実行計画ってどんなものなのか in MySQL

---

- MySQLでは、**EXPLAIN**をクエリの実頭につければ実行計画が見れる！

```
EXPLAIN  
SELECT s.name, s.age, c.name  
FROM student s  
  INNER JOIN club_member cm ON s.sid = cm.sid  
  INNER JOIN club c ON cm.cid = c.cid;
```

# 実行計画ってどんなものなのか in MySQL

- MySQLでは、**EXPLAIN**をクエリの実頭につければ実行計画が見れる！

**EXPLAIN**  
**SELECT** s.name, s.age, c.name  
**FROM** student s  
    **INNER JOIN** club\_member cm **ON** s.sid = cm.sid  
    **INNER JOIN** club c **ON** cm.cid = c.cid;

id	select_type	table	partitio...	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	cm	NULL	index	PRIMARY,coveringIndex	coveringIndex	8	NULL	3	100.00	Using index
1	SIMPLE	c	NULL	ALL	PRIMARY	NULL	NULL	NULL	3	33.33	Using where; Using join buffer (
1	SIMPLE	s	NULL	eq_ref	PRIMARY	PRIMARY	4	testdb.cm.sid	1	100.00	NULL

- table: アクセス対象のテーブル
- type: **テーブルへのアクセス方法**(インデックス利用有無、スキャン範囲などがわかる)
- possible\_keys: 利用可能なインデックス
- key: 選択されたインデックス
- key\_len: 読み取ったインデックスのバイト数
- rows: スキャンする**見積もり**行数

# 例：カバリングインデックスが有効か確認

```
EXPLAIN
SELECT cid, joined_at
FROM club_member
```

id	select_type	table	partitio...	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	club_member	NULL	ALL	NULL	NULL	NULL	NULL	3	100.00	NULL

```
CREATE INDEX coveringIndex ON club_member (cid, joined_at);
```

```
EXPLAIN
SELECT cid, joined_at
FROM club_member
```

id	select_type	table	partitio...	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	club_member	NULL	index	NULL	coveringIndex	8	NULL	3	100.00	Using index

# 補足：統計情報は常に正しい？

---

- A. いいえ、正しくありません
- 統計の自動再計算は非同期なので最新状態が反映されていないこともある
  - インデックスが効かないもう一つの原因！
- 特に、テーブルのデータが大きく更新されたら、実質と統計との差分も大きくなる
- 簡単な解決策は、明示的に統計の再計算を宣言すること

```
ANALYZE TABLE student;
```

# 実行計画を確認しよう



インデックスを貼る前後でクエリの実行計画にどんな変化があるか確認しよう！

Q1. WHERE句の条件列にインデックス

Q2. ORDER BY句の条件列にインデックス

Q3. GROUP BY句の条件列にインデックス

Q4. 複合インデックス

- 都度、前のインデックスはDROP

```
1.  
CREATE INDEX idx_major on student (major);  
EXPLAIN SELECT * FROM student WHERE major = "計算幾何学";  
  
2.  
CREATE INDEX idx_age on student (age);  
EXPLAIN SELECT * FROM student WHERE ORDER BY age;  
  
3.  
CREATE INDEX idx_major on student (major);  
EXPLAIN SELECT major, COUNT(*) FROM student WHERE age > 20 GROUP BY major;  
  
4.  
CREATE INDEX idx_major_age on student (major);  
EXPLAIN SELECT * FROM student WHERE major = "計算幾何学" ORDER BY age;
```



# 補足: 実行計画のVISUAL EXPLAIN

- MySQL Workbench の VISUAL EXPLAINを使うと直感的なフロー図がみれる

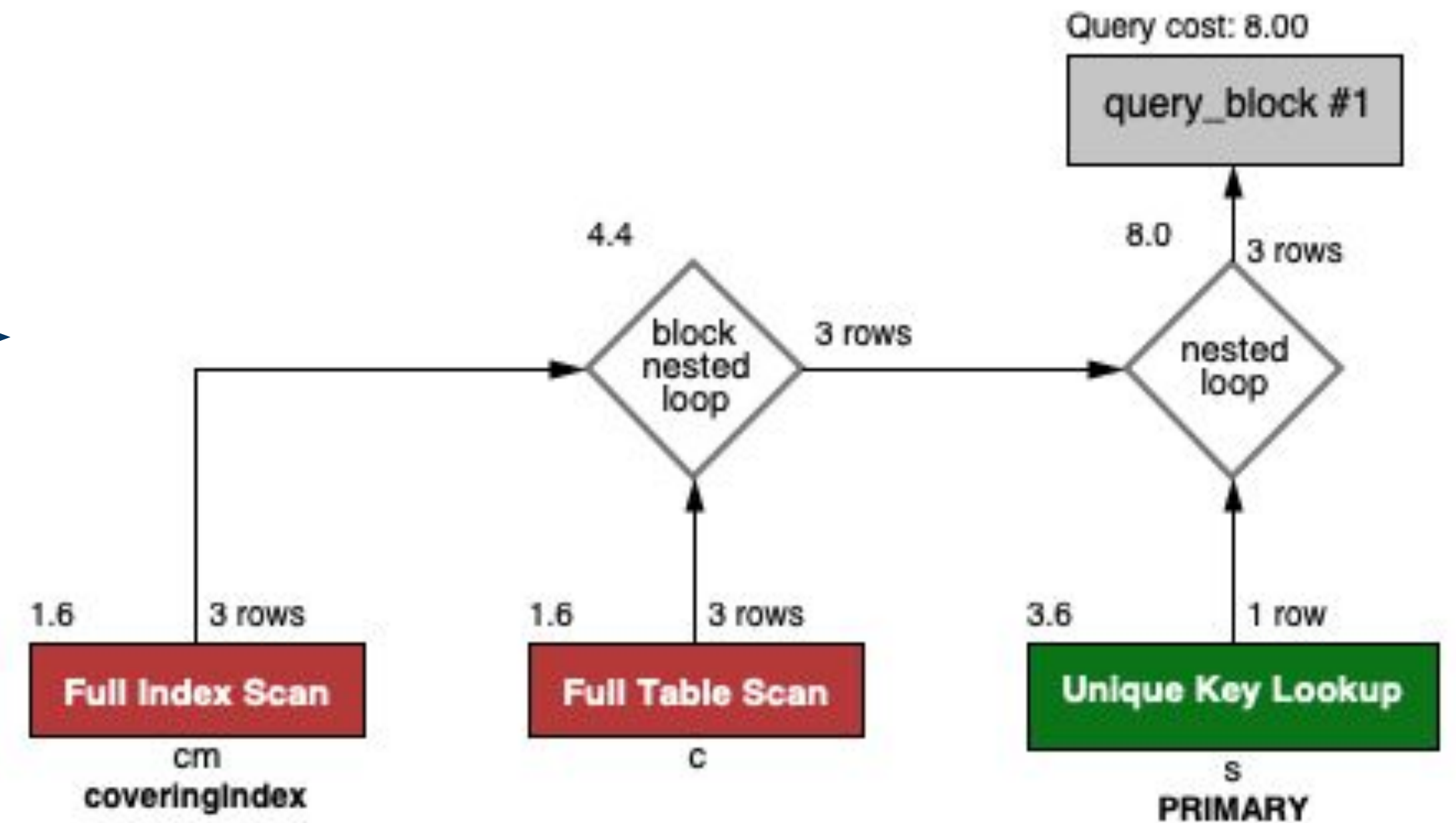
**EXPLAIN**

**SELECT** s.name, s.age, c.name

**FROM** student s

**INNER JOIN** club\_member cm **ON** s.sid = cm.sid

**INNER JOIN** club c **ON** cm.cid = c.cid;



7

SQL II





# 入れ子型質問 (サブクエリ)

---

- 入れ子型質問 (nested query)
  - クエリを含んだクエリ
  - 入れ子は何重にもできる
  - サブクエリの結果もテーブル

例: クラブに参加している学生一覧

```
SELECT name  
FROM student  
WHERE sid IN (SELECT sid FROM club_member);
```

サブクエリ



# 入れ子型質問(サブクエリ)

---

- サブクエリは、ほぼどこにでも書ける
  - SELECT句, FROM句, WHERE句, HAVING句, ....

例: 専攻人数が30人未満の分野を専攻している学生一覧

```
SELECT s.name, s.major
FROM student s INNER JOIN
  (SELECT major
   FROM student
   GROUP BY major
   HAVING COUNT(*) < 30) m
ON s.major = m.major;
```

# CASE式

---

- CASE式は**条件分岐**を記述するためのもの
- WHEN句の評価式が評価され、THEN句の式が返される

例: 学生に年齢カテゴリを付与

```
SELECT
name,
CASE
  WHEN age >= 22 THEN "22歳以上"
  WHEN age >= 20 THEN "20歳以上22歳未満"
  ELSE "20歳未満"
END as age_category
FROM student s;
```

# CASE式

---

- CASE式は**条件分岐**を記述するためのもの
- WHEN句の評価式が評価され、THEN句の式が返される

例: 学生に年齢カテゴリを付与

```
SELECT
  name,
  CASE
    WHEN age >= 22 THEN "22歳以上"
    WHEN age >= 20 THEN "20歳以上22歳未満"
    ELSE "20歳未満"
  END as age_category
FROM student s;
```

書くときのポイント

- 各分岐が返すデータ型を統一
- 短絡評価を意識
- ELSE句は必ず書く (ないと暗黙にELSE NULL)

# CASE式

---

- こういうこともできる

例：各分野における年齢カテゴリごとの学生数

専攻	22歳以上の数	20歳以上22歳未満の数	20歳未満の数
文化人類学	2	11	8
計算機科学	11	20	10
経済学	9	12	9

# CASE式

- こういうこともできる

例：各分野における年齢カテゴリごとの学生数

専攻	22歳以上の数	20歳以上22歳未満の数	20歳未満の数
文化人類学	2	11	8
計算機科学	11	20	10
経済学	9	12	9

集約関数の中で使うと、行を列に変換できる

```
SELECT
major,
SUM(CASE WHEN age >= 22 THEN 1 ELSE 0 END) as "22歳以上の数",
SUM(CASE WHEN age >= 20 AND age < 22 THEN 1 ELSE 0 END) as "20歳以上22歳未満の数",
SUM(CASE WHEN age < 20 THEN 1 ELSE 0 END) as "20歳未満の数"
FROM student
GROUP BY major;
```

# クエリを書いてみよう (cont.)



Q1. 31アイスクリームのフレーバーの組み合わせ一覧がほしい (“ダブルサイズ”の選択肢をください)

- ただし、同じフレーバーはなし
- 同じ組み合わせは一覧にいれないでほしい
- カロリーの合計もついでにほしい

Q2. 上の一覧から最適な組み合わせを一つ選んでほしい

- 合計カロリーが350以下
- どちらかのフレーバーの種類は必ずELEGANT
- この条件にあうもので一番カロリーが低いもの

Q3. 同じ感じで、最適なトリプルの組み合わせもほしいな...

Q4. 条件にあうフレーバートップ3を順番に出してほしい

- 各種類の中で一番カロリーが低いものが対象 (■ 最小のものは1つに決まる前提)
- 好きな種類の順番は ELEGANT -> THE 31 -> それ以外

Q5. 大豆を含まないフレーバーの一覧が欲しい

```
ice_cream_flavor(name CHAR, kind CHAR, calorie INTEGER)
```

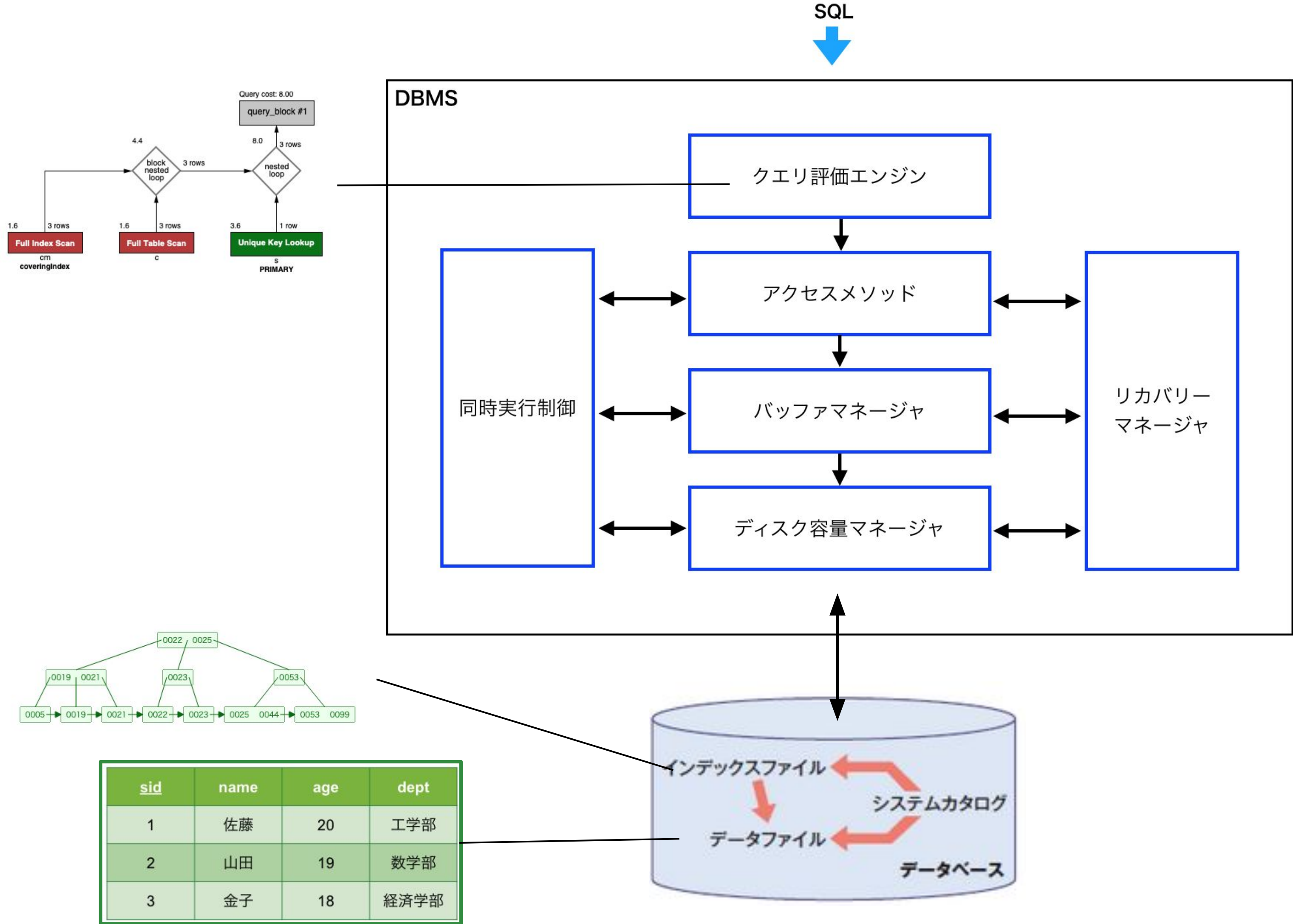
```
ice_cream_flavor_allergy(ice_name CHAR, allergy_name CHAR)
```



# 最後

- 大事なところ

- データ独立性
- データモデル (関係モデル)
- 宣言型プログラミング (SQL)
- トランザクション (ACID)
- データベース設計 (ER, 正規化)
- インデックス (B+tree, カーディナリティ)
- 実行計画



# 主な出典

---

- 達人に学ぶSQL徹底指南書 第2版 初級者で終わりたくないあなたへ
- 理論から学ぶデータベース実践入門 ―― リレーショナルモデルによる効率的なSQL
- SQL実践入門 ――高速でわかりやすいクエリの書き方
- Webエンジニアのための データベース技術[実践]入門
- リレーショナルデータベース入門―データモデル・SQL・管理システム・NoSQL 第3版
- 失敗から学ぶRDBの正しい歩き方
- データ指向アプリケーションデザイン―信頼性、拡張性、保守性の高い分散システム設計の原理
- CS 15-445/645 Database Systems at Carnegie Mellon University
- CS186 Introduction to Database Systems at University of California, Berkeley