データベース講義資料

1　データベースの基礎概念

1.1　データとは?

- データベースの用語では,「データ」とは単なる数値や文字列ではなく，実世界の事物の持つ性質を反映したものを指す (例えば，ある授業の受講生が 40 人とか，ある学科に所属する学生が 50 人など，実世界に実際に存在する数).
- データは計算機上に格納されるまでに，ユーザが検索できるように編集・加工が成される（図 1 参照）.
- データと情報：実世界の事物を反映して，統一的な形式で表現される（図 2 参照）ことに加えて、データにそれの持つ意味を付加し，ユーザに取って価値のあるものにしたものを特に「情報」と呼び,「データ」と区別する（図 3 参照）.

1.2　データベースとは?

1.2.1　データベースの条件

データベースは，単純にデータを収集してまとめたものではなく，次の条件を満たす必要がある.

- 実世界の事物の持つ属性や事物間の相互の関連が，データの構造として表現されている.
- データが複数の組織で共有できる資源となっている.

1.2.2　ファイルとデータベース

ファイルとデータベースの違いを説明するため，まずファイルについて説明する.

応用プログラム: データを参照するプログラムを「応用プログラム」と呼ぶ.

ファイル: 応用プログラムに依存（図 4 参照）している.

- ファイルは特定の応用プログラムにより作成される.
- ファイルのフォーマットは応用プログラムごとに異なる（ある応用プログラムが作成したファイルを別の応用プログラムから直接参照することはできず，応用プログラムごとにデータ変換が必要）.
- 同時に複数の応用プログラムから同じファイルを更新するときは，応用プログラム側に相互排除等の更新制御手続きが必要となる.

1.2.3　データ管理における問題点

通常のファイルを使ったデータの管理では，しばしば次のような問題が生じる．後述するように，データベースを構築することで以下の問題を解決することができる.

(1) データ独立性の未達成（2.3.1 節参照）
（例）データを入れたファイルを別のディスクに移動したところ，そのファイルのパス名の変更等により今まで使っていたソフトウェアからファイルが読めなくなった.

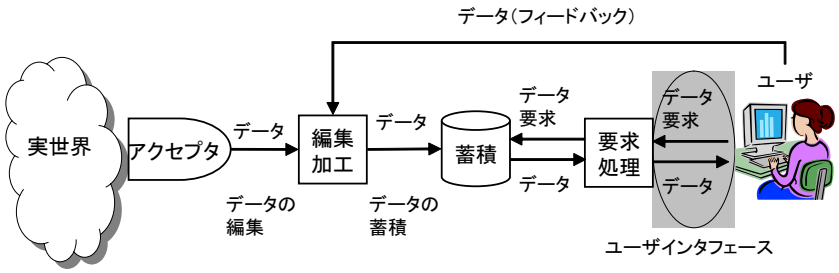


図 1: データの格納過程

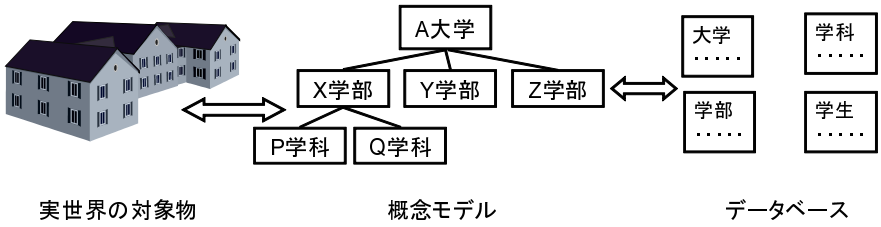


図 2: 実世界の事物を反映したデータの表現

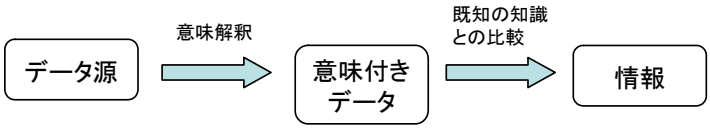


図 3: データと情報

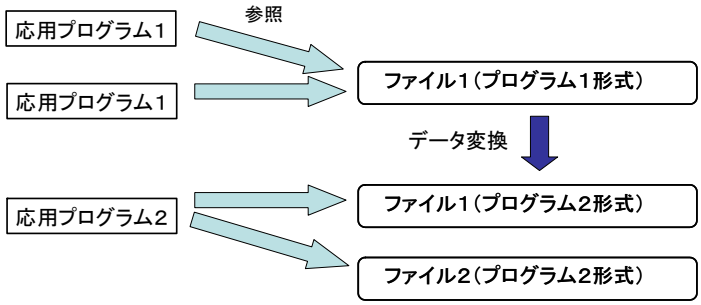


図 4: 応用プログラムとファイルの関係

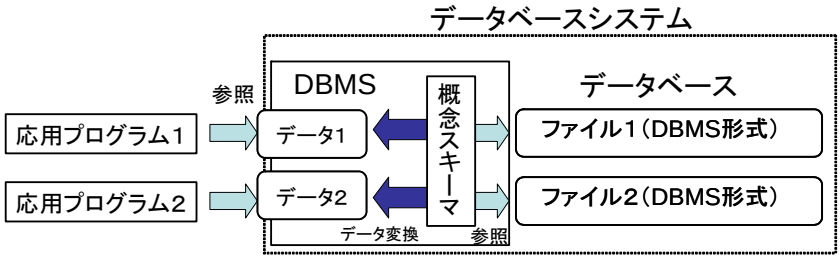


図 5: DBMS を介したファイルの参照

(2) データ一貫性の欠如
（例）データを複数のファイルに分散して入れていたが，更新を繰り返すうちに，各ファイルのデータ間の対応関係がわからなくなり，対応が取れなくなった．
(3) 効率的なデータ格納の問題
（例）同一のデータを入れたファイルを，多数の人がそれぞれ自分用のコピーを作ろうとしたため，ファイルの使用容量が増加し，ディスクの制限容量を越えた．
(4) データ標準化の未達成
（例）同一のデータを，複数の異なるソフトウェアで利用したいが，ソフトウェアごとにファイル形式が異なるため，あるソフトウェアで作成したファイルは，形式を変換しないと別のソフトウェアからは読むことができない．
(5) データの機密保護の問題
（例）あるデータの一部を，特定の人にだけ検索できるようにしたいが，ファイル中の一部のデータのみへのアクセスを許可する設定ができない．

1.2.4 データベース管理システム

データベース: 応用プログラムとは独立（図 5 参照）しており，データベースの構成に変更があっても，DBMS と各応用プログラムの間のインタフェースは変更する必要がない．

- データベースは，応用プログラムから直接作成することはできず，**データベース管理システム** (以下，DBMS) に作成を依頼することにより，DBMS によって作成される．
- DBMS を使ってデータベースを構築すれば，データ独立性の達成，データの一貫性の保証，データ蓄積の効率化（重複データの抑制），データの標準化，データの機密保護などが容易となる（詳しくは 2.3 節を参照）．
- 複数の応用プログラムから同一のデータベースを更新するときも，DBMS が各更新処理が正しくファイルの内容に反映されるよう更新制御を行うので，応用プログラムでは更新制御を意識する必要がない．

その他の DBMS の機能:

- 2 次記憶上のデータへの高水準かつ効率の良いアクセス手段を提供（例：ある条件を満たすデータの集合を求める）．
- エンドユーザ向けに対話型インターフェース，応用プログラムに対しては API（応用プログラムインターフェース）を提供．
- 標準的なデータベース言語の提供 (SQL)．

2 データモデリング

2.1 データモデリングとは？

データベースを開発する過程では，実世界の事物を DBMS で扱えるデータの形式で記述する必要がある．この過程を**データモデリング**と呼ぶ．

データモデリングは，次の 2 段階で行われる．

概念モデリング: 実世界の事物とそれらの相互の関連をもとに，それを表現した概念モデルを作成する．記号系としては，概念モデル記述言語が使われる．

論理モデリング: 概念モデリングで作られた概念モデルをもとに，計算機上で実際にデータベースを設計するための論理モデルを作成する．記号系としては，**データモデル**が使われる．

2.2 概念モデルと論理モデル

2.2.1 概念モデル

- 実世界の事物を忠実に反映するモデル．
- 計算機上で直ちに実装可能かどうかは問わない．
- 概念モデル記述言語で表現される．

概念モデル記述言語

(1) **実体－関連モデル** (Entity-Relationship Model; E-R モデル)
実体（entity, 実世界に存在する事物に対応）と**関連** (relationship, 実体と実体との間に成り立つ対応関係) により記述．実体と関連は，それらの持つ特徴や性質を表す**属性**を持っている．

実体－関連モデルでは表記の方法が何通りかある．本講義では Chen の記法（図 6 参照）で表記するが，他にも Crow's Foot 記法（図 7 参照）で表記することがある．

(2) 意味的データモデル (semantic data model)
FDM (functional data model), GSM(generalized semantic data model).

(3) CAD の形状モデル
ワイヤーフレーム，サーフェイス，ソリッド

(4) UML (Unified Modeling Language): 本来はオブジェクト指向プログラミングでプログラムの仕様をモデル化する際に使われる表記法であるが，クラス図は概念モデルの記述にも使うことが可能である（図 10 参照）．

(5) 自然言語：概念モデルを言葉で表現する．

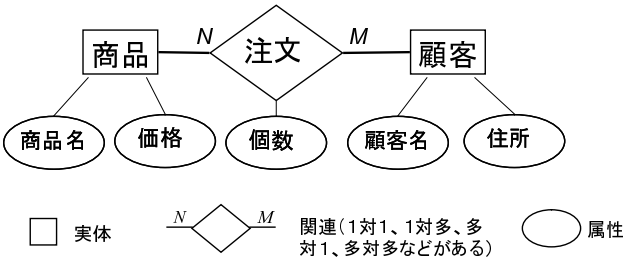
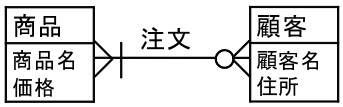
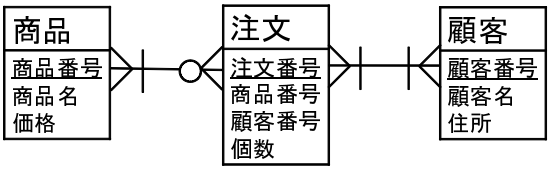


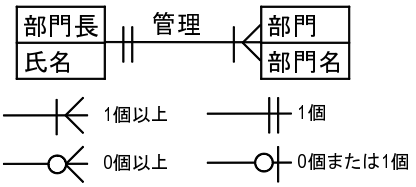
図 6: 実体－関連モデルの例（Chen の記法）



(a) 関連の属性を省略する表記



(b) 論理モデルに相当する表記 （関連の属性を表記．主キーとなる属性に下線.）



(c) 関連で対応付けられた実体の個数を細かく表記

図 7: 実体－関連モデルの例（Crow's Foot 記法）

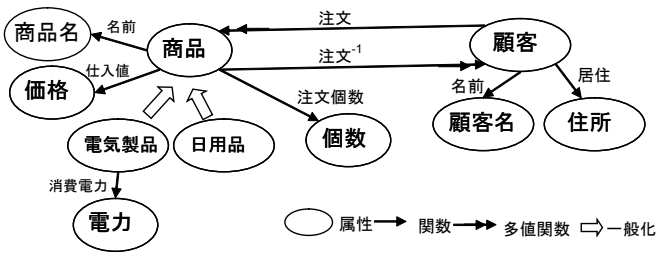


図 8: FDM モデルの例

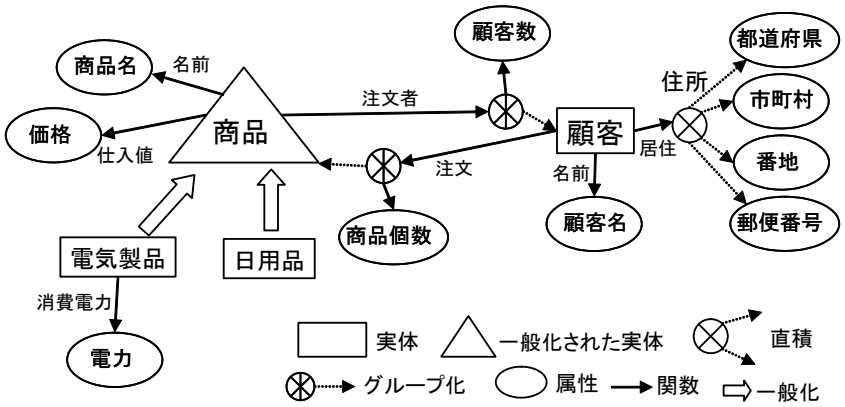


図 9: GSM モデルの例

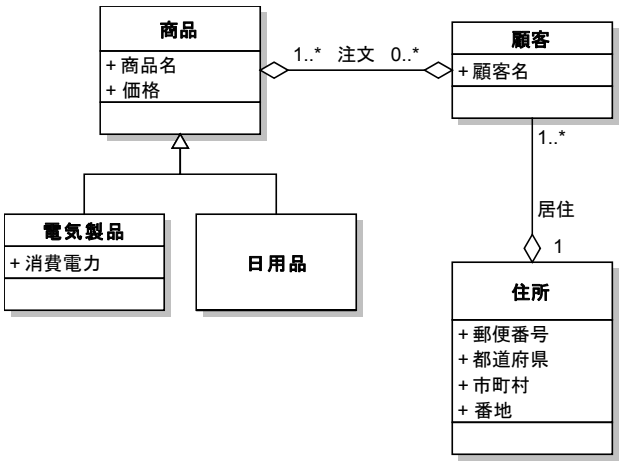


図 10: UML のクラス図の例

2.2.2 論理モデル

- 概念モデルをもとに，実際に計算機上で実装するために作成される．
- 記述能力よりも，2 次記憶上で表現しやすく，アクセス効率が高い形式であることが重要である．
- 論理モデルの記述に用いられる記号系は**データモデル**と呼ばれる．代表的なデータモデルには次のようなものがある．

(a) ハイアラキカルデータモデル (Hierarchical Data Model)

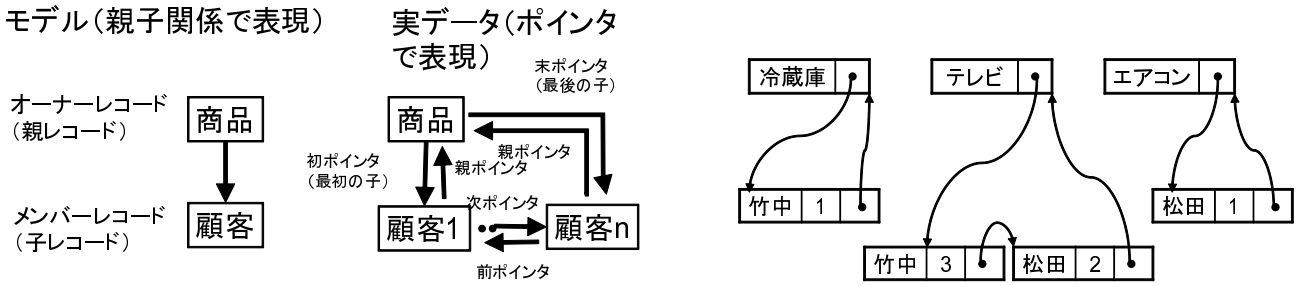
- データを親子関係をもとにした階層的な形式で表記する（図 11 (a) 参照）．階層的な表記法なので，複数の親を持つような関係は表現できない．
- このモデルでは，論理的には図 11 (b) のような形で表記するが，計算機上では図 12 のように実装される．

(b) ネットワークデータモデル (Network Data Model)

- データを親子関連型（set type）と呼ばれる形式で表現する（図 13 (a) 参照）．この形式では，複数の親を持つような表現が可能である．
- このモデルでは，論理的には図 13 (b) のような形で表記する．

(c) リレーショナルデータモデル (Relational Data Model)

- E. F. Codd により 1969 年に提案されたデータモデル．
- データをリレーションと呼ばれる表形式で表記する（図 14 参照）．



(a) モデルの概要　　(b) モデルの表記例

図 11: ハイアラキカルデータモデル

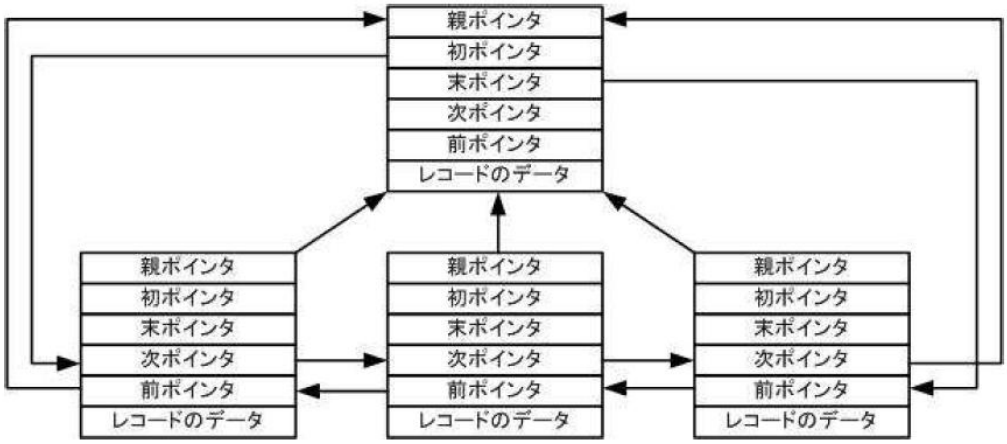
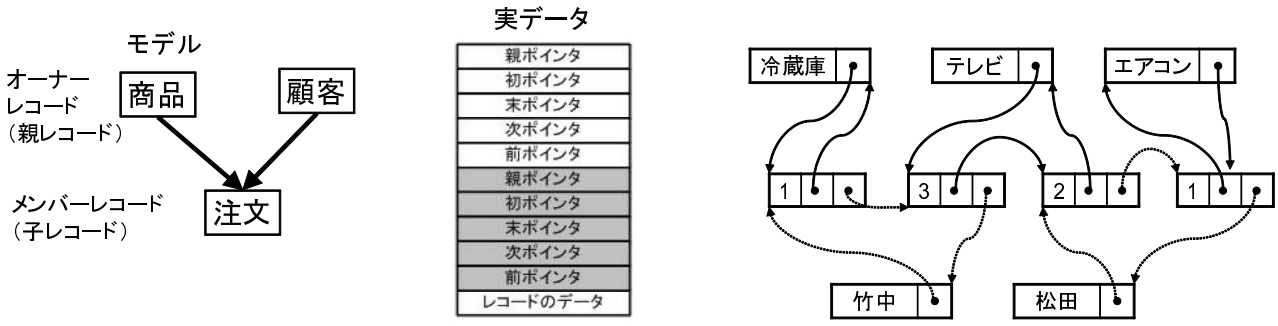


図 12: ハイアラキカルデータモデルの実装



(a) モデルの概要　　(b) モデルの表記例

図 13: ネットワークデータモデル

商品

商品番号	商品名
G1	冷蔵庫
G2	テレビ
G3	エアコン

顧客

顧客番号	顧客名
C1	竹中
C2	松田

注文

注文番号	商品番号	顧客番号	個数
O1	G1	C1	1
O2	G2	C1	3
O3	G2	C2	2
O4	G3	C2	1

図 14: リレーショナルデータモデルの表記例

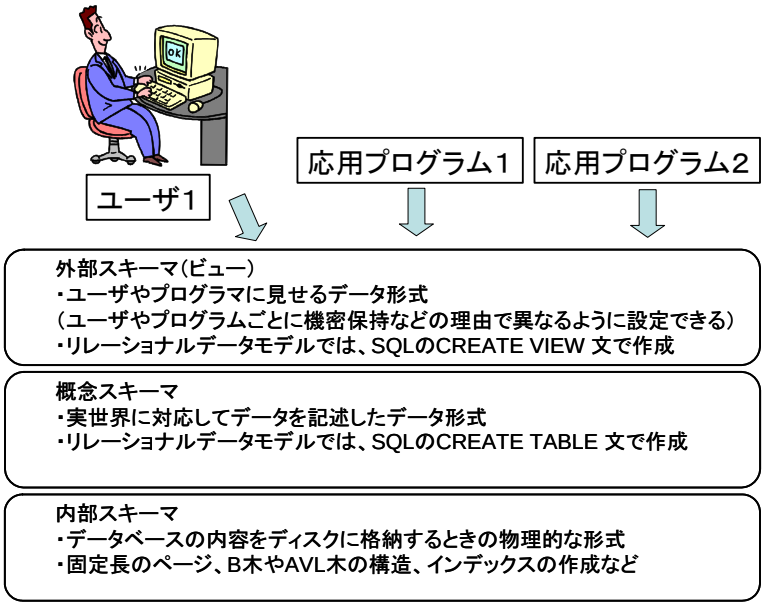


図 15: 3 層スキーマモデル

- リレーション間の関連の表現では，データをポインタで直接結ぶのではなく，データの識別子 (ID) を使った意味的なポインタで表現する．これにより，データの物理的な実装（内部スキーマ）と，論理的な表現（概念スキーマ）の分離が容易になり，物理的データ独立性の達成が可能となる．
- 現在では，標準的なデータモデルとなっており，Oracle や Sybase など多数のデータベース管理システムで採用されている．

2.2.3 データモデリングを 2 段階に分ける意義

- データモデリングには，実世界のデータを忠実に表現するという目標と，計算機上で実装可能でアクセス効率が良いデータベースを設計するという目標の 2 種類の目標が存在する．1 回の変換でこれらを同時に達成するのは困難だが，モデリングを 2 段階にすれば，この 2 つの目標が段階的に達成できる．
- 概念モデルは実世界の事物と対応しているので，実世界に変化があっても概念モデルを通すことによりデータベースの修正が容易となる．
- データベースの実装上の理由などで論理モデルの変更が必要になっても，概念モデルから変更後の論理モデルを作るだけですみ，実世界のデータを反映する作業を再度行う必要がない．

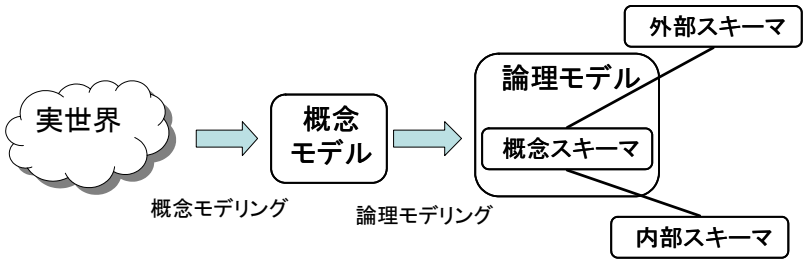


図 16: 3層スキーマモデルとデータモデリングのモデルとの関係

2.2.4 データベース・システムにおける抽象化のレベル

ユーザ，データベースの管理者，DBMS のどの立場に立つかで，データベースの捉え方が異なる．ANSI/SPARC の 3 層スキーマモデルでは，これらそれぞれに対応した 3 層を規定している（図 15 参照）．

外部スキーマ： 個々のユーザや応用プログラムごとのデータベースの見え方を規定するもの．ビューとも呼ぶ．
概念スキーマ： データベースの管理者が，データベースの構成を論理的に記述したもの．
内部スキーマ： 使用する OS やディスクなどのハードウェア構成によって規定される物理的なデータの格納形式．

データモデリングで作成されるモデルは，この 3 層スキーマと用語が似ているが，これらは全く別のものである（図 16 参照）．

2.3 データベース構築の意義

DBMS を使ってデータベースを構築することには以下のような意義がある．

(1) データ独立性の達成
(2) データの一貫性の保証
(3) データ蓄積の効率化（重複データの抑制）
(4) データの標準化
(5) データの機密保護

(1) と (2) は特に重要なので以下にまとめておく．

2.3.1 データ独立性

データ独立性: データ独立性とは，データベースの構成の変更が応用プログラムに影響を及ぼさないことであり，大量のデータを効率良く管理するためには必須の概念で，データベースを構築する意義の一つである．次に述べるように物理的データ独立性と論理的データ独立性の 2 種類がある．

[物理的データ独立性]
内部スキーマ（DBMS がデータベース中のデータを物理的にディスク上で管理している形式）と概念スキーマ（データベース管理者がデータベースを設計・管理するときの形式）が独立（つまり，物理的データ独立性により，内部スキーマの変更が概念スキーマに影響しないこと）．

例えば，データベース管理者は，新たにデータベースを設計しなおすことなしに，データベースを置いているディスクや，DBMS の稼働している OS や計算機，さらには同じデータモデルを使う限り，DBMS そのものも変更することができる．

[論理的データ独立性]
概念スキーマと外部スキーマ（応用プログラムがデータベースを検索・更新する形式）が独立（つまり，論理的データ独立性により，概念スキーマの変更が外部スキーマに影響しないこと）．

例えば，応用プログラムに変更を加えることなしに，データベース管理者は，データベースの構成の変更（主にデータ項目の追加・削除）を行うことができる．すなわち，概念スキーマが変わっても応用プログラムを変更する必要がない．

2.3.2 データの一貫性

データの**一貫性** (integrity, 保全性とも呼ばれる)：データベースに対して意味的にありえないデータの登録や更新が行われていない状態 (詳細は 7 章参照).

一貫性制約 (データの一貫性を保証するための条件) の例：最低賃金，年齢制限，最大労働時間.

3　リレーショナルデータベースの構造

3.1　リレーショナルデータモデル

- E. F. Codd により 1969 年に提案.
- 現在のデータベースシステムでの標準的なモデル.
- データ独立性 (2.3.1 節参照) の達成がネットワークデータモデルやハイアラキカルデータモデルよりも容易（概念レベルでデータベースの構造に変化があっても応用プログラムを変更する必要がない）.
- データベースの構成単位は表形式のリレーションであり，操作が単純・明解
- 集合論，述語論理学という理論的基盤を持つ.

3.2　リレーション

[リレーションの定義]（図 17，図 18 参照）
ドメイン $D_1, D_2, \cdots, D_n$ 上の**リレーション** R とは直積 $D_1 \times D_2 \times \cdots \times D_n$ の有限部分集合である.

直積　：$D_1 \times D_2 \times \cdots \times D_n = \{(d_1, d_2, \cdots, d_n) | d_1 \in D_1, d_2 \in D_2, \cdots, d_n \in D_n\}$.
タップル　：直積の要素，$t = (d_1, d_2, \cdots, d_n)$: n- タップル．リレーションは直積の有限部分集合で，その要素はタップルである.
リレーションの属性　：各リレーションでそのリレーションを構成しているドメインを個別に指定する．ある属性 A_i で指定されたドメインを $dom(A_i)$ と表す.
リレーションの次数　：リレーションが持つ属性の数.
リレーションの濃度　：リレーションが要素として持つタップルの数.
タップルの成分　：タップルが持つ個別のデータ．属性により指定される（例：リレーション R の要素であるタップル t での属性 A_i に対応する成分は $t[A_i]$ と表す）.

リレーションはタップルを要素とする「集合」であるので，一つのリレーションが同じタップルを複数個要素として持つことはない.
cf. マルチ集合（要素の重複を許す）．SQL ではマルチ集合が使われる（5 章参照）.

(注意) ドメインはデータの型を規定するための概念であり，属性はあるリレーションが要素として持つタップルの各成分を規定する．従って，ドメインはリレーションとは独立しており複数のリレーションが同じドメインから構成されることがあり得るが，属性はリレーションごとに決められておりリレーションが異なれば別の属性となる.

3.3　リレーショナルデータモデルの実装

リレーショナルデータモデルの実装では，レコード（タップルの値を入れるデータ領域）をポインタでつないで管理する（図 19 参照）.

このレコード間のポインタは，ハイアラキカルデータモデルやネットワークデータモデルとは異なり，論理モデルには一切現れないため，ユーザが意識する必要はない.

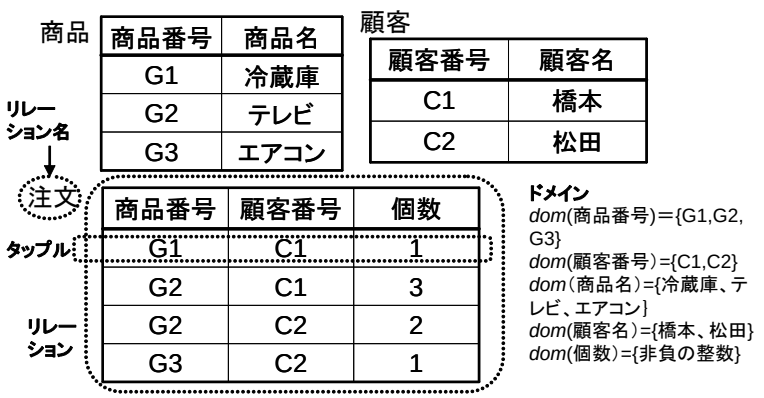


図 17: リレーショナルデータモデルの構造

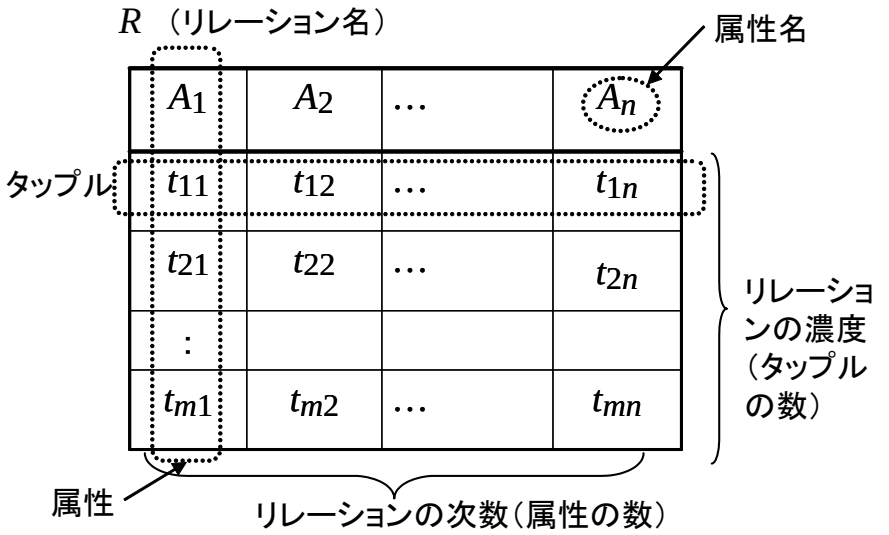


図 18: リレーションの構造

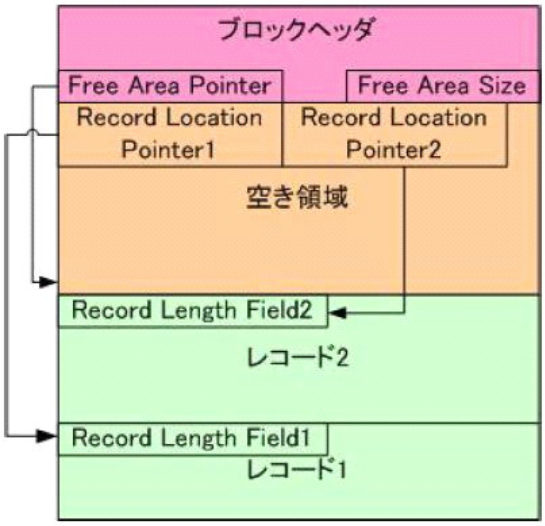


図 19: リレーショナルデータモデルの実装

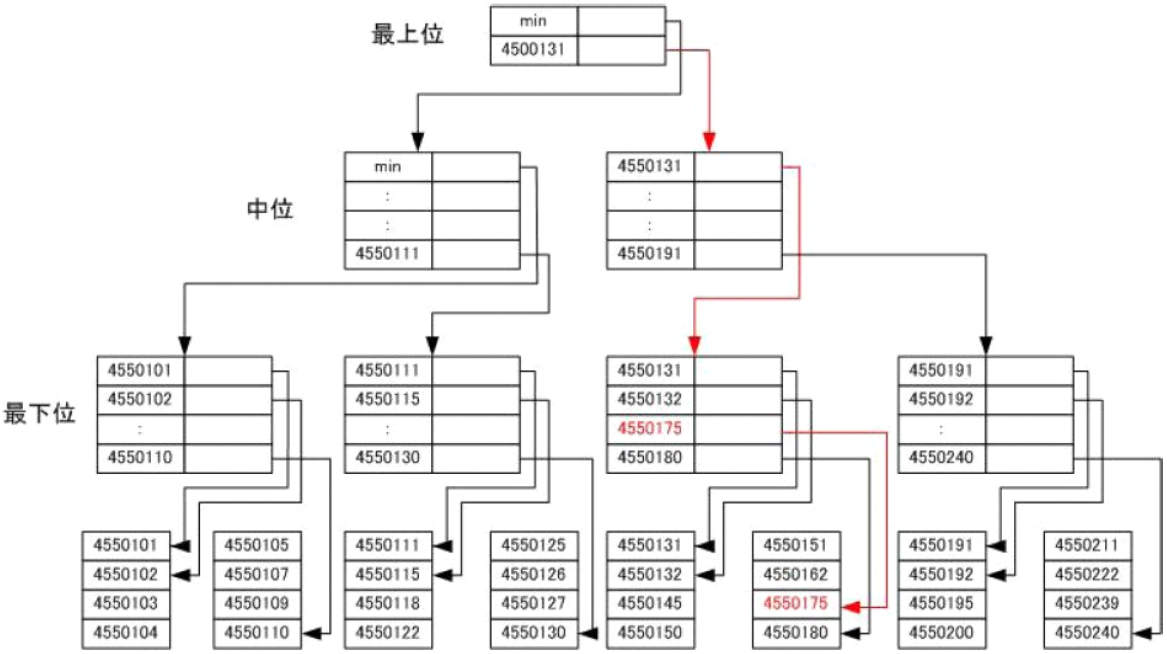


図 20: B 木によるインデックスの表現

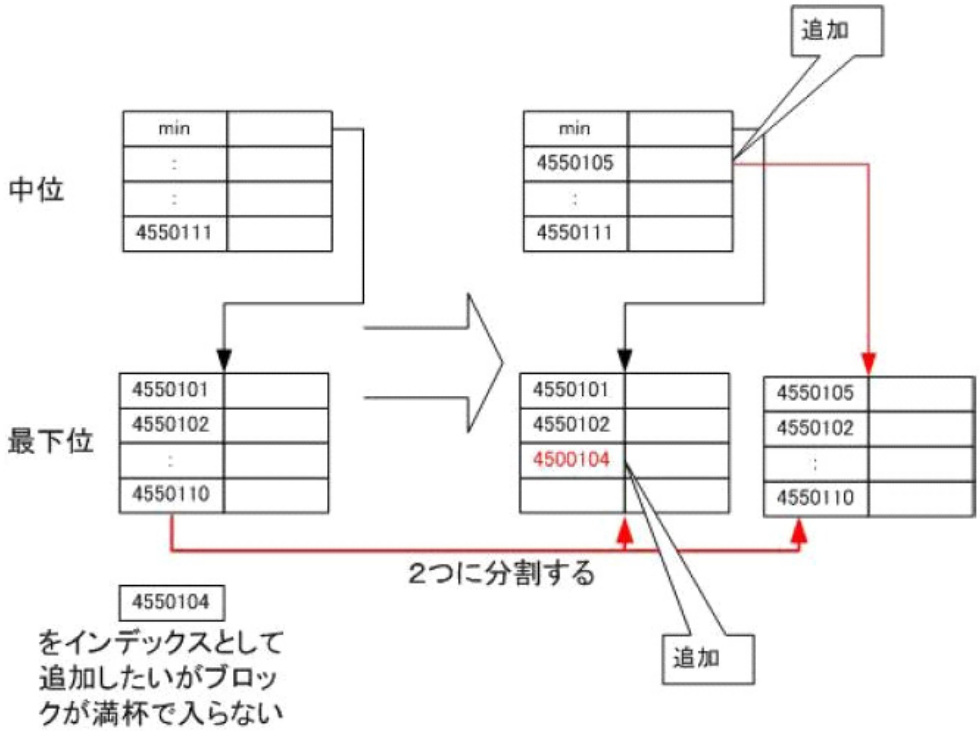


図 21: レコードの挿入時のインデックスの更新

タップルの検索を効率的に行うため，属性の値のインデックスを B 木 (B-tree) で構成する（図 20 参照）．B 木は各葉接点の深さがバランスするように構成されており，インデックスへの挿入では図 21 のようにしてバランスを維持するようにしている．

3.4 リレーショナルデータモデルでのリレーションの条件

リレーショナルデータモデルでは，すべてのリレーションは第一正規形でなくてはならない．

第一正規形

あるリレーションを構成しているすべてのドメインが単純であるとき，そのリレーションは第一正規形であるという．

単純なドメインの定義：

(1) そのドメインが他のドメインの直積（の部分集合）ではなく，かつ，
(2) そのドメインが他のドメインの巾集合（の部分集合）ではない．

正規化：

- 正規形でないリレーションを正規形に変換．
- 直積および巾集合を分解する．

正規化の例：（図 22，図 23 参照）

- 直積の正規化：郵便番号，都道府県，市町村などのデータの組で表されている住所をそれぞれの項目ごとに別の属性として分解する（図 22 参照）．
  あるいは，(緯度，経度) の組で表されている座標を，緯度と経度の二つの属性に分解する（注：緯度，経度はさらに分解することが考えられる．例えば，N31 が北緯 31 度を表現しているとすると北緯を表す N と緯度を表す 31 とをさらに分解すれば「北半球にある都市をすべて求めよ」などの問合せが属性「南北」を使ってできるという点で望ましいといえる）．直積の正規化は属性の分解であり，リレーションの次数（属性の数）は増えるが，リレーションの濃度（タップルの数）は変わらない．
- 巾集合の正規化：世帯のように集合値を取る属性を，世帯の構成員一人一人について 1 タップルとなるように分解する（図 23 参照）．
  リレーションの濃度（タップルの数）が増えるが，リレーションの次数（属性の数）は変わらない．

3.5 リレーションスキーマとインスタンス

リレーションは，その構造を規定しているリレーションスキーマと，その内容であるインスタンスに区別して表現されることがある．

リレーションスキーマ $\mathbf{R}$ リレーション R の構造を規定するものであり，時間の経過によらず不変なドメイン，属性，キー，従属性などの情報を指す．

$\Omega_{\mathbf{R}}$ ：リレーションスキーマ $\mathbf{R}$ の全属性集合

インスタンス ：リレーションの内容を指す．時間とともに変化するので，厳密には R_t（時刻 t でのリレーション R の内容）というように時刻を指定した形で表現する．

住所（正規化前）

氏名	住所
大阪太郎	560-8531大阪府豊中市待兼山町
神戸次郎	657-8501兵庫県神戸市灘区六甲台町

⇩

住所（正規化後）

氏名	郵便番号	都道府県	市区	町村
大阪太郎	560-8531	大阪府	豊中市	待兼山町
神戸次郎	657-8501	兵庫県	神戸市灘区	六甲台町

図 22: 正規化の例（直積）

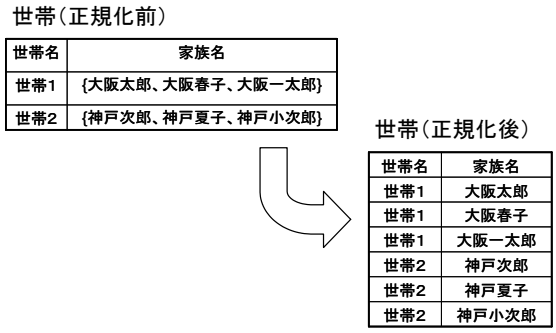


図 23: 正規化の例（巾集合）

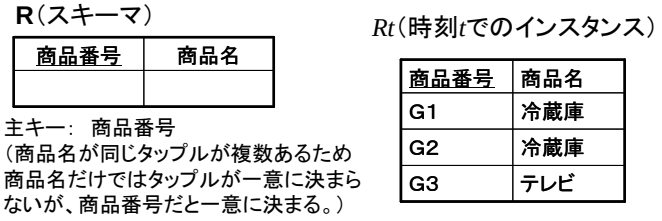


図 24: リレーションスキーマとインスタンスの例

3.6 候補キーと超キー，主キー

リレーションスキーマ $\mathbf{R}$ の属性集合である K が以下の条件 (1), (2) を満たしたとき，K は $\mathbf{R}$ の**候補キー**と呼ばれる．

(1) R をリレーションスキーマ $\mathbf{R}$ の任意のインスタンスとすると $(\forall t, t' \in R)(t[K] = t'[K] \Rightarrow t = t')$. が成り立つ．

(2) K から属性を 1 個でも欠落させると (1) が成立しない．

補足：

- (1) の条件だけが成り立つ属性集合は，**超キー** (または**スーパキー**) と呼ばれる．
- 候補キーから意味的に最も適切なものを 1 個選んで**主キー**とする（例えば,「大学生」のリレーションを考えると，候補キーとしては学籍番号，保険番号，電話番号などが考えられるが，大学生を一意に特定する候補キーとしては学籍番号が最も適切であるため，これを主キーとする）．
- 全属性集合 $\mathbf{\Omega_R}$ は必ず超キーとなる ($\mathbf{R}$ の次数より少ない数の属性からなる候補キーが全くない場合には，全属性集合 $\mathbf{\Omega_R}$ が候補キーとなる)．
- キーには，この他に**外部キー**がある（7.2.3 節参照）．

4 リレーショナルデータベースの操作言語 (リレーショナル代数)

リレーショナルデータベースを操作するための言語としては，リレーショナル代数と SQL がある．前者はリレーショナルデータベースの操作を理論的に説明するための言語であり，計算機上で実装するためのものではない．それに対して，後者は計算機上で実装されるための言語であり，多くの処理系が存在する．ここではリレーショナル代数について説明する．

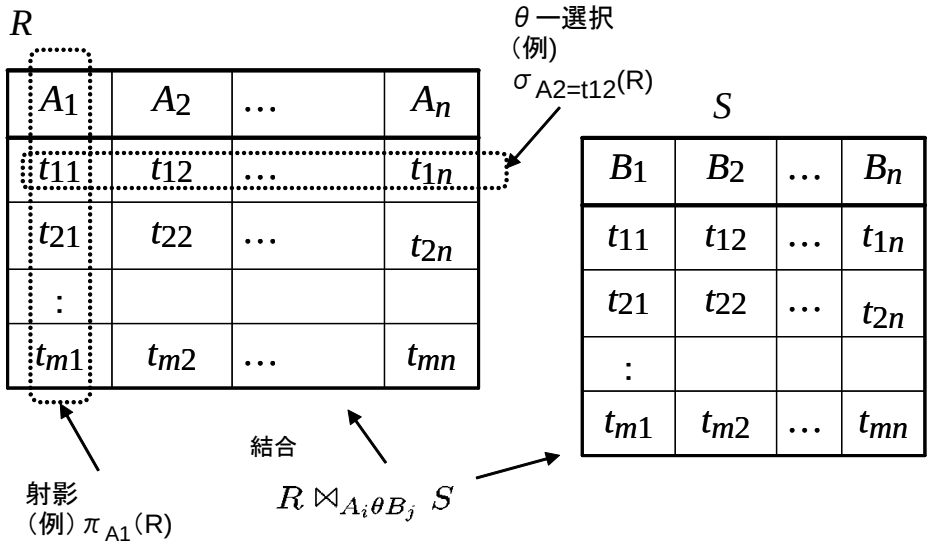


図 25: リレーショナル代数に特有の演算

4.1 集合演算

- 和両立な 2 個のリレーションにおいて可能な集合演算
  リレーションのスキーマ $\mathbf{R}(A_1, A_2, \cdots, A_n)$ と $\mathbf{S}(B_1, B_2, \cdots, B_m)$ が**和両立**とは，次の 2 個の条件を満たしているときをいう．
  (1) $n = m$
  (2) 各 $i(1 \leq i \leq n)$ に対して，$dom(A_i) = dom(B_i)$ が成立する．

  和集合演算 $R \cup S \stackrel{def}{=} \{t | t \in R \vee t \in S\}$
  差集合演算 $R - S \stackrel{def}{=} \{t | t \in R \wedge t \notin S\}$
  共通集合演算 $R \cap S \stackrel{def}{=} \{t | t \in R \wedge t \in S\}$
- 任意の 2 個のリレーションにおいて可能な集合演算
  (拡張型) 直積集合演算 $R \times S \stackrel{def}{=} \{(r, s) | r \in R \wedge s \in S\}$

4.2 リレーショナル代数に特有の演算

リレーショナル代数に特有の演算には次のようなものがある (図 25 参照)．以下で，リレーションスキーマは $\mathbf{R}(A_1, A_2, \cdots, A_n)$ と $\mathbf{S}(B_1, B_2, \cdots, B_m)$ となっている．

射影演算 $\pi_X(R) \stackrel{def}{=} \{u | t \in R \wedge u = t[X]\}$

$\theta-$ 選択演算 $\sigma_{A_i \theta A_j}(R) \stackrel{def}{=} \{t | t \in R \wedge t[A_i]\ \theta\ t[A_j]\}$
(ただし，A_i と A_j は $\theta-$ 比較可能)

$\theta-$ 結合演算 $R \bowtie_{A_i \theta B_j} S \stackrel{def}{=} \{v | v \in R \times S \wedge v[R.A_i]\ \theta\ v[S.B_j]\}$
または，
$R \bowtie_{A_i \theta B_j} S \stackrel{def}{=} \sigma_{R.A_i\ \theta\ S.B_j}(R \times S)$
(ただし，A_i と B_j は $\theta-$ 比較可能)

自然結合演算 $R \bowtie S \stackrel{def}{=} \pi_{R.A_1, \cdots, R.A_n, S.D_l, \cdots, S.D_{m-l}}(\sigma_{R.C_1 = S.C_1, \cdots, R.C_l = S.C_l}(R \times S))$

商演算 $R \div S \stackrel{def}{=} \{v | v \in \pi_{A_1, \cdots, A_{n-m}}(R) \wedge (\forall u \in S)((v, u) \in R)\}$
または，
$R \div S \stackrel{def}{=} \pi_{A_1, \cdots, A_{n-m}}(R) - \pi_{A_1, \cdots, A_{n-m}}((\pi_{A_1, \cdots, A_{n-m}}(R) \times S) - R)$

(注意) 各演算の対象はタップル集合なので，演算の結果により同一リレーション中で重複するタップルが生じたときには，1 個を残して残りのタップルは取り除かれる．

自然結合演算は，等結合演算（$\theta-$ 結合演算の θ が $=$ になったもの）と結合演算としての動作は同じだが，演算結果で重複した属性が除去される点が等結合演算と異なる（以下の例を参照）．
(参考)2 個のリレーションについて直積演算を行った後で商演算を行えば元のリレーションに戻ることが知られている（ $(R \times S) \div S = R$ ）．このため，商演算は直積演算の逆演算と見ることができる．

4.3 リレーショナル代数に特有の演算の実行例

以下では次の 2 個のリレーション emp と dept を例として使う．

emp

empno	ename	deptno	salary
E01	Smith	D01	100
E02	Morgan	D01	80
E03	Robert	D01	90
E04	Washington	D02	120
E05	Lincoln	D02	100

dept

deptno	dname	manager
D01	Account	E01
D02	Personnel	E04

射影演算

(1) リレーション emp から deptno の一覧を求めよ．

$\pi_{\texttt{deptno}}(\texttt{emp})$

結果は，以下の通り（重複が取り除かれることに注意）．

deptno
D01
D02

選択演算

(2) リレーション emp から属性 salary の値が 100 以上の全データを求めよ．

$\sigma_{\texttt{salary} \geq 100}(\texttt{emp})$

結果は，以下の通り．

empno	ename	deptno	salary
E01	Smith	D01	100
E04	Washington	D02	120
E05	Lincoln	D02	100

結合演算

(3) 2 個のリレーション emp と dept を，属性 deptno が等しいという条件で結合せよ．

$\texttt{emp} \bowtie_{\texttt{emp.deptno=dept.deptno}} \texttt{dept}$

結果は，以下の通り．

emp .empno	emp .ename	emp .deptno	emp .salary	dept .deptno	dept .dname	dept .manager
E01	Smith	D01	100	D01	Account	E01
E02	Morgan	D01	80	D01	Account	E01
E03	Robert	D01	90	D01	Account	E01
E04	Washington	D02	120	D02	Personnel	E04
E05	Lincoln	D02	100	D02	Personnel	E04

（注意）上記結果の各属性名は，例えば一番左の属性は emp.empno であるが，紙面の関係で 2 段に分けて表現している．

自然結合演算

(4) 2 個のリレーション emp と dept を 1 個のリレーションに結合せよ.

emp ⋈ dept

結果は，以下の通り（(3) との違いに注意）.

empno	ename	deptno	salary	dname	manager
E01	Smith	D01	100	Account	E01
E02	Morgan	D01	80	Account	E01
E03	Robert	D01	90	Account	E01
E04	Washington	D02	120	Personnel	E04
E05	Lincoln	D02	100	Personnel	E04

商演算

この演算に限り以下のリレーション store と shopping_list を例として使う.

store

store_name	goods
A	bread
A	butter
A	milk
B	bread
B	jam
C	coffee

shopping_list

goods
bread
butter
milk

(5) リレーション store 中で属性 store_name が同じタップル集合のうちで，リレーション shopping_list 中のタップルが持つすべての値の組を含むようなタップルの属性 store_name の値を求めよ.

store ÷ shopping_list

結果は，以下の通り.

store_name
A

5 リレーショナルデータベースの操作言語 SQL

5.1 SQL の概要

- SQL は，現在最も標準的なリレーショナルデータベース言語で，多くのデータベース管理システムの処理系で採用されている.
- 規格として制定された時期の違いにより，SQL89(または SQL1), SQL92(または SQL2), SQL99(または SQL3), SQL:2003, SQL:2006, SQL:2008, SQL:2011 などの規格がある.
- 実際のリレーショナルデータベース管理システムでは、SQL:1999 に準拠した実装が多い.
- データ操作言語 (DML) だけでなくデータ定義言語 (DDL) の機能も持っている.
- データの基本単位は**表**（タップルのマルチ集合）である（リレーショナル代数ではデータの基本単位はリレーション（タップルの集合）であったことに注意）.
- リレーションの属性を SQL では**列** (column) と呼ぶ.
- タップルの成分として値のない（もしくは値がまだ決まっていない）状況を**空値**として NULL で表現できる（文字列の 'NULL' とは異なることに注意）.

5.2 SQL での問合せ指定

SQL では問合せ指定を英語に似た宣言的な記述である select 文で表現される（図 26 参照）.

```
select [all | distinct] <列リスト> from <表リスト>
[where <検索条件>]
[group by <グループ化する列のリスト>] [having <タップルの選択条件>]
[[union [all] | intersect | except] 副問合せ]
[order by <ソート条件> [asc | desc]]
```

一般形
select 列リスト from 表リスト where 検索条件
- 実際の処理手順は from-where-selectの順で考えたほうが理解しやすい。リレーショナル代数と対応させると次のようになる。

例 select R.A from R,S where R.A=S.B
from R, S ・・・・・・・・R×S（RとSの直積）
where R.A=S.B・・・・$\sigma_{R.A=S.B}$（選択演算）
select R.A ・・・・・・・・$\pi_{R.A}$(射影演算）

図 26: SQL の構文

[ ] は省略可能である項目を表す．また，[a | b] とあるのは，この項目の位置で a または b または省略することが選択できる（省略した場合は a を書いたと見なされる）ことを示す．

select 文では，<表リスト>で指定された表（複数の表を指定して良い）から，<検索条件>を満たすタップルをすべて選択し，その中の<列リスト>で指定された成分のみを取り出して表にまとめたものが，結果として出力される．

select 文で得られる結果も表であるが，<表リスト>に直接 select 文を書くことはできない（ただし，Oracle など SQL99 に準拠した一部の処理系では書けるように拡張されている）．

select 文の中で，さらに別の select 文の結果を使いたければ，以下の (4) 部分問合せ (入れ子型問合せ) のように<検索条件>で，比較演算子と組み合わせたり，in 述語や exists 述語を利用することで表現できる．

<検索条件>では，次のような比較演算子を利用することができる．

```
=, <>, >, <, <=, >=, like
```

ここで like は部分文字列比較の演算子であり，like "emp%" で emp で始まる文字列，like "%loyee" で loyee で終わる文字列，like "%loy%" で loy を含む文字列との比較であれば結果が真となる．

また，これらの比較演算子の式は，AND や OR でつなぐことができる．

5.3 SQL での問合せの種類

以下では次の 2 個の表 emp と dept を例として使う．

emp

empno	ename	deptno	salary
E01	Smith	D01	100
E02	Morgan	D01	80
E03	Robert	D01	90
E04	Washington	D02	120
E05	Lincoln	D02	100
E93	Matsuda	D91	90

dept

deptno	dname	manager
D01	Account	E01
D02	Personnel	E04

(1) 単純問合せ（リレーショナル代数の射影や選択の演算に相当する問合せ）

(1-1) 表 emp から全データを求めよ．

```
select * from emp
```

これは以下と等価．

```
select empno, ename, deptno, salary from emp
```

結果は，以下の通り．

empno	ename	deptno	salary
E01	Smith	D01	100
E02	Morgan	D01	80
E03	Robert	D01	90
E04	Washington	D02	120
E05	Lincoln	D02	100
E93	Matsuda	D91	90

(1-2) 表 emp から deptno の一覧を求めよ.

```
select deptno from emp
```

結果は，以下の通り.

deptno
D01
D01
D01
D02
D02
D91

リレーショナル代数の射影演算 $\pi_{\texttt{deptno}}(\texttt{emp})$ と同様，重複したタップルを取り除いた結果が得たければ以下のようにする.

```
select distinct deptno from emp
```

結果は，以下の通り.

deptno
D01
D02
D91

(1-3) 表 emp から salary が 100 以上の全データを求めよ.

```
select * from emp where salary >= 100
```

これは，リレーショナル代数の $\sigma_{\texttt{salary}\geq 100}(\texttt{emp})$ と等価な問合せである.

結果は，以下の通り.

empno	ename	deptno	salary
E01	Smith	D01	100
E04	Washington	D02	120
E05	Lincoln	D02	100

(2) 集約演算とグループ化を使う問合せ

SQL では，問合せの結果として得られる列に対して最小や最大，総和，平均，要素の個数などの演算を行うことができ，これらは**集約演算** (aggregation operation) と呼ばれる．集約演算はそれぞれ，avg（平均），sum （総和），max（最大），min（最小値），count（要素の個数）で指定される.

集約演算では，問合せの結果全体に対して演算するだけでなく，特定の列の項目ごとに演算することもできる（例えば，所属ごとの給与の平均値，試験科目ごとの点数の平均値など）．SQL では group by を使った**グループ化**により項目ごとの集約演算が行える.

さらに having によりグループ化する列の項目に条件を付けることができる（例えば，従業員が 3 人以上いる所属，受験者が 5 人以上いる試験科目など）.

(2-1) （全タップルについての集約演算）表 emp から deptno が D01 であるタップルについて，salary の平均値を求めよ.

```
select avg(salary) from emp where deptno = 'D01'
```

結果は，以下の通り.

avg(salary)
90

(2-2) （グループ化による集約演算）表 emp から deptno の値でグループ化したときの，salary の平均値を求めよ.

```
select deptno, avg(salary) from emp group by deptno
```

結果は，以下の通り．

deptno	avg(salary)
D01	90
D02	110
D91	90

(2-3)（having によるグループ化する条件の指定）表 emp からタップルが 3 個以上ある deptno について，その deptno の salary の平均値を求めよ．

```
select deptno, avg(salary) from emp group by deptno
having count(*) >= 3
```

結果は，以下の通り．

deptno	avg(salary)
D01	90

(3) 結合問合せ

結合問合せとは，複数の表を指定して，それらの表にまたがって検索条件を満たすタップルを求める問合せである．指定された複数の表のタップルの直積が取られた後，検索条件を満たすものだけが出力される．

リレーショナル代数の自然結合演算は，結合問合せで検索条件と列リストを指定することで等価な結果を得ることができるが，2 個のリレーションの自然結合演算で比較する属性名（SQL では表の列名）が同じときには，SQL の**自然結合** (natural join) によっても求めることができる．

等結合（複数の表の列の値が等しいことを条件として行う結合）の問合せで，条件で指定された列同士で，一方の表にある値が，別の表には存在しない場合，通常の結合問合せでは検索条件不成立とみなされるが，これでは結合問合せの結果からそのタップルの値が消えてしまう．このような時でも，**外結合** (outer join) を行うことにより，存在しないところは空値にして結合結果を出すことで，元の表からのタップルの欠損を防ぐことができる（以下の (3-2) の例を参照）．外結合には，結合する左側の表のみのタップルの欠損を防ぐ**左外結合** (left outer join)，右側の表のみのタップルの欠損を防ぐ**右外結合** (right outer join)，両側の表のタップルの欠損を防ぐ**完全外結合** (full outer join) がある．

問合せが複数の表にまたがるため，列名を指定するときは，その列が属している表の名前を前につけて示す必要がある（例えば、表 emp の中の列 ename であれば，emp.ename と書く）．

同じ表を 2 回以上使ってそれらの表にまたがる結合問合せを行うこともできるが，表名を書く方法では列の区別が付かない．このようなときは，表の名前として**別名**を使うことで，同じ名前の表でも 1 回目の参照か 2 回目の参照かなどを区別できる（以下の例を参照）．

(3-1) 表 emp と表 dept とから，deptno の dname が Account である者の ename を求めよ．

```
select emp.ename from emp, dept
where emp.deptno = dept.deptno and
      dept.dname = 'Account'
```

（参考）SQL:1999 から以下の構文が導入されている．

```
select emp.ename from emp join dept
on emp.deptno = dept.deptno
where dept.dname = 'Account'
```

(3-1) の問合せは，別名を使って次のようにも書ける（a，b が別名）．

```
select a.ename from emp a, dept b
where a.deptno = b.deptno and
      b.dname = 'Account'
```

（参考）上の問合せは，リレーショナル代数の

$$\pi_{\texttt{emp.ename}}(\sigma_{\texttt{dept.dname='Account'}}(\texttt{emp} \bowtie_{\texttt{emp.deptno=dept.deptno}} \texttt{dept}))$$

と等価である．

結果は，以下の通り．

ename
Smith
Morgan
Robert

(3-2) 表 emp と表 dept をリレーションと考え，自然結合演算により結合せよ．
この問合せは，検索条件で deptno が同じという条件を書くことで，

```
select emp.*, dept.dname, dept.manager from emp, dept
where emp.deptno = dept.deptno
```

で表現でき，結果は次の通りとなる．

empno	ename	deptno	salary	dname	manager
E01	Smith	D01	100	Account	E01
E02	Morgan	D01	80	Account	E01
E03	Robert	D01	90	Account	E01
E04	Washington	D02	120	Personnel	E04
E05	Lincoln	D02	100	Personnel	E04

自然結合演算で比較する列名が表 emp と表 dept で同じ名前（この場合はどちらも deptno）のときは，SQL:1999 で導入された構文を使って次のように表現できる．

```
select * from emp natural join dept
```

ただし，結果で出力される列の順番は，必ずしも元の表の列の順番とはなるとは限らない（次のように，2 個の表に共通する列名 deptno が最初にくる）．

deptno	empno	ename	salary	dname	manager
D01	E01	Smith	100	Account	E01
D01	E02	Morgan	80	Account	E01
D01	E03	Robert	90	Account	E01
D02	E04	Washington	120	Personnel	E04
D02	E05	Lincoln	100	Personnel	E04

natural join を使うときには，表の結合が先に行われてから出力する列名を見るため，

```
select emp.*, dept.* from emp natural join dept
```

のように列名で表を指定することはできないことに注意．

(3-3) 表 emp と表 dept とから，ename と dname の対応一覧表を作成せよ．
この問合せは，通常の等結合により，

```
select a.ename, b.dname from emp a, dept b
where a.deptno = b.deptno
```

で表現できる．また，natural join を使っても次のように表現できる．

```
select ename, dname from emp natural join dept
```

このとき，結果は次の通りとなる．

ename	dname
Smith	Account
Morgan	Account
Robert	Account
Washington	Personnel
Lincoln	Personnel

しかし，等結合では表 emp にある Matsuda のタップルが消えてしまう．結果から元の表のタップルを欠損させたくない場合は，外結合を使うことによりもう一方の表で対応するタップルのないところを空値に置き換えて出力することで欠損を補って結果を出力することができる．

```
select a.ename, b.dname
  from emp a left outer join dept b on a.deptno = b.deptno
```

（注意）この例は左の表の欠損を防ぐ左外結合を表す（右外結合は `right outer join` で表す）.

結果は，以下の通り（NULL は実際には表示されない）.

ename	dname
Smith	Account
Morgan	Account
Robert	Account
Washington	Personnel
Lincoln	Personnel
Matsuda	NULL

(4) 部分問合せ (入れ子型問合せ)

検索条件の中で別の select 文（**副問合せ**と呼ばれる）を指定して，その結果を使って検索できる．副問合せに対してそれを検索条件に含んでいる元の問合せを**主問合せ**と呼ぶ.

部分問合せには，検索条件の中で副問合せの結果をどのように扱うかに応じて以下のような種類がある.

- 比較演算子との組合せ

副問合せの select 文の結果が単一の値であるときには，副問合せと比較演算子を組み合わせて部分問合せを構成できる.

(4-1) 表 emp と表 dept とから，dname が Account である dept の manager の salary を求めよ.

```
select salary from emp
where empno =
      (select manager from dept where dname = 'Account')
```

- `in` 述語の利用

副問合せの select 文の結果がある列の値の集合であり，その中のどれかと一致する値を成分に持つタップルを求めたいときは，副問合せと in 述語とを組み合わせて検索することができる.

(4-2) 表 emp と表 dept とから，deptno の dname が Account である者の ename を求めよ（問合せ (3-1) を部分問合せにより表現）.

```
select ename from emp
where deptno in
      (select deptno from dept where dname = 'Account')
```

（注意）`in` 述語が使えるのは 1 個の列に対してだけである．また，副問合せの結果のどれとも異なる成分を持つタップルは `not in` と `in` の前に `not` を使うことにより検索できる.

- `exists` 述語の利用

副問合せの select 文の結果が表であり，その表中のタップルとある条件を満たすタップルを求めたいときは，副問合せと exists 述語とを組み合わせて検索することができる.

exists 述語を使うと 2 個の表から共通集合を求めることができる．具体的な質問の例は集合演算のところで述べる.

(5)SQL による数値演算

SQL では，他のプログラミング言語と同様，数値演算を表現できる（列リストおよび検索条件中で数式として記述できる）.

- 算術演算: +（加算），-（減算），*（乗算），/（除算），mod(x,y)（x を y で割った余り）
- 数値関数: ln(x)（x の自然対数），log(x,y)（x を底とする y の対数），三角関数など

（例）
```
select empno,salary-20 from emp
      where salary >= 100
```

（注意）集約演算は数値関数ではないので，列リストのみで使え，検索条件では使えない.
集約演算が使えない例（平均以下の給与をもらっている社員を求める）

```
select empno, salary from emp
where salary < avg(salary)
```

この例のような問合せは，SQL では次のようにすれば記述できる．

```
select empno, salary from emp
where salary < (select avg(salary) from emp)
```

(6)SQL による空値の取扱

SQL の論理式は，真と偽以外に未定義の値も取る **3 値論理**である．
次の 2 種類の問合せの結果は同じにならない可能性がある．

```
(Q1) select count(*) from ref

(Q2) select count(*) from ref
     where startpage < 100 or startpage >= 100
```

startpage が値として必ず数値を取れば (Q1) と (Q2) の結果は同じになるが，もし空値があると，where 以下の条件式の論理値が未定義となり，条件が成立しないと見なされる．
このようなことを防ぐには，x is null （x が空値のとき真），x is not null （x が空値のとき偽）などの条件式を使えばよい．
次の (Q3) の結果は，(Q1) の結果と同じになる．

```
(Q3) select count(*) from ref
     where startpage is null or startpage < 100 or startpage >= 100
```

5.4 SQL による集合演算

以下では 5.2 節で例にあげたリレーション emp と同じスキーマを持つ 2 個の表 emp1 と emp2 を考える．

(1) 和集合演算

```
[一般形] <問合せ指定> union <問合せ指定> [union <問合せ指定>]
[例]  select * from emp1 union select * from emp2
```

(注意)union は集合演算の演算子なので，distinct がなくても，さらに例え元の表に重複したタップルがあっても問合せ結果から重複が取り除かれる．重複を取り除きたくなければ，union の後に all を指定して union all とすれば 2 個の表を単純に併合したものが得られる．

```
[例]  select * from emp1 union all  select * from emp2
```

(2) 共通集合演算

```
[一般形] <問合せ指定> intersect <問合せ指定> [intersect <問合せ指定>]
[例]  select * from emp1 intersect select * from emp2
```

共通集合演算は，exists 述語を使った部分質問により次のようにも記述できる．

```
[一般形]  select * from <表名 1> where exists
            (select * from <表名 2> where <表名 1>.<列名 1>=<表名 2>.<列名 2>...)

[例]  select * from emp1 where exists
        (select * from emp2 where emp1.empno = emp2.empno)
      または，別名を使うことにより，
      select * from emp1 x where exists
        (select * from emp2 y where x.empno = y.empno)
```

R

X	Y
a	1
a	2
b	1

S

Y
1
2

(a)

X	Y
a	1
a	2
b	1
b	2

(b)

X	Y
b	2

(c)

X
b

(d)

X
a

図 27: 商演算実行の経過

（注意 1）exists 述語は副問合せの結果が「存在」するときに主問合せの結果を取り出すという働きがある．exists 述語を持つ部分問合せでは，主問合せで表リストで指定した表のタップルと副問合せで表リストで指定した表のタップルのすべての組合せを比較して，副問合せの検索条件を満足するときの主問合せ側のタップルを結果として出力する．
（注意 2）共通集合を求めるのに exists 述語を使うときは，副問合せの where 以下は 2 個の表のタップルを識別するのに必要な条件比較が並ぶ．上の例では，empno が emp1 と emp2 のキーであると考えており，この列のみ比較している．

(3) 差集合演算

[一般形] <問合せ指定> except <問合せ指定> [except <問合せ指定>]
[例] select * from emp1 except select * from emp2

(注意)intersect，except は結果から常に重複を取り除く．重複を取り除かないような intersect all，except all といった指定はできない．

差集合演算は，exists 述語を使った部分質問により次のようにも記述できる．

```
[一般形]  select * from <表名 1> where not exists
            (select * from <表名 2> where <表名 1>.<列名 1>=<表名 2>.<列名 2>...)

[例]  select * from emp1 where not exists
        (select * from emp2 where emp1.empno = emp2.empno)
      または，別名を使うことにより，
      select * from emp1 x where not exists
        (select * from emp2 y where x.empno =y.empno)
```

(4) 直積演算

```
[一般形]  select <表名 1>.<列名 1>, <表名 2>.<列名 2> from <表名 1>, <表名 2>
[例]  select emp1.ename, emp2.deptno from emp1, emp2
      または，別名を使うことにより，
      select x.ename, y.deptno from emp1 x, emp2 y
```

（注意）同一の表の中で直積を取るときは，表名では区別できないため，常に別名を使う必要がある．

```
[例]  select x.ename, y.deptno from emp1 x, emp1 y
```

(5) 商演算

商演算は定義から明らかなように直積演算，差集合演算，射影演算があれば実現できる．図 27 の 2 個の表 R と S をリレーションと見たときの $R \div S$ は，SQL では以下のように表現できる．

```
select distinct a.X from R a
  except select distinct e.X from
          (select distinct b.X, c.Y from R b, S c
             except select d.* from R d) e
```

解説 図 27 の R, S に対する商演算 $R \div S$ は，以下のリレーショナル代数式で表される．

$R \div S = \pi_X(R) - \pi_X((\pi_X(R) \times S) - R)$

これを SQL に変換することを考える．まず，最も内側の演算 $\pi_X(R) \times S$ は，次の等価な SQL 文 (1) に変換でき，上記の例の場合，結果は図 27(a) のようになる．

```
  select distinct b.X, c.Y from R b, S c                    (1)
```

次に，そのすぐ外側の差集合演算を組み合わせると，次の SQL 文 (2) になり，結果は図 27(b) のようになる．

```
select distinct b.X, c.Y from R b, S c                      (2)
  except select d.* from R d
```

さらに (2) の結果に射影 π_X を取るが，これは次のように列名の指定を付けるだけで良く，SQL 文は (3) のようになり，結果は図 27(c) のようになる．

```
select distinct e.X from
  (select distinct b.X, c.Y from R b, S c                   (3)
     except select d.* from R d) e
```

最後に $\pi_X(R)$ と (3) の結果との差集合を取る．SQL 文は (4) のようになり，結果は図 27(d) のようになる．

```
select distinct a.X from R a
  except select distinct e.X from                           (4)
         (select distinct b.X, c.Y from R b, S c
            except select d.* from R d) e
```

(注意) 上記では集合演算子 except を使っているため，distinct が全くなくても問合せ結果では重複が取り除かれるため，最終的な問合せの SQL 文では distinct を省略できるが，途中の説明の都合上省略せずに書いている．

5.5 SQL によるデータベース更新操作

(1) 挿入 (insert 文)

[一般形] insert into <表名> values (<挿入値リスト>|<問合せ指定>)

```
[例]  insert into emp values ('E01', 'Smith', 'D01', 100)
      insert into R1 values select X, Y from R2
```

(2) 削除 (delete 文)

[一般形] delete from <表名> where <探索条件>

```
[例]  delete from emp where ename = 'Smith'
```

(3) 修正 (update 文)

[一般形] update <表名> set <設定> where <探索条件>

```
[例]  update emp set deptno = 'D02' where empno = 'E01'
      update emp set salary = salary-10 where deptno = 'D01'
```

5.6 表の定義

表の定義文（create table 文）

[一般形] create table <表名> (<列定義>[, <列定義>...])

```
[例]  create table emp
        (empno char(3), ename varchar2(15),
         deptno char(3), salary number(6))
```

（参考）<列定義>で使われるデータの型と記述方法については，表 1 を参照．
定義した表を削除するには drop table 文，表の定義を修正するには alter table 文をそれぞれ使用する．

表 1: データの型と記述方法

データ型	説明	データ例
char(サイズ)	固定長の文字データ。サイズ不足分は空白で埋める。サイズは1～2000	'ABC　'
varchar2（サイズ)	可変長の文字データ。サイズは1～4000	'ABC'
number(p,s)	数値データ。精度pと位取りsを指定しない場合は浮動小数点（38けた）	123、123.45
date	日付データ。4けたの年、月、日、時間、分、秒を紀元前4712年1月1日から西暦9999年12月31日まで扱える	'2006-01-15 17:15:30'

注文商品

商品番号	商品名	個数
G1	冷蔵庫	1
G2	テレビ	5
G3	エアコン	1

図 28: ビューの例

5.7 ビュー (視点) の定義

ビューの定義文（create view 文）
　ビューにより，3 層スキーマモデルにおける外部スキーマを実現することができる．

[一般形] `create view <ビュー名> as <問合せ指定> [with check option]`

```
[例]  create view 注文商品
        as select 商品. 商品番号, 商品. 商品名, sum(個数) from 商品, 注文
        where 商品. 商品番号 = 注文. 商品番号
        group by 商品. 商品番号, 商品. 商品名
```

　結果は，図 28 のようになる．

```
[例]  create view high_salary
        as select * from emp where salary >= 100
        with check option
```

（注意） with check option を付けると<問合せ指定>の中にある探索条件に合わない以下のような更新操作を受け付けないようにすることができる．

```
  update high_salary set salary = 50 where empno = 'E01'
```

5.8 問合せ処理と最適化

　SQL などのデータ操作言語により記述された問合せは，DBMS 中のデータ操作言語処理系により解析され，基本データ操作からなる一連の手続きが生成される．このデータ操作手続きのことを，**問合せ実行プラン** (query execution plan) と呼ぶ．問合せ実行プランに記述された基本データ操作を処理系が順次実行することで問合せは処理される．
　問合せ実行プランの生成は，データ操作言語による論理的かつ非手続き的問合せ記述から，物理的かつ手続き的問合せ記述への変換の過程と考えることができる．これは，次の 2 つのフェーズで行われる．

フェーズ１：SQL などのデータ操作言語によって記述された問合せを，リレーショナル代数演算子列へ変換.
フェーズ２：各リレーショナル代数演算あるいはその組合せのデータ操作を，具体的に実行する基本データ操作列へ変換.

例：次の２つの表についての問合せ処理を考える.

emp

empno	ename	deptno	salary
E01	Smith	D01	100
E02	Morgan	D01	80
E03	Robert	D01	90
E04	Washington	D02	120
E05	Lincoln	D02	100
E91	Matsuda	D91	90

dept

deptno	dname	manager
D01	Account	E01
D02	Personnel	E04
D91	Personnel	E91

問合せ

```
select distinct emp.empno, emp.ename, emp.deptno, dept.dname
from emp, dept
where emp.deptno = dept.deptno and emp.empno = 'E01'
```

この問合せに対する処理のフェーズ１で生成され得る演算子列としては，次のようなものが考えられる（以下では，emp を a，dept を b という別名で表現している）.

(1) $\pi_{a.empno,a.ename,a.deptno,b.dname}(\sigma_{a.deptno=b.deptno \wedge a.empno='E01'}(a \times b))$
(2) $\pi_{a.empno,a.ename,a.deptno,b.dname}(\sigma_{a.empno='E01'}(a \bowtie_{a.deptno=b.deptno} b))$
(3) $\pi_{a.empno,a.ename,a.deptno,b.dname}($
$\pi_{a.empno,a.ename,a.deptno}(\sigma_{a.empno='E01'}(a)) \bowtie_{a.deptno=b.deptno} \pi_{b.deptno,b.dname}(b))$

フェーズ２では，さらに各リレーショナル代数演算あるいはその組合せのデータ操作をどのような手順で実行するかが決定され，最終的な問合せ実行プランが生成される.

一般に，各代数演算操作を実行するための手順は一通りとは限らず，またその適用可能性や処理時間は物理的なデータ構成に依存することが多い.

問合せ実行プランの生成においては，一つの問合せに対して複数のプランが考え得ることに注意する必要がある．上で示したフェーズ１で変換後の演算子列は，入力として与えられるデータベースが同じときには同じ結果を返すという意味では等価である．しかし，各プランの処理時間や必要なメモリ領域の大きさは一般に大きく異なる．上の三つの演算子列においては，3 番目が一般には最も実行効率がよいものとなる.

このように，一つの問合せに対して複数のプランが存在し，かつそれらの実行コストが異なるため，問合せ処理においてはできるだけ実行コストの小さい実行プランを選択することが求められる．このことを**問合せ最適化** (query optimization) という.

(1) 規則を利用した問合せ最適化 (Rule-based Query Optimization)

問合せ最適化は，(1) 選択（特に１つだけの表についての選択）をできるだけ早い段階で適用（AND で結ばれた条件を持つ選択を，可能なら AND の前と後の条件で分けた複数の選択に分割する），(2) 射影による不要な属性の削除をできるだけ早い段階で実行，(3) 直積の後に選択を行うときは，可能な限り結合にまとめる，(4) 連続した選択および射影を，単一の選択，単一の射影，または単一の選択とそれに続く単一の射影にする，という規則で行われる.

上記のような変換を考える上では処理木（processing tree) を用いるのが便利である．処理木の例を図 29 に示す.

(2) コスト見積りによる問合せ最適化 (Cost-based Query Optimization)

コスト見積りによる問合せ最適化では，問合せを実行するコストを，問合せの結果を求めるのに必要な各種の演算の実行時間を見積もってそれらを合計することで求める.

コストの見積りは，データベース中で問合せの対象となるリレーションのタップルの数（濃度）や，選択率（選択演算の前後で，選択演算により絞られるタップルの数の比）などに依存する（ここでは詳細は省略する）.

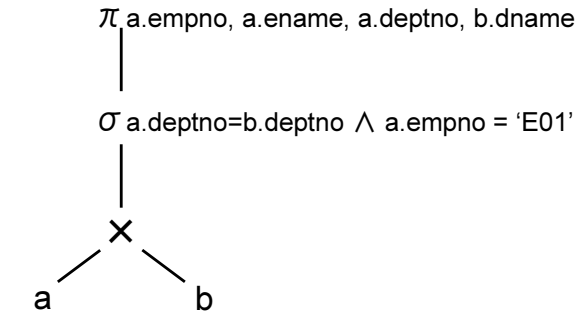
5.9 基本データ操作の実行方法

5.9.1 選択の実行方法

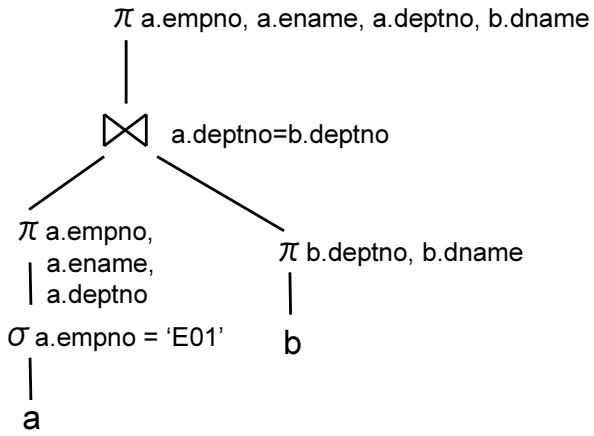
選択の実行方法には次のようなものがある.

(A) 線形探索

表のデータファイルの全タップルを順次読み出し，選択条件を満たすものを抽出する.



(1)のリレーショナル代数式に対応する処理木



(3)のリレーショナル代数式に対応する処理木

図 29: 問合せ最適化の処理木による表現

(B) 索引を用いた探索
表のデータファイルがハッシュファイル，索引付きファイル，B^+ 木などの索引をもつファイル編成で構成されており，選択条件で索引が利用できる場合には，索引を用いて選択条件を満たすタプルを探索する．

5.9.2 結合の実行方法

結合の実行方法には次のようなものがある．以下では，２つの表 R と S の $R \bowtie_{R.X=S.Y} S$ という結合演算を考える．また，それぞれの表のタップルを，$R_1, R_2, ..., R_m$ と $S_1, S_2, ..., S_n$ で表す．

(1) 入れ子ループ結合 (nested loop join)
入れ子ループ結合は結合演算を行うための最も基本的な手順である．次の擬似コードで表現される．

```
for i := 1 to m do
  for j := 1 to n do
    if Ri[X] = Sj[Y] then R1[X] と Sj[Y] を結合したタプルを出力
```

(2) 索引を用いた結合
次の擬似コードで表現される．

```
for i := 1 to m do
  S.Y に関する索引により Ri[X] = Sj[Y] を満たす S のタプル Sj を探索
  Ri[X] と Sj[Y] を結合したタプルを出力
```

(3) マージ結合 (merge join)
R と S がそれぞれ X と Y でソートされている場合にはマージ結合を適用できる．ソートが昇順であるときのマージ結合の手順は次のようになる．

```
i := 1
j := 1
while true do
  while Ri[X] > Sj[Y] do
    j := j+1
    if j>n then 終了
  while Ri[X] < Rj[Y] do
    i := i+1
    if i>m then 終了
  if Ri[X] = Sj[Y] then
    k := j
    repeat
      Ri[X] と Sj[Y] を結合したタプルを出力
      j := j+1
    until j>n または Ri[X] ≠ Sj[Y]
    i := i+1
    if i>m then 終了
    if Ri[X]=Sk[Y] then j := k
    else if j>n then 終了
```

(4) ハッシュ結合 (hash join)
ハッシュ結合は，一方のファイル中のタプルをハッシュ長に展開することにより結合条件を満たすタプルを効率良く求めるものである．最も基本的なハッシュ結合の手順を以下に示す．なお，ハッシュ関数を hash とし，ハッシュ表は B(1), ..., B(K) からなるものとする．

```
for i:=1 to n do
  タプル Ri を B(hash(Ri[X])) に格納
for j:=1 to m do
  foreach タプル Ri in B(hash(Sj[Y]))
    if Ri[X]=Sj[Y] do
      Ri[X] と Sj[Y] を結合したタプルを出力
```

6　リレーショナルデータベースの実装

6.1　データベース管理システムの構成

データベース管理システムは次の構成要素を基にして実装される．

(1) データベースファイル
(2) 共有メモリ
(3) データベースプロセス

6.1.1　データベースファイル

データベースファイルとは，データベースを管理するのに必要なファイル群を指す．多くのデータベース管理システムは次のファイルを持っている．

- データファイル
- ログファイル
- 制御ファイル

データファイルは，表の内容を保持しているファイルである．実運用されているデータベースシステムのデータ量は増加の一途をたどっており，データ量が何テラバイトにも及ぶデータベースも珍しくなくなっている．このため，1 個のデータベースの内容を複数のデータファイルに分けて管理する機能が必要となる．データファイル中のデータには，表のデータ以外に 3.3 節で述べたように，インデックスのデータも含まれる．

ログファイルは，主にデータベースの変更履歴を格納するファイルである．ログファイルは単に変更履歴の格納だけでなく，障害が発生してしまった場合の復旧に必要となる．障害でデータが失われてしまったとしても，ログファイルにある履歴を基に処理を再実行することでデータを復旧させることが可能となる．データベースの障害からの復旧操作に重要なファイルである．

制御ファイルには，データベースを構成するファイルの名前や格納されている場所，および各ファイルの現在の状態など，データベースに関する管理情報が格納されている。データベース管理システムは，データベース起動時にこの制御ファイルを読み込むことで，データファイルやログファイルの場所や状態を把握する．

6.1.2　共有メモリ

データベース管理システムが，データの検索や変更を行うたびに上記で説明したデータベースファイルにアクセスすると，ディスクへのアクセスが頻繁に発生し，データベース管理システムの性能低下を招く恐れがある．共有メモリはディスクの読み書き操作のキャッシュとして使われる．

共有メモリには，主に次のようなものがある．

- データベースバッファキャッシュ
- ログバッファ

データベースバッファキャッシュは，データベースの表操作で使われる．データベースへの問合せ時は，このキャッシュに対象の表データがあるかどうかを調べ，あればそれを検索し，なければデータファイルから表データを読み込んでキャッシュに格納した後，問合せ操作を実行する．

ログバッファは，ログファイルの読み書きで使われるメモリ領域である．データベースの変更履歴などログファイルに書き込むデータは，いったんログバッファにためておき，必要なタイミングでまとめてログファイルに書き込むことでディスクへのアクセス回数を軽減している．

6.1.3　データベースプロセス

データベース管理システムは，データベース管理のために種々の操作を実行する必要があり，全部を一つのプロセスで実行すると，そのプロセスの負荷が多大となる．このため，データベースの操作の種類ごとに複数のプロセスに役割を分担することで処理の効率化を図っている．

データベースプロセスには次のようなものがある．

- サーバプロセス
- データベースライタープロセス
- ログライタープロセス

サーバプロセスは，データベースに対する問合せや更新の操作を担当する．一つのサーバプロセスが全部の操作を行うのではなく，複数のプロセスで処理を分担するのが通常である．操作の処理では，まずデータベースバッファキャッシュにアクセスし，キャッシュになければデータファイルから読み出す．

データベースライタープロセスは，データファイルに書き込まれたデータベースの更新データをデータファイルに書き込む処理を担当する．

ログライタープロセスは，ログバッファに書き込まれたデータベースの変更履歴等の情報をログファイルに書き込む処理を担当する．

6.2 トランザクション処理

データベース内のデータに対する一連の「論理的」操作群のことをトランザクションと呼ぶ．ただし，このトランザクションという概念は障害に対する復旧操作とも関係しており，どの操作からどの操作までが 1 個のトランザクションかは，データベース管理システムが自動的に知ることはできない．このため,「ここからここまでが 1 個のトランザクションである」ということは，ユーザが明示的に指定する必要がある．

トランザクションが満たすべき性質としては次のものがある．以下のそれぞれの頭文字を取って ACID と呼ばれる．

- 原子性（atomicity, 不可分性とも呼ばれる）
- 整合性（consistency）
- 独立性（isolation）
- 永続性（durability）

原子性は，オペレーティングシステムにおける臨界領域と類似の概念であり，トランザクションに含まれる個々の操作は不可分であり，各操作がすべて実行されるか，あるいは全く実行されないことを保証する性質をいう．

整合性は，7 章で述べる一貫性を満たしている状態を指す．一貫性と呼ぶときは，データベースが常に 7 章で述べる条件を満たしていることを指すのに対して，整合性と呼ぶときにはある時点（例えば，トランザクションの開始時と終了時）にのみ満たされている（逆に言えば，トランザクションの処理中は必ずしも満たされなくてもよい）性質を指す．

独立性は，トランザクション中に行われる操作の過程が，そのトランザクションの外の操作から隠蔽されることを指す．トランザクションの外部からは，トランザクションの開始時または終了時のデータベースの内容しか参照できない（つまり，トランザクション実行中の途中のデータベースの内容は参照できない）ことを保証する．

永続性は，トランザクション操作の完了通知をユーザが受けた時点で，その操作は永続的となり，結果が失われないことを指す．前述のログファイルの機能により，トランザクションが完了できない場合はトランザクションの開始時の内容，完了されたときには終了時の内容となり，障害等によりトランザクション実行過程の途中の状態のまま保持されることがないことを保証する．

7 リレーショナルデータベースの一貫性

7.1 一貫性と一貫性制約

一貫性 (integrity, 保全性とも呼ばれる)

- データベースの内容がモデリングの対象となった実世界の事物およびそれらの間の関連を正しく表現していること．

一貫性制約 (integrity constraint, 保全性制約，意味制約とも呼ばれる)

- データベースの一貫性を保証するための条件．
- ドメイン制約，キー制約，外部キー制約，関数従属性，多値従属性などがある．
- 一貫性制約はリレーションスキーマにおいて成り立つ条件である．つまり，そのリレーションスキーマを持つリレーションのすべてのインスタンスにおいてその性質が成り立っていることを表す．
- 一貫性制約に反するようなリレーションの更新（タップルの挿入，削除，修正）は許されない．

7.2 一貫性制約の種類

7.2.1 ドメイン制約 (domain constraint)

- リレーションにおける各属性のドメインに対する制約 (例 最低賃金，年齢制限，最大労働時間)．

[SQL での記述]

```
create domain ドメイン名 as データ型
```

```
  check 表名. 列名 between 下限 and 上限
create table 表名
  (列名 ドメイン名, ...)
```

7.2.2 キー制約 (key constraint)

- 主キーに対する制約：空値でなく，かつ候補キーの条件を満たす.
- 主キー以外の候補キーに対する制約：空値か，または空値でないときは候補キーの条件を満たす.

［参考］K がリレーションスキーマ $\mathbf{R}$ の候補キーであるための条件（3.6 節参照）
(1) R をリレーションスキーマ $\mathbf{R}$ の任意のインスタンスとすると $(\forall t, t' \in R)(t[K] = t'[K] \Rightarrow t = t')$. が成り立つ.
(2) K から属性を 1 個でも欠落させると (1) が成立しない.

```
[SQL での記述]
create table 表名
(列名　ドメイン名 not null, ...) ←空値をとらない
create table 表名
(列名　ドメイン名 unique, ...) ←この属性が同じ値をとるタップルは 1 つしかない
create table 表名
(列名　ドメイン名 primary key, ...) ←この属性が主キーである
```

7.2.3 外部キー (foreign key)

外部キーとは，あるリレーションの主キーを参照できる属性をいう.
リレーションスキーマ $\mathbf{R}$, $\mathbf{S}$ の任意の時刻 τ でのインスタンスを $R\tau$, $S\tau$ とすると，$\mathbf{R}$ のある属性 H について以下の外部キー制約が成り立つとき，H は $\mathbf{S}$ の外部キーとなっている.

外部キー制約 (foreign key constraint)

(1) $R\tau$ の任意のタップル t において，$t[H]$ は空値か，そうでなければ
(2) $t[H] = s[K]$ となる $S\tau$ のタップル s が存在する (但し K は $\mathbf{S}$ の主キー)

(注意)$\mathbf{R}$ と $\mathbf{S}$ は同じリレーションスキーマであっても良い (例社員表における上司)

```
[SQL での記述]
create table 表 1
  (列 1 ドメイン名, ...
    foreign key(列 1) references 表 2) ←列 1 は表 2 の外部キー
```

7.3 関数従属性 (functional dependency)

リレーションスキーマ $\mathbf{R}$ の属性集合 X, Y (X と Y は互いに素である必要はない) について次の条件を満たすとき,「X から Y への関数従属性が存在する」といい，$X \to Y$ と書く.

$$(\forall t, t' \in R)(t[X] = t'[X] \Rightarrow t[Y] = t'[Y]).$$

ここで R はリレーションスキーマ $\mathbf{R}$ の任意のインスタンスである.
[参考] $X \to Y$ を,「X は Y を関数的に決定する」または「Y は X に関数従属している」ともいう.

関数従属性の意義

関数従属性は，リレーションスキーマにおいて成り立つ（任意のインスタンスとしてのリレーションで条件が満たされないといけない）.
また，定義されている関数従属性に反するようなリレーションの更新（タップルの挿入、削除、修正）は許されないことから，一貫性制約の一つと考えることができる.
なお，関数従属性は，後述するようにリレーションの更新時の異状を抑制するための正規形を規定する基礎となっている.

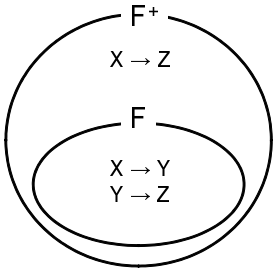


図 30: 関数従属性の閉包の例

完全関数従属性 (full functional dependency)

リレーションスキーマ $\mathbf{R}$ の属性集合 X, Y (X と Y は互いに素である必要はない) について次の 2 つの条件がともに成立するとき,「X から Y への完全関数従属性が存在する」(または「Y は X に完全関数従属している」) という.

(1) $X \to Y$
(2) X のいかなる真部分集合 X' に対しても $X' \to Y$ が成立しない.

自明な関数従属性 (trivial functional dependency)

次の関数従属性は常に成立するため自明な関数従属性と呼ばれる.

(1) $Y \subseteq X$ のとき, $X \to Y$,
(2) $X \to \phi$,
(3) $X \to X$.

関数従属性と超キー (スーパキー)

H がリレーションスキーマ $\mathbf{R}$ の超キーであることは, 関数従属性を使って $H \to \overline{H}$ ($\overline{H} = \Omega_R - H$) と表すことができる.

関数従属性の閉包

リレーションスキーマ $\mathbf{R}$ の関数従属性の集合 F が与えられたとき, F から導出されるすべての関数従属性の集合を F の閉包 (closure) と呼び, F^+ で表す (図 30 参照).

関数従属性の公理系

関数従属性の公理系 (アームストロングの公理系とも呼ばれる) は, リレーションスキーマ $\mathbf{R}$ の属性集合 $X, Y, \quad Z$ に対して次のように表される.

反射律: もし $Y \subseteq X$ ならば, $X \to Y$ (自明な関数従属性の一つ),
添加律: もし $X \to Y$ で, $Z \subseteq \Omega_R$ ならば, $X \cup Z \to Y \cup Z$ ($\cup$ は和集合演算),
推移律: もし $X \to Y$ かつ $Y \to Z$ ならば, $X \to Z$.

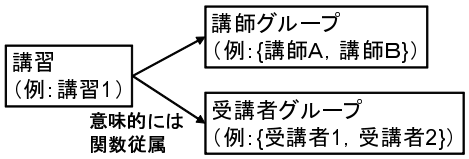
関数従属性の公理系は, 健全 (sound) でかつ完全 (complete) であることが知られている. すなわち, あるリレーションスキーマ $\mathbf{R}$ とそこで初期条件として成り立つ関数従属性の集合 F が与えられたとき, $\mathbf{R}$ で成り立つ任意の関数従属性は F に対して上記の公理系の導出規則を有限回適用することにより得ることができる.

関数従属性の導出の例

与えられた関数従属性に対して, 関数従属性の公理系を適用することで, 別の関数従属性を導出することができる (図 31 参照).

（与えられている関数従属性）
｛学番，科目｝→得点、　｛科目，得点｝→判定
（導出したい関数従属性）｛学番，科目｝→判定
（導出過程）
｛学番，科目｝→得点　　①‥所与
｛学番，科目｝→｛得点，科目｝　②‥①に添加律を適用
｛科目，得点｝→判定　　③‥所与
｛学番，科目｝→判定　　④‥②と③に推移律を適用

図 31: 関数従属性の導出の例



講習	講師グループ	受講者グループ
講習1	講師A	受講者1
講習1	講師A	受講者2
講習1	講師B	受講者1
講習1	講師B	受講者2

図 32: 多値従属性の直感的な説明

7.4 多値従属性 (multivalued dependency)

リレーションスキーマ $\mathbf{R}$ の属性集合 X, Y (X と Y は互いに素である必要はない) について次の条件を満たすとき,「X から Y への多値従属性が存在する」といい，$X \twoheadrightarrow Y$ と書く.

$$(\forall t, t' \in R)(t[X] = t'[X] \Rightarrow (t[X \cup Y], t'[Z]) \in R \wedge (t'[X \cup Y], t[Z]) \in R).$$

ここで R はリレーションスキーマ $\mathbf{R}$ の任意のインスタンスであり，$Z = \Omega_R - (X \cup Y)$ である.

[参考] $X \twoheadrightarrow Y$ を,「X は Y を多値に決定する」または「Y は X に多値に従属している」ともいう.

リレーショナルデータモデルでは，単純でないドメインを持つ属性は，正規化により分解する必要があった．多値従属性は，分解前の属性に対する関連（意味的には関数従属性）と考えることができる（図 32 参照）.

多値従属性の性質

(1) リレーションスキーマ $\mathbf{R}$ の属性集合 X, Y の間に多値従属性 $X \twoheadrightarrow Y$ が成り立つとき，$\mathbf{R}$ の任意のインスタンス R における射影 $\pi_X(R)$ と $\pi_Y(R)$ の間の関係と，射影 $\pi_X(R)$ と $\pi_Z(R)$ の間の関係は直交 (orthogonal) している，つまり，任意のタップル $t, t'(\in R)$ において，$t[X] = t'[X]$ であれば，$t[Y]$ と $t'[Z]$ のすべての組合せが R 中に出現していないといけない（図 33 を参照）.

(2) リレーションスキーマ $\mathbf{R}$ の属性集合 X, Y において，$X \twoheadrightarrow Y$ が成り立つとき，$Z = \Omega_R - (X \cup Y)$ である Z についても $X \twoheadrightarrow Z$ が成り立つ（このことを強調するため，$X \twoheadrightarrow Y|Z$ と書くことがある）.

自明な多値従属性 (trivial multivalued dependency)

次の多値従属性は常に成立するため自明な多値従属性と呼ばれる.

(1) $X \cup Y = \Omega_R$ のとき，$X \twoheadrightarrow Y|\phi$,

(2) $Y \subseteq X$ のとき，$X \twoheadrightarrow Y$.

関数従属性と多値従属性

すべての関数従属性は多値従属性である．つまり，$X, Y \subseteq \Omega_R$ である任意の属性集合 X と Y について，$X \to Y$ であれば $X \twoheadrightarrow Y$ が成り立つ．この意味で，多値従属性は関数従属性の一般化と見なすことができる.

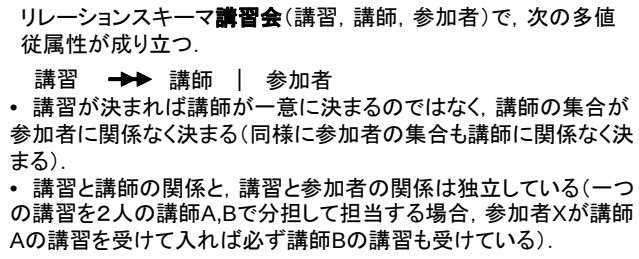


図 33: 多値従属性の例

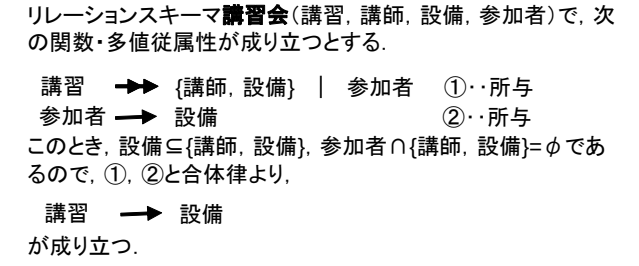


図 34: 多値従属性の導出の例

多値従属性の公理系

リレーションスキーマ $\mathbf{R}$ 上の関数従属性の集合を F, 多値従属性の集合を M とすると，$F \cup M$ から導出されるすべての関数従属性または多値従属性の集合 ($F \cup M$ の閉包) を得るための，完全でかつ健全な次のような公理系が知られている．

以下では X, Y, Z をリレーションスキーマ $\mathbf{R}$ の属性集合とする．

関数従属性の反射律: もし $Y \subseteq X$ ならば，$X \to Y$,
関数従属性の添加律: もし $X \to Y$ で $Z \subseteq \Omega_R$ ならば，$X \cup Z \to Y \cup Z$,
関数従属性の推移律: もし $X \to Y$ かつ $Y \to Z$ ならば，$X \to Z$,
多値従属性の相補律: もし $X \twoheadrightarrow Y$ で $Z = \Omega_R - (X \cup Y)$ ならば，$X \twoheadrightarrow Z$,
多値従属性の添加律: もし $X \twoheadrightarrow Y$ で $Z \subseteq W \subseteq \Omega_R$ ならば，$X \cup W \twoheadrightarrow Y \cup Z$,
多値従属性の推移律: もし $X \twoheadrightarrow Y$ かつ $Y \twoheadrightarrow Z$ ならば，$X \twoheadrightarrow (Z - Y)$,
模写律: もし $X \to Y$ ならば，$X \twoheadrightarrow Y$,
合体律: もし $X \twoheadrightarrow Y$ で $Z \subseteq Y$ であるとき，$W \cap Y = \phi$ である W について $W \to Z$ であれば，$X \to Z$.

多値従属性の導出の例

与えられた関数従属性や多値従属性に対して，多値従属性の公理系を適用することで，別の関数従属性や多値従属性を導出することができる（図 34 参照）.

8 リレーションスキーマの設計

リレーションスキーマを設計するときの基本的な考え方として，多数の属性を持つ大きなリレーションスキーマを作り，検索時には射影演算により必要な属性のみを取り出す方法と，少ない属性を持つリレーションスキーマを多数作り，検索時には必要に応じて結合演算によりリレーションを結合しながら，必要な属性を組み合わせて行く方法がある．

データベースの検索の点からは，一般的に前者の方が，多数のリレーションを結合する手間がないため効率が良いが，データベースに対する更新を考えると，前者の方法は必ずしも良いとは限らない．その理由として，前者の方法では，以下で述べる更新時異状が発生しやすいことがあげられる．

8.1 更新時異状 (update anomalies)

更新時異状とは，必要とされるデータベースの更新が，一貫性制約に反するために行えない状況をいう．更新時異状には次の 3 種類がある．

(1) タップル挿入時異状 (insertion anomaly): 必要なタップルの挿入が，一貫性制約に反するためできない.
(2) タップル削除時異状 (deletion anomaly): 必要なタップルの削除が，一貫性制約に反するためできない.
(3) タップル修正時異状 (modification anomaly): 必要なタップルの修正が，一貫性制約に反するためできない.

更新時異状の例 1

以下，リレーションスキーマ $\mathbf{R}(X,Y,Z,V)$（主キーは $\{X,Y\}$ で，関数従属性 $X \to V$ が成り立っているとする）のインスタンスであるリレーション R を例に，上の 3 種類の更新時異状の例をあげる（例えば，R =「卸売」，X =「商品名」，Y =「卸先名」，Z =「卸値」，V = 「仕入値」）.

- タップル挿入時異状:
  あるタップル $t(\in R)$ において，$t[X]$ に対する $t[V]$ の値だけを R に格納するため，$t[Y]$ と $t[Z]$ は空値にしたタップルを R に挿入しようとしても，$t[Y]$ が空値ではキー制約に反するためできない.
- タップル削除時異状:
  あるタップル $t(\in R)$ において，$t[Y]$ と $t[Z]$ の値がなくなったため，このタップル t を R から削除すると，$t[X]$ の値により決定されていた $t[V]$ の値まで R から消えてしまう．かといって，$t[Y], t[Z]$ の成分だけ空値にしようとしても，キー制約に反するためできない.
- タップル修正時異状:
  あるタップル $t(\in R)$ において，$t[Y]$ と $t[X]$ との関連が変更されたため $t[X]$ の値を修正すると，もとの $t[X]$ の値により決定されていた $t[V]$ の値まで R から消えてしまう．かといって，もとの $t[X]$ と $t[V]$ の成分だけを持つタップルを R に挿入しようとしても，キー制約に反するためできない.

更新時異状の例 2

リレーションスキーマ**授業** (科目，担当，年度，教室) で，関数従属性 { 科目，年度 } → 担当 および 科目 → 教室 が成り立っているものとする．また，リレーションスキーマ**授業**の主キーは { 科目，年度 } であるとする.

このとき，リレーションスキーマ**授業**のインスタンスの更新で，発生可能な更新時異状の例としては次のものがある.

- タップル挿入時異状:
  ある科目について，年度はまだ決まっていないが，科目の性質上，教室が一意に定まるので，科目と教室の成分だけを含むタップルを挿入したいが，この挿入は年度が空値なので，主キー制約に反するためできない.
- タップル削除時異状:
  ある科目について，一時的に開講されなくなったので，年度の情報を消そうとしてそのタップルを削除すると，その科目が行われていた教室の情報まで消える．かといって，年度の成分だけ空値にしようとしても，主キー制約に反するためできない.
- タップル修正時異状:
  ある科目について，科目と教室との対応関係が変わったとき，年度ごとにある複数のタップルの中の教室の成分を一度に修正しなければいけない.

情報無損失分解 (information lossless decomposition)

上の例 1 であげた更新時異状は，リレーション R を，$R_1 = \pi_{XYZ}(R)$ と $R_2 = \pi_{XV}(R)$ に分解した後，更新することにすれば発生しない.

さらに，この分解をしても，R の属性 X,Y,Z のデータは R_1 から検索できるし，R_1 にない属性 V のデータについても，元の R において関数従属性 $X \to V$ が成り立っていることから，$R_1 \bowtie R_2$（R_1 と R_2 の自然結合）により，R_1 の X と R_2 の X とを照合させることで検索することができる．つまり，元のリレーションの情報を失わない.

また，例 2 では，リレーション授業を，2 つのリレーション $\pi_{\text{科目, 担当}}$(授業), $\pi_{\text{科目, 教室}}$(授業) に分解する.

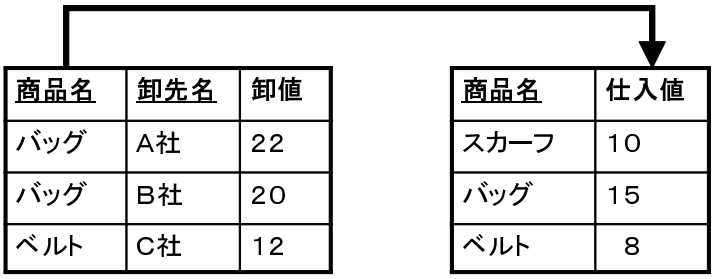
この分解により，科目と教室の成分のみを持つタップルの挿入が可能となり，科目と担当を含むタップルを削除しても科目と教室の情報は消えず，科目と担当との対応関係の修正を科目と教室との対応関係とは独立に修正できるので，更新時異状は生じなくなる.

このように分解した結果できた複数のリレーションを自然結合することにより元のリレーションを復元できるような分解を，**情報無損失分解**と呼ぶ.

情報無損失分解の判定条件

リレーションのある分解が情報無損失分解であることを示すために，次の二つの定理が知られている.

[**定理 1：多値従属性に関する情報無損失分解の条件**]



商品名	卸先名	卸値
バッグ	A社	22
バッグ	B社	20
ベルト	C社	12

商品名	仕入値
スカーフ	10
バッグ	15
ベルト	8

図 35: 情報無損失分解の例

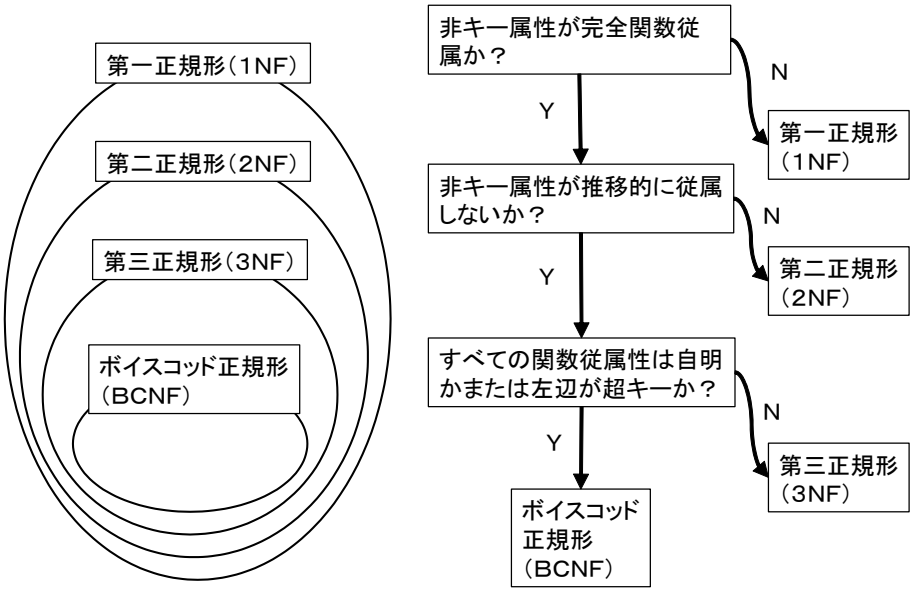


図 36: 関数従属性に関する正規形

R をリレーション，X と Y を $X \cup Y = \Omega_R$ かつ $X \cap Y \neq \phi$ であるような属性集合とするとき，R が射影 $\pi_X(R)$ と $\pi_Y(R)$ に情報無損失分解されるための必要十分条件は，R で多値従属性 $X \cap Y \twoheadrightarrow X|Y$ が成り立つことである．

[定理 2：関数従属性に関する情報無損失分解の条件]

R をリレーション，X と Y を $X \cup Y = \Omega_R$ かつ $X \cap Y \neq \phi$ であるような属性集合とするとき，R が射影 $\pi_X(R)$ と $\pi_Y(R)$ に情報無損失分解されるための十分条件は，R で関数従属性 $X \cap Y \to X - Y$ または $X \cap Y \to Y - X$ が成り立つことである．

（注意）上の二つの定理において，多値従属性は関数従属性の一般的な場合なので，リレーション R に関数従属性のみが成り立つときは，後者の定理は必要十分条件であるが，多値従属性も含めて成り立つときは，後者の定理は十分条件ではあるが，必要条件とはならない．

例 1 で，リレーション卸売を，2 つのリレーション $\pi_{\text{商品名, 卸先名, 卸値}}$(卸売), $\pi_{\text{商品名, 仕入値}}$(卸売) に分解したものが，情報無損失分解になっていることは，次でわかる．

[情報無損失分解であることの証明]

$X =$ { 商品名，卸先名，卸値 }, $Y =$ { 商品名，仕入値 } と置き換えると，明らかに $X \cup Y = \Omega_{\text{卸売}}$ かつ $X \cap Y \neq \phi$．また，商品名 $\to$ 仕入値 が成り立っていることから $X \cap Y \to Y - X$．すなわち，この分解は関数従属性に関する情報無損失分解の条件を満たしている．したがって，この分解は情報無損失分解である．

9　リレーショナルデータベースの正規化理論

以前にリレーショナル代数によるデータベース操作を行うための条件として非正規形を第一正規形に変換する正規化について説明したが，問合せ操作だけでなく更新操作までを含めると第一正規形というだけでは不十分で，さらに条件の付いた高次の正規形が必要になる．

高次の正規形のうち，従属性として関数従属性のみが成り立っているリレーションでは，関数従属性に関する高次正規形のどれに相当するかを調べる必要がある（図 36 参照）．関数従属性のみが成り立っているリレーションでは，ボイスコッド正規形になっているか，あるいは情報無損失分解によりボイスコッド正規形にまで分解できれば更新時異状が発生しないことが知られている．しかし，ボイスコッド正規形への分解では，関数従属性が保存されないような分解が生じることも知られているため，元のリレーションに存在する関数従属性を分解後も保持しておきたければ，ボイスコッド正規形より低次の正規形で分解を止める必要がある．

関数従属性の他に多値従属性・結合従属性が成り立っているリレーションでは，ボイスコッド正規形でも更新時異状が発生する可能性があり，さらに高次の正規形に情報無損失分解する必要がある（図 37 参照）．

9.1 第二正規形

リレーションスキーマ $\mathbf{R}$ が第二正規形 (second normal form, 2NF) であるとは，次の二つの条件を満たすときをいう．

(1) $\mathbf{R}$ は第一正規形である．
(2) $\mathbf{R}$ のすべての非キー属性は $\mathbf{R}$ の各候補キーに完全関数従属している．

[参考] 完全関数従属性

リレーションスキーマ $\mathbf{R}$ の属性集合 X, Y (X と Y は互いに素である必要はない) について次の 2 つの条件がともに成立するとき,「X から Y への完全関数従属性が存在する」（または「Y は X に完全関数従属している」）という．

(1) $X \to Y$
(2) X のいかなる真部分集合 X' に対しても $X' \to Y$ が成立しない．

[第一正規形であって第二正規形でない例]
リレーションスキーマ**卸売** (商品名，卸先名，卸値，仕入値)
主キー: { 商品名，卸先名 }
関数従属性: { 商品名，卸先名 } → 卸値, 商品名 → 仕入値．（非キー属性 仕入値 は候補キーに完全関数従属していない）．

[情報無損失分解の例]
$\pi_{\text{商品名，卸先名，卸値}}$(卸売) と $\pi_{\text{商品名，仕入値}}$(卸売) とに分解．

9.2 第三正規形

リレーションスキーマ $\mathbf{R}$ が第三正規形 (third normal form, 3NF) であるとは，次の二つの条件を満たすときをいう．

(1) $\mathbf{R}$ は第二正規形である．
(2) $\mathbf{R}$ のすべての非キー属性は $\mathbf{R}$ のいかなる候補キーにも推移的に関数従属しない．

推移的関数従属性 (transitive functional dependency)

リレーションスキーマ $\mathbf{R}$ の属性集合 X, Y について次の 3 つの条件が成立するとする．

- $X \to Y$
- $Y \not\to X$ ($Y \to X$ は存在しない)
- $Y \to Z$ （自明ではないとする）

すると，次が成立する．

- $X \to Z$
- $Z \not\to X$ ($Z \to X$ は存在しない)

このとき，新しく得られた $X \to Z$ という関数従属性を**推移的関数従属性**という．

[第二正規形であって第三正規形でない例]
リレーションスキーマ**配属** (社員名，プロジェクト名，契約タイプ)
主キー: 社員名
関数従属性: 社員名 → プロジェクト名, プロジェクト名 → 契約タイプ．（非キー属性 契約タイプ は候補キー 社員名 に推移的に従属している）．

[情報無損失分解の例]
$\pi_{\text{社員名，プロジェクト名}}$(配属) と $\pi_{\text{プロジェクト名，契約タイプ}}$(配属) とに分解．

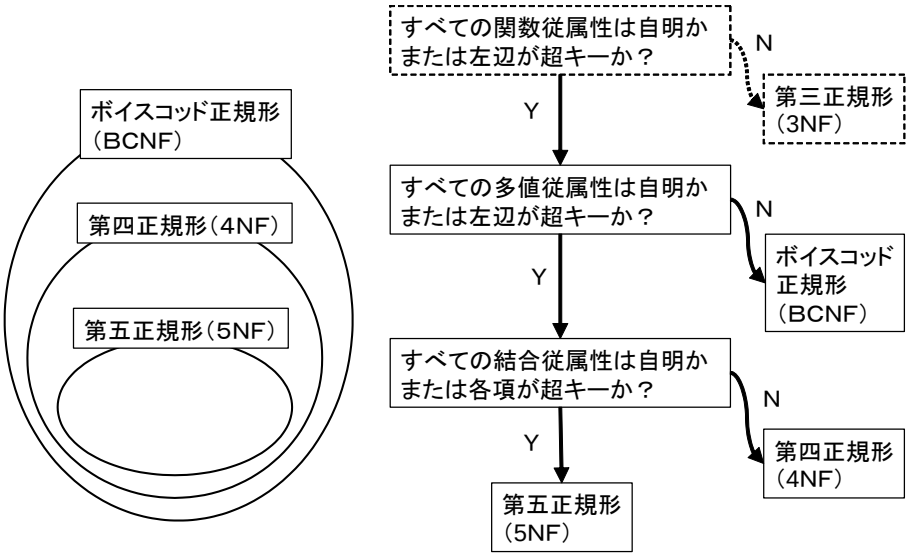


図 37: 多値従属性・結合従属性に関する正規形

9.3 ボイス-コッド正規形

リレーションスキーマ $\mathbf{R}$ がボイス-コッド正規形 (Boyce-Codd normal form, BCNF) であるとは，次の条件が成立するときをいう．

$X \to Y$ を $\mathbf{R}$ の任意の関数従属性とするとき，

(1) $X \to Y$ は自明な関数従属性であるか，または

(2) X は $\mathbf{R}$ の超キーである．

[第三正規形であってボイスコッド正規形でない例]
リレーションスキーマ**受講** (学生名，科目名，教員名)
主キー: { 学生名，科目名 }
関数従属性: { 学生名，科目名 } → 教員名, 教員名 → 科目名．（{ 学生名，科目名 } だけでなく，{ 学生名，教員名 } も候補キーとなる．したがって，非キー属性がないので第三正規形．関数従属性 教員名 → 科目名 は自明な関数従属性ではなく，かつ属性 教員名 は**受講**の超キーではないので，ボイスコッド正規形ではない）．

[情報無損失分解の例]
$\pi_{\text{教員名, 科目名}}$(受講) と $\pi_{\text{教員名, 学生名}}$(受講) とに分解．
（注意）この分解では，関数従属性 { 学生名，科目名 } → 教員名 が保存されない．

9.4 第四正規形

リレーションスキーマ $\mathbf{R}$ が第四正規形 (fourth normal form, 4NF) であるとは，次の条件が成立するときをいう．

$X \twoheadrightarrow Y$ を $\mathbf{R}$ の多値従属性とするとき，

(1) $X \twoheadrightarrow Y$ は自明な多値従属性であるか，または

(2) X は $\mathbf{R}$ の超キーである．

[参考 1] 自明な多値従属性には以下の 2 つがある．

(1) $X \cup Y = \Omega_R$ なら $X \twoheadrightarrow Y|\phi$,

(2) $Y \subseteq X$ なら $X \twoheadrightarrow Y$.

[参考 2] 多値従属性の公理系のうちの模写律にあるように，$X \to Y$ なら $X \twoheadrightarrow Y$ である．第四正規形では，$\mathbf{R}$ から導出されるすべての多値従属性について上の条件が成立するので，第四正規形はボイス-コッド正規形に含まれている，つまり，ボイス-コッド正規形より高次の正規形である．

[ボイス-コッド正規形であって第四正規形でない例]
リレーションスキーマ: **講習会** (講習名，指導員名，参加者名)
主キー: { 講習名，指導員名，参加者名 }
多値従属性: 講習名 $\twoheadrightarrow$ 指導員名 | 参加者名

（自明な関数従属性以外の関数従属性は成り立っていないので，**講習会**はボイス-コッド正規形である．多値従属性 講習名 $\twoheadrightarrow$ 指導員名 | 参加者名 は自明な多値従属性ではなく，かつ属性 講習名 は**講習会**の超キーではない．したがって，**講習会**は第四正規形ではない）．

[情報無損失分解の例]

$\pi_{\text{講習名，指導員名}}$(講習会) と $\pi_{\text{講習名，参加者名}}$(講習会) とに分解．

($X =$ { 講習名，指導員名 }, $Y =$ { 講習名，参加者名 } とすると，明らかに $X \cup Y = \Omega_R$, $X \cap Y \neq \phi$ であり，さらに，$X \cap Y \twoheadrightarrow X|Y$ が成り立っているので，多値従属性に関する情報無損失分解の条件より，この分解は情報無損失分解である．

9.5 結合従属性

第四正規形よりさらに高次な正規形を導入するため，リレーションを同時に $n(\geq 2)$ 個の射影に分解することを考える．このような分解が情報無損失分解であることを保証する従属性が，以下にあげる結合従属性である．

結合従属性 (join dependency, JD)

$X_1, X_2, \cdots, X_n$ を，$X_1 \cup X_2 \cup \cdots \cup X_n = \Omega_R$ となるような，リレーションスキーマ $\mathbf{R}$ の任意の属性集合とすると，$\mathbf{R}$ の任意のインスタンス R で，

$$R = \pi_{X_1}(R) \bowtie \pi_{X_2}(R) \bowtie \cdots \bowtie \pi_{X_n}(R)$$

が成り立つとき，$\mathbf{R}$ で結合従属性が成立するといい，$*(X_1, X_2, \cdots, X_n)$ と表す．

自明な結合従属性 (trivial join dependency)

もし，ある i $(1 \leq i \leq n)$ で $X_i = \Omega_R$ であれば，結合従属性 $*(X_1, X_2, \cdots, X_n)$ は常に成り立つ．これを，自明な結合従属性という．

9.6 第五正規形

リレーションスキーマ $\mathbf{R}$ が第五正規形 (fifth normal form, 5NF) であるとは，次の条件が成立するときをいう．

$X_1, X_2, \cdots, X_n$ を $X_1 \cup X_2 \cup \cdots \cup X_n = \Omega_R$ となるような，$\mathbf{R}$ の任意の属性集合とするとき，

(1) $*(X_1, X_2, \cdots, X_n)$ は自明な結合従属性であるか，または

(2) 各 $X_i (1 \leq i \leq n)$ は $\mathbf{R}$ の超キーである．

[第四正規形であって第五正規形でない例]

リレーションスキーマ: **ツアー** (ツアー名，キャリア名，代理店名)

主キー: { ツアー名，キャリア名，代理店名 }

結合従属性: $*$({ ツアー名，キャリア名 }, { キャリア名，代理店名 }, { 代理店名，ツアー名 })

（自明でない関数従属性と多値従属性はないので，明らかに**ツアー**は第四正規形であるが，結合従属性 $*$({ ツアー名，キャリア名 }, { キャリア名，代理店名 }, { 代理店名，ツアー名 }) を構成しているどの属性集合も $\Omega_{ツアー}$ ではないし，**ツアー**の超キーにもなっていない．したがって，**ツアー**は第五正規形でない）．

[情報無損失分解の例]

$\pi_{\text{\{ ツアー名，キャリア名 \}}}$(ツアー)，$\pi_{\text{\{ キャリア名，代理店名 \}}}$(ツアー)，$\pi_{\text{\{ 代理店名，ツアー名 \}}}$(ツアー) の 3 個に分解．

結合従属性 $*$({ ツアー名，キャリア名 }, { キャリア名，代理店名 }, { 代理店名，ツアー名 }) が成り立っているので，結合従属性の定義より，分解したすべてのリレーションを自然結合すると元のリレーションが復元できる．したがって，上の分解は情報無損失分解である．

10 データベースの将来：ビッグデータへの対応

10.1 ビッグデータとは？

ビッグデータとは，典型的なデータベースソフトウェアが管理できる能力を超えたサイズのデータを指すとされている．

ビッグデータの特徴として、以下の 4V があげられている。

http://www.ibmbigdatahub.com/infographic/extracting-business-value-4-vs-big-data

Volume（データ量）（例）2020 年までに生成されるデータ量は 40ZB（ゼタバイト、10^{21} バイト）に達すると言われている．
Velocity（頻度・スピード）　（例）ニューヨーク証券取引所では、1 回の取引で 1TB のデータを取り扱っている．
Variety（多様性）（例）Facebook では毎月 300 億個ものデータを共有している．
Veracity（正確性）（例）ある調査によると調査したデータのうち 27%が不正確だった．

10.2　ビッグデータへの取組み

2012 年 3 月 29 日に米国政府がビッグデータ関連の総額 2 億ドル以上を投じた研究開発イニシアティブの概要を発表した．このイニシアティブでは，米国科学技術政策局（OSTP）・米国国防総省（DoD）・米国国立衛生研究所（NIH）・米国国立科学財団（NSF）・米国エネルギー省（DoE），米国地質調査所（USGS）の 6 つの政府機関が主導して，巨大なデジタルデータの組織化やそこからの知識抽出等を行うための技術やツールの開発を行うとされている．

同イニシアティブでは，(1) 巨大な量のデータの収集，保存，運用，分析，共有に必要な中核技術の進歩，(2) 科学技術分野での発見速度の加速や，国家安全保障の強化，教育・学習の変化への当該技術の活用，(3) ビッグデータ技術の発展・活用に必要な労働人口の拡大を目指すとしている．

10.3　ビッグデータの活用

ビッグデータの活用については，現在，検索，電子商取引，ソーシャルメディア等のウェブサービス分野において多量に生成・収集等されるデータを各種サービスの提供のために活用することを中心に進展してきている．その代表例が，Amazon, Apple，Facebook，Google など米国のネット系プラットフォーム事業者である．各社は，利用者の商品・デジタルコンテンツ等の購買履歴や決済情報，コミュニケーションの発信履歴など膨大なデータを蓄積しており，それらのデータを活用しつつサービス革新等を進めることが，各社の競争力につながっている側面があると考えられる．

10.4　RDBMS の限界

RDBMS（リレーショナルデータベース管理システム）は，高頻度に更新されるのは小規模のデータであり、大規模データについては更新は低頻度であるような，トランザクションに最適化されて設計されているため、ビッグデータに基づく応用事例では性能が劣化してしまう．

また，ビッグデータの格納場所は単一のサーバからネットワークに分散する複数のサーバへと移りつつあるが，RDBMS だとネットワークの通信速度がボトルネックとなる．

10.5　NoSQL

NoSQL とは，RDBMS 以外のデータベース管理システムを指し，リレーショナルデータベースの長い歴史を打破するものとして，広い意味でリレーショナルデータモデル以外に属するデータベースの発展を促進させようとする活動を指す．

NoSQL という言葉から，当初は「SQL は不要」という主張という印象があったが，現在は “Not Only SQL”(SQL だけでない）と解釈されている．

RDBMS の限界を克服するため登場したのが NoSQL データベースである．

NoSQL データベースでは，分散環境で高い処理性能を発揮することを念頭に置いて設計されている．

10.6　NoSQL データベースの分類

以下のようなものがある．

キー・バリュー型　キーに対して一つの値という単純な構造
列指向　キーに対してカラム（列の名前と値の組み合わせ）の集合が対応．ほぼ，リレーショナルデータベースのタプルに相当．
ドキュメント指向　文書データの格納を指向したフォーマット（現状では多くの場合 JSON フォーマット）でデータを記述．

10.7 NoSQL とリレーショナルデータベースの違い

リレーショナルデータベースでは，データベース中のデータが，リレーショナルデータモデルに基づくスキーマに厳密に従うように（つまり，第一正規形であり，かつ，7 章で述べる一貫性を満たすように）しなければならないのに対して，NoSQL，特にドキュメントデータベースではある程度柔軟にデータを表現したり，新たな属性の追加などが許されるという違いがある．

また，6.2 節で述べているトランザクション処理の ACID を厳密には満たさないことで処理効率を上げる機能を持つシステムがある．

RDBMS のトランザクション処理では，ACID を満たすため，複数の更新リクエストを受け取ったときは，他を待たせておいて，更新リクエストを一つずつ処理していく．このため、大量の更新リクエストが発生すると待ち時間が大きくかかってしまう．

NoSQL では，トランザクション処理を完全に行うことをある程度あきらめることで，効率の良い処理を実現している実装がある．

なお，リレーショナルデータベースと異なり，キー・バリュー型や列指向の NoSQL データベースではキーの値でしか検索できない．キー以外で検索するときは転置インデックスを使用する必要がある．

10.8 NoSQL のシステム例

10.8.1 Riak

代表的なキー・バリュー型 NoSQL データベースシステムの一つ（オープンソース版あり）．

データはバケット（SQL の表に対応）ごとに保存される．

キーとして時刻を指定して、センサーからのデータを格納することができる（IoT 向け応用）．

データは自動的に複数のサーバに複製が取られる．特定のサーバに障害が起きても別のサーバで代替できる．データの参照が複数のサーバに分散されるため効率がよいが、更新を反映するのに時間がかかる（結果整合性）．

利用例：BestBuy 社のシステム

10.8.2 HBase

列指向データベースの代表的なシステム例の一つである．

リレーショナルデータベースでは扱えないような大規模データの処理を，分散システムによる並列処理で解決することを目指しており，Facebook や Twitter など大規模なデータを抱える企業で使われている．

キーバリュー型データストアのシステムの一つである Hadoop をベースとして，その上に構築されている．

大規模なデータを大量のマシンで並列に処理するための分散計算技術である MapReduce をサポートしているのが特徴である．

MapReduce という名前は処理を「Map フェーズ」と「Reduce フェーズ」という 2 段階のフェーズに分割することに由来している．Map フェーズでは大量の情報を分解し，必要な情報を抜き出して出力する．Reduce フェーズでは Map フェーズで抽出された情報を集約し，それに対して計算を行い結果を出力する．

Map、Reduce のそれぞれのフェーズでは他のマシンと通信することがないため，数万台のマシンに効率よくタスクを分配することができ，台数に比例する能力でデータを処理することができる．タスクを分配する際には，データを保持しているノードでそのデータを処理する計算を始めることで，余計なファイルの転送を防ぐことができる．またマシンが途中で故障して処理が中断した場合も自動的に他のノードにデータのコピー（レプリカ）を作成し，同じ処理を続けるなど、耐障害性についても配慮されている．

10.8.3 MongoDB

ドキュメントデータベースの代表的なシステムの一つである．

データベースの内容は，表形式ではなく，JSON (JavaScript Object Notation) と呼ばれる，プログラム間のデータ交換でよく使われているデータ形式で表現される（厳密には，JSON を拡張した BSON が使われている）．

また，データベースの構成では，リレーショナルデータモデルでいうリレーションは「コレクション」，タプルは「ドキュメント」，属性（正確には各タプルの成分）は「フィールド」と呼んでいる．

以下に，MongoDB のドキュメントの例をあげる．

```
{ "_id" : ObjectId("XXXXXX"), "name" : "matsuda", "lectures" : [  "operating system",  "database" ] }
```

_id は各ドキュメントに自動的に付加される ID であり，name, lectures はフィールド名である．lectures のフィールドは配列を格納している．MongoDB では，フィールドのデータ型として配列をとることができる．つまり，巾集合を持つような非正規形のデータを取り扱うことができる．

MongoDB のデータベースに対する捜査は，CRUD (Creation, Read, Update, Delete) 操作と呼ばれる．

検索（MongoDB では Read Operation）は，find(検索条件) で実行できる．ほぼ，リレーショナルデータベースの SQL の持つ検索と同等の操作ができる．ただし，MongoDB には結合問合せ (Join) の機能がない点が異なる．

検索時には，B 木を使ったインデックスを作成して利用することができ，複数のフィールドにまたがるインデックスの作成など豊富な機能が提供されている．

更新も，SQL とほぼ同じ操作ができる．大きな違いは 6.2 節で述べたトランザクション処理をはサポートしていないことである．代わりに，findAndModify という検索と更新を一度に行う操作などが提供されている．

NoSQL の多くのシステム例と同様，MapReduce による集約演算をサポートしている（後述）．

10.9 NoSQL データベースの特徴

リレーショナルデータベースとの比較によりまとめる．

	NoSQL	リレーショナル
大量データへの対応方式	スケールアウト	スケールアップ
データアクセス	特定の属性からのアクセスを効率化	どの属性でも均一にアクセス
データベースの整合性	整合性を緩和し効率を重視	効率よりも整合性を重視
スキーマの必要性	スキーマは不要（スキーマレス）	スキーマが必須

(注)
スケールアウト：サーバの数を増やしていくことで処理性能を向上させる．
スケールアップ：サーバ単体の性能を向上させて，ソフトウェアの処理性能を上げる．

10.10 NewSQL

NoSQL データベースはビッグデータへの対応では成功したが、以下の問題点がある．

- トランザクション処理を備えていない、もしくは弱い（ACID 特性の一部が厳密には保証されない）．
- 複数データセットにまたがる問合せ（SQL では結合問合せ）と問合せ最適化を備えていない、もしくは弱い．

NoSQL データベースをベースに、RDBMS が備える上記の機能を実装しようという動きがあり、NewSQL と呼ばれている．

NewSQL の例：Spanner, VoltDB, ScaleDB, NuoDB, Clustrix, FoundationDB, CockroachDB

NewSQL は、NoSQL データベースをベースに上記の機能を実装することで、ビッグデータに対応し RDBMS の機能も併せ持つデータベースの構築を目指している．

参考文献

[1] R. Hull and R. King, Semantic Database Modeling: Survey, Applications, and Research Issues, *ACM Computing Surveys*, vol.19, no.13, pp.201–260, 1987 (邦訳： 田中克己,「意味データベースモデリング：サーベイ，応用，研究課題」，コンピュータサイエンス，bit 7 月号別冊, 1989).

[2] 増永良文, リレーショナルデータベース入門ーデータモデル・SQL・管理システムー，オーム社, 2003.

[3] 北川博之, データベースシステム (情報系教科書シリーズ第 14 巻)，昭晃堂, 1996.

[4] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems (6th Edition), Addison-Wesley, 2010.

[5] S. Tiwari, Professional NoSQL, Wrox, 2011 (邦訳: 長尾高弘 訳, 中村泰久 監修, NoSQL プログラミング実践活用技法, 翔泳社, 2012).