さわって実感

リレーショナルデータベース設計

データベース設計（1）

日本PostgreSQLユーザ会理事長 石井達夫 ISHII Tatsuo
ishii@jp.postgresql.org

魚拓本？

先日，拙著『PostgreSQL完全攻略ガイド（第2版）』（通称『シーラカンス本』）の台湾版が出版され，見本が送られてきました．メーリングリストで台湾でのPostgreSQLユーザの存在は知っていましたが，書籍の需要があるほどだとは思っていなかったので，嬉しい驚きでした．

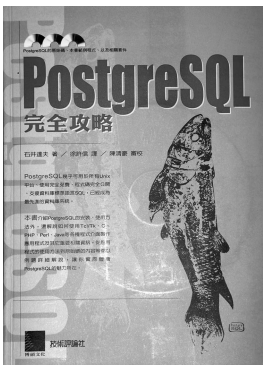
筆者は中国語はまったくわかりませんが，それでもページをめくっていると，なんとなく内容がわかるような気がするから不思議です．たとえば，

- 案装　：インストール
- 系統　：システム
- 資料表　：テーブル
- 交易　：トランザクション

と翻訳されているようです．

ところで，台湾版の表紙ですが，図1のような感じです．誌面ではわかりませんが，しぶめの黄色というか，オレンジ色の上にシーラカンスの魚拓？が乗っているようなデザインで，ちょっとレトロな感じがします．ご丁寧にシーラカンスの右下には“PostgreSQL”というなにやら印鑑らしきものまで押してあるので，ますます魚拓のように見えます．筆者は密かに『魚拓本』と呼ぶことにしました:-)．

図1　魚拓本



リレーショナルデータベース設計の工程

前回はリレーショナルデータベースの基礎を紹介しましたが，今回からいよいよ，本連載の主題であるリレーショナルデータベース設計について解説していきます．

ところで，読者の皆さんはテーブルを作るときにどうしているでしょうか？簡単なものなら，いきなりテーブルを作ることもあるでしょうが，ちょっと複雑になると，頭の中で構想を練ったり，ときには紙の上に図やメモを書いて検討することがあると思います．

これから解説する方法は，まさにそうして無意識のうちに行っている作業を体系化したものです．

2 データベース設計（1）

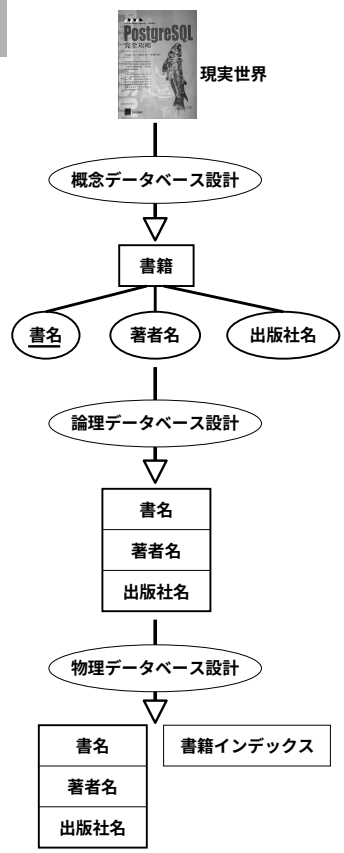
概念データベース設計

この段階では，データベースで管理したい現実世界をできるだけ忠実にモデル化します[注1]．これを概念データベース設計と呼び，概念設計の結果できたドキュメントを概念モデルといいます．

概念モデルを作成するにあたっては，特定のデータモデルを意識しませんし，ましてやコンピュータで処理する際の効率など一切考えません．そういった細かいことや雑事にとらわれずに，データベースの骨格を作るのがこの段階です．

すべてを単純な表で表現するリレーショナルモデルと，複雑な現実の世界の間には，あまりにもギャップがありすぎるため，概念設計の段階ではリレーショナルモデルよりも高レベルのデータモデルを使います．もっともよく使われるのが，ERモデル（実体関連モデル，Entity-Relationship model）です．ERモデルについては後で詳しく説明します．

図2　データベース設計の工程



論理データベース設計

概念データモデルを特定のデータモデル（本連載ではリレーショナルモデル）に対応したモデルに変換します．具体的には，ERモデルをリレーショナルモデルの表，SQLのテーブルに変換します．幸いにも，ERモデルからテーブルへの変換は機械的に行うことができます．ただし，この段階で得られたテーブルにはリレーショナルモデルの観点から見て好ましくない性質を持ったものが含まれる可能性があるため，その場合はさらにテーブル定義（スキーマ定義）を修正します．これを正規化と呼びます．

この段階でも，普通はシステムの動作効率のことはまだ考えません．

物理データベース設計

論理データベース設計の結果得られたテーブルを，効率を考慮して変更したり（非正規化），インデックスという高速アクセス機構を追加します．また，実際に使うデータベースシステム（ここではPostgreSQL）が持っている固有の機能を利用することもあります．

こうして実際に使用できるスキーマ定義が完成します．

◇ ◇ ◇

以上をまとめると図2のようになります．

ERモデル

ERモデルは，1976年にPeter P.Chenが提案したデータモデルです．リレーショナルモデルと違って，設計フェーズでのみ用いられ，ERモデルを直接実装したデータベースはありません．ERモデルには，Chenのオリジナルを発展させたさまざまなバリエーションがありますが，本稿ではオリジナルのモデルを用いま

注1）もっとも現実世界は途方もなく複雑ですから，すべてを写し取ることはできません．実際にはデータベースで管理したいものだけを抜き出します．

す.

ERモデルでは，実体（entity）と関連（relation）を使ってモデリングを行います．実体は，その名の通り世界を構成する実体を表します．それに対して関連は，実体の間の関係を表現したものです．実体と関連は，属性（attribute）を持つことができます．

ER図

ERモデルを記述する場合，通常ER図（Entity-Relationship diagram）を用います．図2で，「書籍」と書いてある四角形が実体を表しています．そこにぶら下がっている「書名」などと書いてある楕円が属性です．図2には書いてありませんが，関連は菱形で表現します．

ER図 記述ツール“dia”

ER図には，他にも図を描く上で細かな決まりがありますが，こういう決まりにしたがって手で絵を描くのはなかなか面倒です．tgifなどの汎用のお絵書きツールを使ったとしても，面倒さはさほど軽減されません．そこでお勧めするのがER図を描くための専用ツールです．

もちろん世の中にはER図を描く商用ソフトはたくさんあります．さすがにそれらのツールは良くできていて，ERモデルとしての決まりに反した絵は描けないようにチェックしてくれるとか，ER図から自動的にスキーマを生成してくれるなどの機能が付いていますが，それなりのお値段がしますし，UNIX系のプラットフォームで使える製品はほとんどないのが気に入りません．

そこで本稿で使うのが“dia”（http://www.lysator.liu.se/~alla/dia/　図3参照）というフリーソフトウェア[注2]です．

diaは汎用のダイアグラム描画ツールで，ER図の他，UML図，回路図なども書けます．書いた図はdiaのオリジナルフォーマット[注3]で保存しますが，pngやepsなどのフォーマットにしてエキスポートすることもできます．

また，ER図の中に矩形，楕円などの図形や文字，さらにはグラフィックスを貼りこむこともできます．実は，図2もdiaを使って書いたものです．

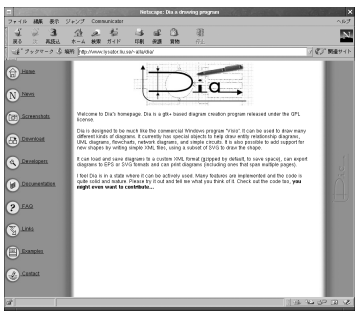
diaを使っていて一番嬉しいのは，オブジェクトとオブジェクトを結んだ線がオブジェクトに「固定」され，オブジェクトが動くと自動的にそれらの線が追随することです．これでER図を修正する際にかなり楽ができます．その他，グループ化やレイヤなど，お絵書きツールとしての基本機能も備わっています（筆者はレイヤ機能は使っていませんが）．

diaの導入

筆者の使用中のプラットフォームVine Linux 2.1では，Vine Plusという追加パッケージにdiaのRPMが入っていたので，それを安直にインストールしました．バージョンは0.86となっています． また，diaはGtkを使うので，Gtkを入れておくことも必要です．

ソースからdiaをコンパイルする場合は，前述のdiaのページからソースを入手します．本稿執筆時点では，0.88というのが最新版のようです．インストールそのものは容易で，configure，make，make installでとりあえず使えるようになるはずですが，筆者の環境ではドキュメントのインストール中にjadeがコンパイルエラーを起こす，日本語が使用できないという問

図3　diaのWebページ



注2）ライセンス形態はGPLです．
注3）実際にはgzipで圧縮されたXMLのテキストになっています．

題がありました.

追求している時間もないので，本稿では，RPM版を使用することにします.

diaの使い方

diaのドキュメントは，ソース中のdoc/en/dia-manual/index.htmlです．インストール後は/usr/local/share/dia/help/en/index.htmlにコピーされます[注4]．詳しい使い方はこれらのドキュメントを見ていただくとして，ここではER図を描くためのごく簡単な説明をします.

とりあえず，引数なしでdiaを起動すると，図4の画面になります.

起動した端末には，なぜか，

```
Gtk-CRITICAL **: file gtkwidget.c: line
3306     (gtk_widget_set_sensitive):
assertion `widget != NULL' failed.
```

のようなメッセージが盛大に出ますが，とりあえず実行には支障がないようなので無視することにします，

1番上には「ファイル」というプルダウンメニューがあり，新規ダイアグラムの作成，既存ダイアグラムの読み込みなどができます．その下は「ツールボックス」と呼ばれ，アイコンをクリックして描画したいオブジェクトを選択できます.

真ん中あたりに“Circuit”（電気回路図）という選択リストがありますが，このボタンを選択すると，他の種類の図も選択できます．さっそく“ER”を選んでみましょう（図5）.

中央よりやや下あたりのパネルの表示が変わり，ER図用のウィジェットが表示されています．ここで，ファイルメニューから“新規ダイアグラム”を選ぶと「キャンバス」が表示され，図を描けるようになりま

図4 diaの起動画面

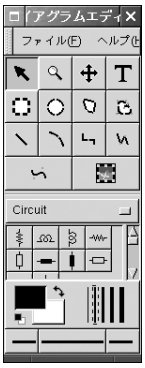


図5 ER図を選択

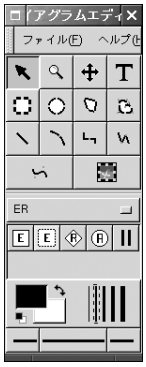


図6 実体を作成

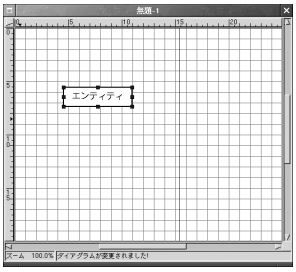


図7 プロパティの表示

注4）rpm版のdiaでは，なぜかドキュメントが見つかりませんでした.

す．試しに，“E”というウィジェットを選び，キャンバス上でマウスの第1ボタンをクリックすると，「エンティティ」と書かれた箱が出現するはずです．これがER図では実体を表します（図6）．

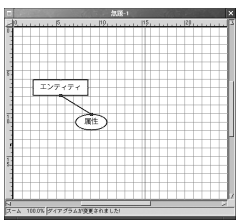
箱の周りの線にところどころ四角い小さな箱が付いていますが，この状態はこのオブジェクトが選択中であることを示します．ここで第1ボタンをダブルクリックすると，プロパティが表示され，箱の中の文字を変更したり，線幅，色などを変えることができるようになります（図7）．オブジェクトの削除は，オブジェクトを選択後，マウスの第3ボタンを押してメニューから「編集」➡「削除」を選ぶか，CTRL＋D（コントロールを押しながらdを押す）です．なお，この編集メニューからコピー，ペースト，undoなどの操作を行うこともできます．オブジェクトの移動は，マウスの第1ボタンを押して選択状態にしたままドラッグすればOKです．

次に，この実体に属性を付与してみましょう．先ほど“E”を選んだパネルから，今度は“A”を選び，キャンバス上でクリックします．「属性」と書かれた楕円が出現するはずです．これが属性です．

ER図では，実体と属性は線で結びますので，diaのメニューの中から線の両端に緑の丸が付いているアイコンを選択し，線を描いてみましょう．線のスタート地点でマウスの第1ボタンを押し，そのまま終了地点までドラッグしてボタンを離します．これで線が引けました（図8）．

このとき，線の両端の箱が赤なのか緑なのかで動作が異なってきます．緑の場合は普通の線ですが，赤の状態では，線がオブジェクトに「くっついた」状態となり，オブジェクトの移動に線が追従します．この状態にするためには，オブジェクトの境界線上のひげのような細い線が出ているところに線の端を持ってきます．色が変わるはずです．

図8　実体と属性を線で結ぶ



ER図の書き方

それでは，早速diaを使ってER図を書いてみたいところですが，その前にER図の記法を説明します．図9を併せてご覧ください．

実体

四角い箱で表します．

属性

楕円で表します．属性にはさらにいくつかの付加的な特性を付与できます．diaでは，プロパティで設定できます（図10）．

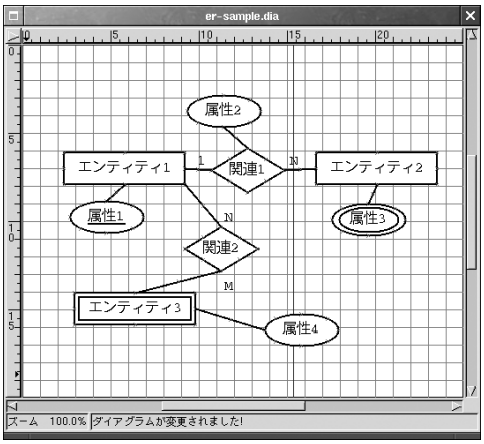
●キー

その実体を唯一識別することのできる属性です．キーについての詳細は連載第1回を御覧ください．ER図では，キー属性は下線で表現します．図9では属性1がその例です．

●複数値

値が複数個ある属性です．たとえば，「人」という実体に「趣味」という属性を設けた場合，人によって

図9　ER図の例



は複数の趣味を持つので，複数値を持つ属性を使用します．

関連

菱形で表します．関連にも付加的な特性があり，プロパティで設定できます（図11）．

●左反転／右反転

関連の左右の実体の間に，数的な制約がある場合に記述します．たとえば図9では，1個のエンティティ1に対してエンティティ2が複数個関係を持つことを示しています．このような場合，「左反転」に“1”，「右反転」に“N”と書けば，エンティティ1とエンティティ2が1対Nの関係にあることを表現できます．

一方，エンティティ1とエンティティ3は多対多の関係にありますが，上下の位置にN，Mが来てほしいので，プロパティの「回転」をチェックしています．

弱実体

実体の一種ですが，必ず他の実体と関係を持たないと独立して存在できないような実体のことで，diaのアイコンでは点線で囲まれた“E”が表しており，図9中ではエンティティ3のように2重線の矩形で表現されています．

実世界の例で言うと，たとえばある品物の実体に対し，発注記録が弱実体として定義されることがあります．このような場合，「発注記録番号」は品物とペアでないと実際にはどの品物の記録なのかわかりません．また，品物が削除されると，発注記録という実体も同時に削除されてしまいます．このように，弱実体は一人前ではない実体と言えます．

雑誌記事データベースを作ろう

それでは，いよいよ実際にデータベースの概念設計に入りましょう．ここで例題として取り上げるのは，「雑誌記事データベース」です．

皆さんもそうだと思いますが，私も毎月たくさんのコンピュータ系の雑誌を読みます．中には「お，これは使える」という記事もあり，そのときは覚えているのですが，3ヵ月も経てばどの雑誌の記事だったかすら覚えていない有り様になってしまいます．また，何か調べものをしていて，「あ，あのことはそういえばどっかの記事に書いてあったはず」と思うこともよくあります．そこで，このような状態を少しでも改善するために，雑誌の記事をデータベース化しようと思います．

もちろん，雑誌記事そのものをデータベースにいれるのではなく，雑誌名やタイトル，著者，キーワードなどの情報だけをデータベースに入れるわけです[注5]．

雑誌記事データベースで扱うデータ

まず最初にしなければならないのは，実世界のデータのうちどこまでを扱うかということを決めることです．当然のことながら，現実世界をすべてモデル化す

図10　属性のプロパティ

図11　関連のプロパティ

注5）実際にはそういう項目を入力するだけでもかなり面倒なので，果たしてそういうデータベースを維持していけるものかどうかはわかりませんが….

ることはできませんから，どこかで妥協が必要です．

とりあえず雑誌という存在を中心に考えてみると，

- 雑誌を発行している出版社
- 雑誌そのもの
- 雑誌に記事を書いている執筆者

くらいは最低限モデル化する必要がありそうです．

ER図の最初のバージョン

さていよいよER図を描くわけですが，ものの順序としては，まず実体を抽出し，次にその間を関連で結んでいくということがよく行われます．そこでまず上に挙げた「出版社」「雑誌」「執筆者」を実体とし，その間を関係で結んでER図の第1バージョンを作ってみましょう．なお，とりあえず属性としては，各実体を識別するキーとなる属性のみ付与しておきます．細かな属性は後から考えることにします（図12）．

ある出版社は，複数の雑誌を発行することもあるので，出版社と雑誌の間の関係は1:Nです．一方，雑誌の執筆者は複数いますし，また執筆者はある雑誌専属ということはなくて，複数の雑誌に執筆する可能性があるので，雑誌と執筆者の間の関係は多対多，N:Mと表現されています．

さて，このER図をよく眺めてみると，いろいろ問題があることがわかります．

問題１：どの記事をどの著者が書いたのかわからない

ある著者がどの雑誌に執筆したかはわかりますが，どの記事を書いたのかという情報がないからです．とりあえず「執筆」関係の属性に記事情報を追加する方法もありますが，記事に関するさまざま情報を検索したくなったときにさらに「執筆」関係にその他の属性を追加しなければならなくなり，不自然です．ここは「記事」という実体を追加したほうがよさそうです．

問題２：ある執筆者が雑誌のどの号に記事を書いたのかわからない

これは，「雑誌」という実体にどの号かという情報がないからです．では「雑誌」に「号」という属性を追加すればそれで済むのでしょうか？

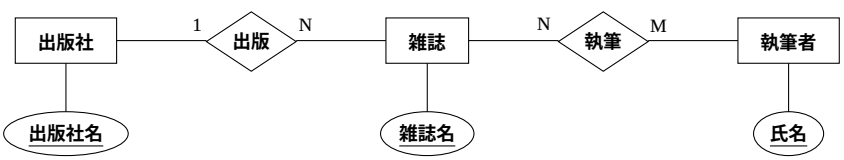
解決策

その前に「雑誌」という実体は実際には現実世界の何に対応しているのかもう少し考えてみましょう．①「ある名前の雑誌」という抽象的なものでしょうか？それとも②実際に発刊された「雑誌XXXX YY年ZZ月号」，などという具体的な個々の雑誌でしょうか？

①と解釈すると，「ある執筆者が雑誌のどの号に記事を書いたのかわからない」という問題が解決できなくなります．では②が適当かというとそうでもありません．たとえば，「ある出版社が出版している雑誌の一覧を知りたい」という問い合わせにはどちらかというと①が適しています．また，企画段階の雑誌では，そもそも現実の雑誌がまだ存在しないため，②では対応できません．

このジレンマは，「雑誌」実体を①と②の2つの実体に分解すれば解決できます．つまり，今までは単に「雑誌」というひとまとめの実体であったものを，ある雑誌に関する基本的な属性，たとえば雑誌の名前その他の属性を持つ実体と，実際に発行された雑誌

図12　ER図の最初のバージョン



2 データベース設計（1）

に関する実体の2つに分けるわけです．そして，執筆者の実体は後者の雑誌実体と関連するようにします．

ER図の改良バージョン

以上を反映した新しいER図を図13に示します．

「雑誌」は雑誌自体の情報を格納する「雑誌」と実際に発行された雑誌である「発行雑誌」の2つに分かれました．また，雑誌の記事は「記事」という実体で表現し，「執筆者」は「記事」と関連付けされました．

また，「雑誌」「発行雑誌」「記事」は弱実体に変更されました．その理由を説明します．

まず「雑誌」ですが，雑誌は出版社なくしては存在し得えないので，その意味で弱実体であると言えます．この場合「雑誌」に対して「出版社」を識別上のオーナと呼びます．「発行雑誌」はもちろん「雑誌」の従属的な存在ですし，「記事」も「発行雑誌」が存在しなくては意味がない実体なので，これらも弱実体となります．

後は必要に応じて適当に属性を加えて肉付けします．実際に検索したい属性を考慮しながら加えていきます（図14）．これで概念データベース設計のフェーズは終わりです．

なお，「記事」の「キーワード」属性は二重の楕円になっていますが，これは属性の値が複数値であることを表しています．記事に付けたいキーワードは複数あり得るのでこうしています．

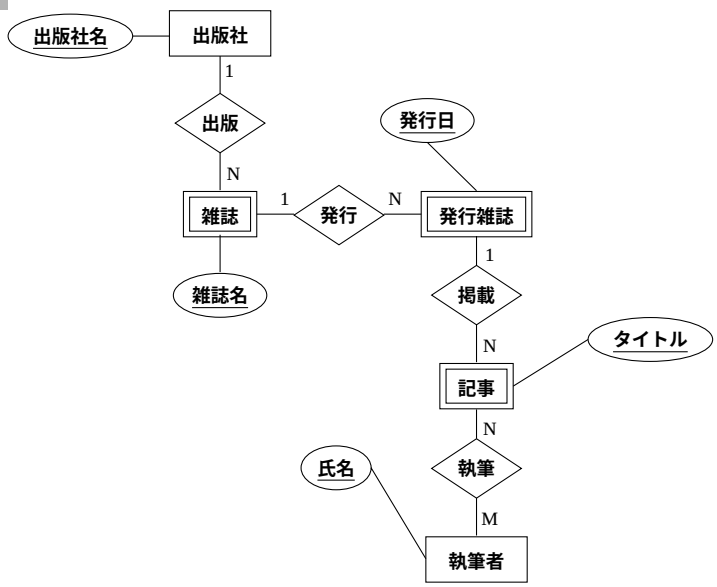
ERモデルからリレーショナルモデルへの変換

次のステップは，論理データベース設計の第1段階である，ERモデルのリレーショナルモデルへの変換です．この変換は機械的にできることが知られています．以下にその規則を示します．

①実体と弱実体はテーブルに変換します．その際，属性はそのテーブルの列になります．

②キー属性は主キーへと変換します．ただし，弱実体の場合は，識別上のオーナに対応するテーブルの主キーを列に加え，キー属性と合わせてこのテーブルの主キーとします．たとえば，「発行雑誌」の主キ

図13　ER図改良バージョン



ーは，「発行年月日」と「雑誌名」の組み合わせになります．

③1:Nの関連では，“N”側のテーブルの主キーに“1”側のテーブルの主キーを追加して主キーとします．たとえば「雑誌」の主キーは，「出版社名」と「雑誌名」の組合わせになります．なお，関連に属性がある場合は，その属性も“N”側のテーブルの列に追加します．

④N:Mの関連では，関連をテーブルに変換し，“N”側のテーブルの主キーと“M”側のテーブルの主キーを組み合わせてそのテーブルの主キーとします．たとえば，「執筆」のN側である「記事」の主キーは「出版社」，「雑誌名」，「発行日」，「タイトル」であり，M側である「執筆者」の主キーは「氏名」ですから，「執筆」の主キーは「出版社」，「雑誌名」，「発行日」，「タイトル」，「氏名」となります．なお，関連に属性がある場合は，その属性もこのテーブルの列に追加します．

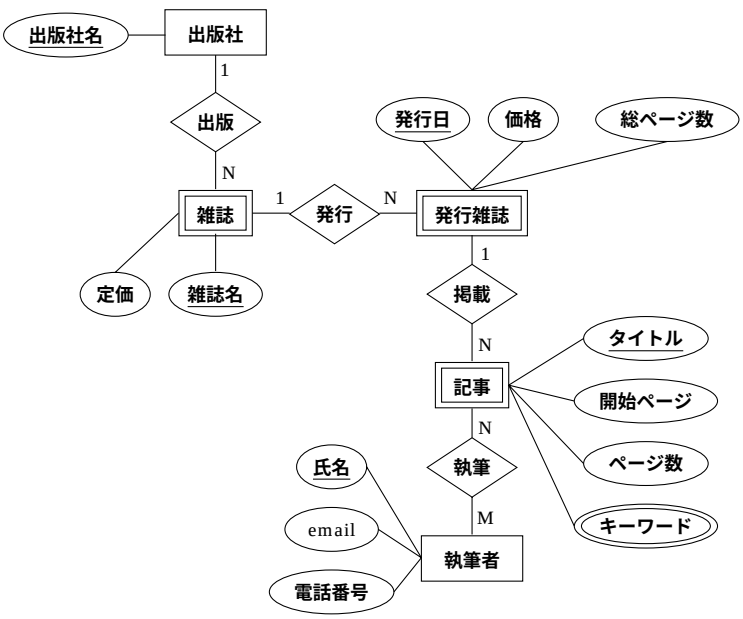
実際にはこれだけでは不十分で，以下のような考慮が必要です．

❶属性に対応する列のデータ型の決定
❷NOT NULL制約
❸参照整合性制約
❹UNIQUE制約
❺CHECK制約

これらをすべて決めてからテーブルへの変換を行うこともできますし，徐々にこれらを取り入れてテーブル定義を「進化」させるさせることもできます．一挙にすべてを決める方法では，設計のミスが後でわかってER図の修正が必要になったときに手戻りが発生します．逆に徐々にテーブルを進化させる方法では，制約を追加したときにはじめてわかるような問題点が予測しにくい欠点があります．要するに一長一短なわけで，ER図の規模や複雑さ，設計者の習熟度に応じてやりやすい方法を選べばよいでしょう．つまり，ER図の規模が小さく，設計者が熟練している場合には，最初からすべてを一気に決める方法を取ってもリスクが小さいと考えられます．

ここでは，説明の都合もあるので，とりあえず❶と

図14　ER図さらに改良バージョン



2 データベース設計（1）

❷を決めてスキーマ定義を作り，それを進化させていく方法をとります．

属性に対応する列のデータ型の決定

「出版社名」のような属性は文字列であり，SQLでは以下のようなデータ型が候補になり得ます．

●CHARACTER（短縮形はCHAR）

“CHAR(10)”のように文字列の長さを指定します．長さの単位は処理系によって異なり，本稿で使用するPostgreSQLでは単位はバイトです．PostgreSQLでは日本語の文字コードは主にEUC[注6]を使用するので，漢字や平仮名などのいわゆる「全角」文字では1文字で2～3バイトを消費します．たとえば「日本語」という文字列を格納するためには6バイトを必要とします．

CHAR型では，文字列が指定した長さに満たない場合は，空白文字が後ろに詰められます．CHAR型は，その列の文字列長がほとんど変化しない場合に適しています．

●CHARACTER VARYING（短縮形はVARCHAR）

CHAR型と同様，“VARCHAR(10)”のように文字列の長さを指定します．CHAR型との違いは，文字列が指定した長さに満たない場合でも空白文字が後ろに詰められない点です．VARCHAR型は，属性の文字列長が変化する場合に適しています．

●TEXT

CHAR，VARCHARがSQL標準のデータ型であるのに対し，TEXTはPostgreSQL固有のデータ型です．CHAR，VARCHARと違って長さの指定が不要であることが特徴です．PostgreSQLの現在のバージョンである7.1では，最大約1Gバイト（＝1024Mバイト）までの文字列が格納できます．また，7.1ではデータ格納時に圧縮がかかるので，実際には1Gバイト以上のデータが格納できる場合もあります．

TEXT型は，文字列の長さの上限があらかじめ確定できず，かつ他のデータベースとの互換性を重視しない場合に利用するとよいでしょう．

●数値データ型

一方，数値を格納するためのデータ型も複数あります．表1をご覧ください．扱う数値の範囲や，必要な精度に応じて選択します．一般に高精度なデータ型ほどデータベース上での領域を多く必要としますし，NUMERICやDECIMALは非常に高精度である反面，ソフトウェアで数値計算をしている関係上処理速度は他のデータ型よりもかなり見劣りします．

●その他のデータ型

時刻，日付関係のデータ型には表2のものがあります．

NOT NULL制約

その列にNULLを許すかどうかを決める必要があります．NULLとは，列の値が未定，不明，あるいはそ

▼表1　数値データ型

データ型	特徴	可能な値の範囲	必要記憶容量
SMALLINT	整数のみ	-32768～+32767	2バイト
INTEGER	整数のみ	-2147483648～+2147483647	4バイト
BIGINT	整数のみ	-9223372036854775808～+9223372036854775807	8バイト
REAL	浮動小数点	ー	4バイト
FLOAT	浮動小数点	ー	8バイト
NUMERIC	高精度な数値計算に使用する	1000桁までの精度の整数と小数	可変長
DECIMAL	高精度な数値計算に使用する	1000桁までの精度の整数と小数	可変長

注6）Extended Unix Codeの略で，主にUNIXの世界で利用されている文字コードです．

もそも値を適用すること自体が不適切な場合に使用する特別な属性値です．

ほとんどの列では，NULLが格納されることを想定しなくて良いはずです．どうしてもNULLを使う必要がある場合は，何かテーブルの設計に問題がないかもう1度見直してみたほうが良いかもしれません．

NOT NULL制約を追加するには，列を宣言するときに以下の例のように“NOT NULL”というキーワードを追加します．

```
CREATE TABLE t1 (i INTEGER NOT NULL);
```

なお，主キー制約は暗黙のうちにNOT NULL制約を含んでいるので，主キーの列にはNOT NULL宣言は必要ありません．

スキーマ定義の最初のバージョン

では，さっそくSQLのスキーマ定義（CREATE TABLE文）を書いてみましょう．ここでは，テーブル名，属性名はER図と同じ，つまり日本語を使用します．前回述べたように，実際にプロジェクトで使用するスキーマ定義では日本語の使用は好ましくありませんが，まだ設計の途中なので，検討のしやすさを重視してこうします（リスト1）．

ここでは，単純さのために，文字列を格納する列はTEXT型，数字を格納する列はINTEGERにしました．記事テーブルのキーワード列は，複数の値を持つ列を表現する方法がSQLにはないため[注7]，キーワード数を最大3つまでと割り切ってこのようにしています．決して良い方法ではありませんが，この段階ではとりあえずこれでよしとします[注8]．

▼表2　その他のデータ型

データ型	使用目的
DATE	年月日
TIME	時分秒
TIMESTAMP	年月日時分秒+タイムゾーン
INTERVAL	時間間隔

リスト1　スキーマ定義最初のバージョン

```
DROP TABLE 出版社;
CREATE TABLE 出版社 (
     出版社名 TEXT PRIMARY KEY
);

DROP TABLE 雑誌;
CREATE TABLE 雑誌 (
     出版社名 TEXT NOT NULL,
     雑誌名 TEXT NOT NULL,
     定価 INTEGER NOT NULL,
     PRIMARY KEY (出版社名,雑誌名)
);

DROP TABLE 発行雑誌;
CREATE TABLE 発行雑誌 (
     出版社名 TEXT NOT NULL,
     雑誌名 TEXT NOT NULL,
     発行日 DATE NOT NULL,
     価格 INTEGER NOT NULL,
     総ページ数 INTEGER NOT NULL,
     PRIMARY KEY(出版社名,雑誌名,発行日)
);

DROP TABLE 記事;
CREATE TABLE 記事 (
     出版社名 TEXT NOT NULL,
     雑誌名 TEXT NOT NULL,
     発行日 DATE NOT NULL,
     タイトル TEXT NOT NULL,
     開始ページ INTEGER NOT NULL,
     ページ数 INTEGER NOT NULL,
     キーワード1 TEXT,
     キーワード2 TEXT,
     キーワード3 TEXT,
     PRIMARY KEY(出版社名,雑誌名,発行日,タイトル)
);

DROP TABLE 執筆;
CREATE TABLE 執筆 (
     出版社名 TEXT NOT NULL,
     雑誌名 TEXT NOT NULL,
     発行日 DATE NOT NULL,
     タイトル TEXT NOT NULL,
     氏名 TEXT NOT NULL,
     PRIMARY KEY(出版社名,雑誌名,発行日,タイトル,氏名)
);

DROP TABLE 執筆者;
CREATE TABLE 執筆者 (
     氏名 TEXT PRIMARY KEY,
     email TEXT,
     電話番号 TEXT
);
```

注7）実はPostgreSQLでは配列を扱うことができるため，TEXT[ ]のようにすれば複数の値を持つ属性をそのまま列に変換できます．ただし，このようにすると検索に制約が出てきます．

注8）次回で対処します．

2 データベース設計（1）

UNIQUE制約とCHECK制約

次のステップは，UNIQUE制約とCHECK制約の追加です．

●UNIQUE制約

UNIQUE制約とは，同じテーブルの中でその列が重複した値を持たないという制約です．ただし，NULLはいくつあっても重複とはみなしません．今回は，執筆者のemailと電話番号に適用しましょう．UNIQUE制約の書き方は簡単で，

```
email TEXT UNIQUE,
```

のように，“UNIQUE”というキーワードを追加するだけです．

●CHECK制約

列の値に関する任意の論理式を書き[注9]，それが満たされることを要求するものです．

たとえば，

```
定価 > 0
```

は，定価が0より大きいことを要求します．ANDを用いて

```
10000 > 定価 AND 定価 > 0
```

とし，定価が0から10000の間であることも表現できます．CHECK制約は，

```
定価 INTEGER CHECK(定価 > 0)
```

のように“CHECK”というキーワードの後に( )を付け，その中に論理式を書きます．複数の列にまたがる制約を書きたい場合は，**テーブル制約**として書きます．テーブル制約は，列の宣言の後にCONSTRAINTキーワード，制約の名前[注10]，そして制約自身を書きます．例を示します．

```
CONSTRAINT mycheck CHECK(価格 > 0 AND 総ページ数 > 0)
```

テーブル制約として制約を記述しておくと，制約に違反するようなデータの追加，更新があったときに制約の名前が一緒に表示されて，エラー原因が明白になるので，できるだけテーブル制約で記述することをお勧めします[注11]．

こうしたチェックをアプリケーションプログラムで行うことも可能ですが，アプリケーションプログラムにバグがあったり，処理忘れがある可能性を考えると，できるだけのチェックをデータベース側で行っておくべきであると言えます．

スキーマ定義改良バージョン

以上，UNIQUE制約とCHECK制約を追加したものを示します（**リスト2**）．

おわりに

次のステップは参照整合性制約の追加ですが，誌面が尽きてしまったので次回で解説します．

次回は，参照整合性制約を付け加えるなどして論理データベース設計フェーズを終了，続いて物理データベース設計を行ってスキーマ定義を完成させる予定です．WD

注9）関数を含むような論理式も可能です．たとえばCHECK(character_length(出版社) > 8)で，出版社名が8文字以上であるという制約も記述できます．ここでcharacter_lengthは，SQLの標準関数で，引数文字列の文字数（バイト数ではありません）を返します．なおPostgreSQLでは今のところ，SELECT文を含む論理式はCHECK制約に記述できません．

注10）制約の名前はそのデータベースの中でユニークである必要があります．そこで筆者は，テーブル名_列名_checkのような名前を付けます．

注11）たとえば，リスト2のスキーマ定義で，出版社名として‘ ’（空文字列）を登録しようとすると，ERROR: ExecAppend: rejected due to CHECK constraint 出版社_出版社名_checkというエラーメッセージが出力されます．

リスト2　スキーマ定義改良バージョン

```
DROP TABLE 出版社;
CREATE TABLE 出版社 (
     出版社名 TEXT PRIMARY KEY,
     CONSTRAINT 出版社_出版社名_check CHECK(character_length(出版社名) > 0)
);

DROP TABLE 雑誌;
CREATE TABLE 雑誌 (
     出版社名 TEXT NOT NULL,
     雑誌名 TEXT NOT NULL,
     定価 INTEGER NOT NULL,
     PRIMARY KEY (出版社名,雑誌名),
     CONSTRAINT 雑誌_出版社名_check CHECK(character_length(出版社名) > 0),
     CONSTRAINT 雑誌_雑誌名_check CHECK(character_length(雑誌名) > 0),
     CONSTRAINT 雑誌_定価_check CHECK(定価 > 0)
);

DROP TABLE 発行雑誌;
CREATE TABLE 発行雑誌 (
     出版社名 TEXT NOT NULL,
     雑誌名 TEXT NOT NULL,
     発行日 DATE NOT NULL,
     価格 INTEGER NOT NULL,
     総ページ数 INTEGER NOT NULL,
     PRIMARY KEY(出版社名,雑誌名,発行日),
     CONSTRAINT 発行雑誌_出版社名_check CHECK(character_length(出版社名) > 0),
     CONSTRAINT 発行雑誌_雑誌名_check CHECK(character_length(雑誌名) > 0),
     CONSTRAINT 発行雑誌_価格_check CHECK(価格 > 0),
     CONSTRAINT 発行雑誌_総ページ数_check CHECK(総ページ数 > 0)
);

DROP TABLE 記事;
CREATE TABLE 記事 (
     出版社名 TEXT NOT NULL,
     雑誌名 TEXT NOT NULL,
     発行日 DATE NOT NULL,
     タイトル TEXT NOT NULL,
     開始ページ INTEGER NOT NULL,
     ページ数 INTEGER NOT NULL,
     キーワード1 TEXT,
     キーワード2 TEXT,
     キーワード3 TEXT,
     PRIMARY KEY(出版社名,雑誌名,発行日,タイトル),
     CONSTRAINT 記事_出版社名_check CHECK(character_length(出版社名) > 0),
     CONSTRAINT 記事_雑誌名_check CHECK(character_length(雑誌名) > 0),
     CONSTRAINT 記事_タイトル_check CHECK(character_length(タイトル) > 0),
     CONSTRAINT 記事_開始ページ_check CHECK(開始ページ > 0),
     CONSTRAINT 記事_ページ数_check CHECK(ページ数 > 0)
);

DROP TABLE 執筆;
CREATE TABLE 執筆 (
     出版社名 TEXT NOT NULL,
     雑誌名 TEXT NOT NULL,
     発行日 DATE NOT NULL,
     タイトル TEXT NOT NULL,
     氏名 TEXT NOT NULL,
     PRIMARY KEY(出版社名,雑誌名,発行日,タイトル,氏名),
     CONSTRAINT 執筆_出版社名_check CHECK(character_length(出版社名) > 0),
     CONSTRAINT 執筆_雑誌名_check CHECK(character_length(雑誌名) > 0),
     CONSTRAINT 執筆_タイトル_check CHECK(character_length(タイトル) > 0),
     CONSTRAINT 執筆_氏名_check CHECK(character_length(氏名) > 0)
);

DROP TABLE 執筆者;
CREATE TABLE 執筆者 (
     氏名 TEXT PRIMARY KEY,
     email TEXT UNIQUE,
     電話番号 TEXT UNIQUE,
     CONSTRAINT 執筆者_氏名_check CHECK(character_length(氏名) > 0)
);
```