情報システムとWeb技術＃2: 情報システムとデータベース

久野　靖[*]

2015.11.24

1　データベースの基礎概念

1.1　なぜデータベースか?

今日のWebが「便利」である背景には、その上で多くの用事が済んでしまうということが挙げられます。オンラインショップやオークションサイトではほとんどあらゆるものが購入できますし、銀行の振り込みも、送った荷物の確認も、すべてブラウザ経由で行えます。

では、こういうサービスの内実はどうなっているのでしょうか? Webが提供しているのはあくまでもサービスに対するアクセス機能であって、サービスの実体はWebサーバの裏側で働いている各種の業務システムによって支えられています。

たとえば、オンラインショップの場合で考えると、支払の情報や、現在の配達状況、商品の在庫など、様々なデータを管理しなければならない事がわかります。当然ながら、これらのデータはなくなってはいけませんし、データ間で矛盾があってもいけません。このように考えてくると、業務システムの中核には「データを確実に保管し出し入れする」機構、すなわち**データベース**があることが分かって来ます (図1)。[1]

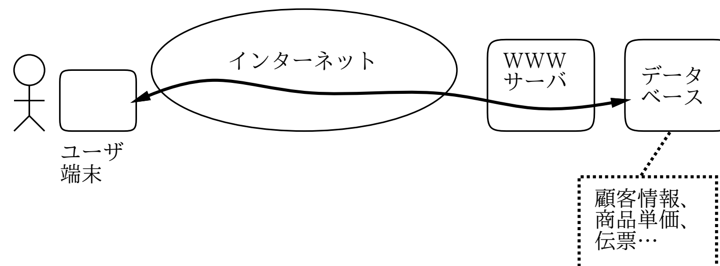

図1: Webから利用できる業務システムの構造

実際のWebサイトでは、多数のユーザから大量のアクセスが来た場合に備えてWebサーバやデータベースサーバを複数用意して**負荷分散**を行い、また一部にトラブルがあってもサービスが継続できるように**障害対策**の仕組みを準備するなどの工夫をしますが、今回ではそこまで立ち入ることはせず、基本的なデータベースの原理と機能について説明して行きます。

1.2　データベースとは?

データベースとは、ごく簡単に言えば「統合化された」「共有可能な」データの格納場所、ということになります。しかし、データをUnixのファイルに格納しておいたらファイルシステムという枠

[*]経営システム科学専攻

[1]もちろん、お金の計算処理などもバグがあっては困りますが、この手の計算は基本的には「普通の足し算引き算」ですから、計算そのものよりは、データをきちんと管理することの方がずっと大変なわけです。

かしますが、それでは駄目なのでしょうか?

もうすこし詳しく考えてみましょう。旧来のやり方、つまりプログラムがファイルを読み書きする、という形でデータを処理していると、各プログラムはその処理に必要な「データファイル (群)」と組み合わせて使う、ということになります。しかしこの方法にはいろいろな問題があります。

たとえば、ある企業の給与計算プログラムがあったとすると、そのプログラムが扱うデータファイルには当然、社員の情報…社員番号とか、氏名とか、年齢とか、部署とかの情報が含まれています。さて、この企業が新たに人事管理のためのプログラムを開発するとしましょう。このプログラムも当然、社員の情報を必要とするでしょう。では、これらの情報をどのようにして取り込んだらいいでしょうか?

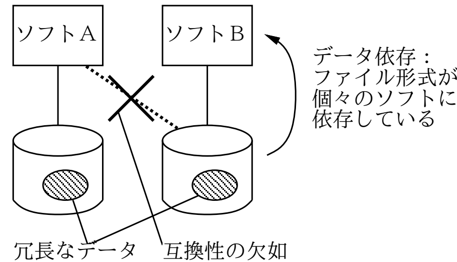

図 2: データ依存

まず頭に浮かぶのは、給与計算プログラムで使っていたファイルを人事管理プログラムでも利用する、という方法でしょうか。しかし、このファイルには給与計算に必要なデータしか入っていなかったので、人事管理のためのデータを追加しなければなりません。すると、そのためにファイル内のデータ形式を変更することになり、それに対応して給与計算プログラムも (新しく追加された自分には不要なデータを無視するように) 修正しなければなりません。このように、複数のプログラムが 1 つのファイルに係わることになると、どれか 1 つのプログラムの都合でファイルの形式を変更した時には、全部のプログラムを修正する必要があります。これでは、プログラムの数が少し多くなると、とてもやりきれません。

このように、ファイルの形式が個々のプログラムに依存していることを**データ依存**と呼びます (図 2)。データ依存が存在すると、次のような問題があるわけです。

- あるデータを複数のプログラムで共同利用するのが難しい
- プログラムの都合に応じてデータを手直ししたり、データの手直しに応じてプログラムを手直しすることが重荷となる

では別の方法として、給与計算プログラムのデータをコピーして来て、新しく人事管理プログラム用のファイルをそれ専用に作るのではどうでしょうか? 一見よさそうですが、今度は同じ情報 (社員番号と氏名や年齢の対応など) が複数のファイルに重複して保持されることになります。このような**冗長なデータ**には多くの問題があります。たとえば社員が退職したら、社員情報を使うすべてのプログラム (とても沢山あるでしょうね!) のファイルから、その社員のデータを削除して廻らなければなりません。そしてある日突然、人事管理ファイルには登録されている社員がなぜか給与計算ファイルにはない、ということが分かったとしたらどうしたらよいでしょうね? つまり、冗長なデータというのは次のような問題点があるわけです。

- 更新の手間がひどく掛かり、
- **データの矛盾**が発生し得る

それでは、データベースを使うとどうなるのでしょうか? データベースを使うということは、概念的にはすべてのプログラム群が必要とするデータを 1 つの巨大なファイル (文字通りデータの「基盤」) に入れてしまうことです (図 3)。これによって、社員番号と氏名等の対応は 1 箇所だけに格納され (冗長度がなくなり)、更新や矛盾の問題がなくなります。

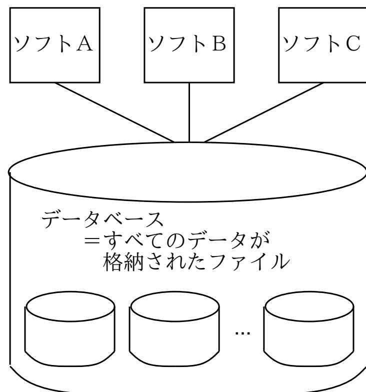

図 3: データベースの概念

では、データ依存の方はどうでしょうか? データベースでは、個々のプログラムは**ビュー**と呼ばれる自分専用のファイル形式の「まぼろし」を通じてデータにアクセスします (図 4)。そして、データ本体はデータベース内にこれらのビュー (ひいては個々のプログラム) とは独立した「中立の」形で格納されています。つまり、プログラムとデータ本体とは互いに依存しない形で存在できるわけです。これをデータ依存と対比して、**データ独立**と呼びます。

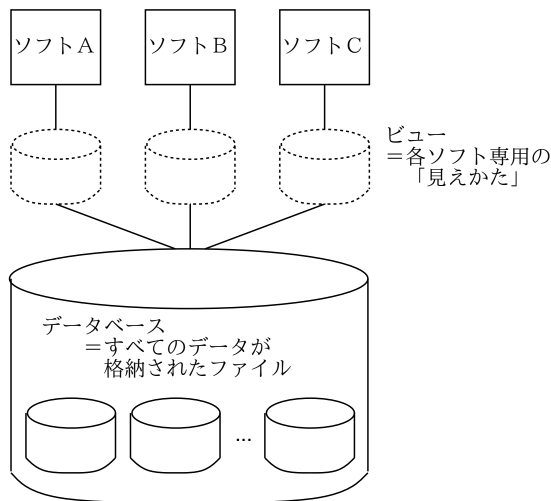

図 4: データベースとビュー

ビューは、この中立のデータから各プログラムの必要とする形式への「対応関係」を定めていて、プログラムの必要に応じて自由に修正したり追加できます。また逆に、新しいデータ要素を追加したり、管理の都合などでデータベース内部の構成を変更した場合でも、ビューをそれに合わせて修正すれば、プログラム群には手を加える必要がありません。

このようにして、データベースでは多数のプログラム (やユーザ) が必要とするデータを 1 箇所に「統合して」保管でき、「共有」させることができるのです。加えてデータベースでは、ただのファイルにはないような、次のような機能を追加して提供できます。

えばさまざまな構造を持った検索が行える

- **並行制御** — 複数のプログラムが並行してデータをアクセス/ 更新しても正しく処理されるように管理
- **排他制御** — あるプログラムがアクセスしているデータが他のプログラムによって変更されたり覗き見されないように制御する
- **障害回復** — システムやアプリケーションに異常が起きた場合に、正しい状況に回復させるための手段を用意する
- **トランザクション** — 複数の操作をひとまとまりのものとして管理し、相互に干渉しないよう制御したり、エラーがあっても中途半端な状態がデータベースに残ったりしないようにする
- **整合性管理** — データ間の整合性を保つ条件を設定しておくと、それを自動的に適用したりチェックしてくれる。
- **セキュリテイ** — どのデータを誰がアクセスできるかについて、細かく設定管理できる

これらの機能は互いに独立しているというわけではありません。たとえば、トランザクション機能は並行制御や障害回復の 1 手段として使われますし、並行制御のためには (内部的に) 排他制御機構が使われます。

演習 あなたやあなたの周辺では、次のような場合から 1 つ以上 (できれば全部) 選び、情報を整理・管理するのに、どのような方法を使っているか、その方法の利点や弱点は何かを振り返ってみて報告しなさい (一般的過ぎるようなら、特定種類のものに絞ってもよい)。いずれもできれば、コンピュータを使う場合と使わない場合の両方について挙げ、それらの比較も行えるとよいでしょう。

a. 自分が個人で保持・管理・活用している情報の管理方法。
b. 自分が仕事のために同僚と共同で保持・管理・活用している情報の管理方法。
c. 自分が仕事以外の場面で知合いや仲間と共同で保持・管理・活用している情報の管理方法。

1.4 データベースの構造と DBMS

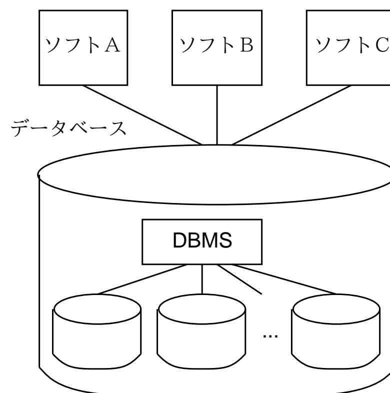

図 5: データベースと DBMS

機能や役割りは分かったとして、データベースはどのようにして実現されているのでしょうか? プロセスやファイルシステムが裸の CPU やディスク上に OS というソフトウェアの働きによって実現されているのと同様、データベースはプロセスやファイルシステムの上で動く **DBMS**(データベース

能を実現するのも DBMS の役割です。DBMS はかつては大規模なソフトウェアであり、高価なソフトウェア製品としてしか入手できませんでしたが、今日ではフリーソフトとして配布されているものもあります。売りものの DBMS としては Oracle、Sybase、DB2、MS SQLServer など、フリーソフトの DBMS としては **PostgreSQL**、**MySQL** などが代表的です。

データベースの中は、論理的には次の 3 つのレベルに分けることができます (図 6)。

- **内部レベル** — ディスク上でそれぞれのデータの格納形態を定めるレベル。物理レベルとも呼ばれる。
- **概念レベル** — すべてのプログラム/ユーザが使用するデータの形態を統合した形で定めるレベル。このレベルによってデータ独立性が実現されている。
- **外部レベル** — 個々のプログラム/ユーザの必要に合わせて、概念レベルのデータをマッピングしたもの。前節でビューと呼んでいたもの (「外部ビュー」と呼ぶ方が正確かも知れません)。

この分類は**データベースの 3 層モデル**として知られています。この考え方を使うと、DBMS の機能や役割りが分かりやすく整理できます。

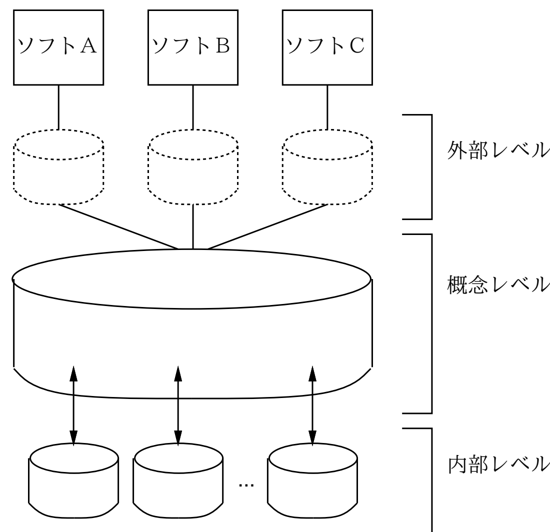

図 6: データベースの 3 層モデル

DBMS が概念レベルや外部レベルでデータをユーザに提示する枠組みのことを**データモデル**と呼びます。データベース上のデータに対して施せる (DBMS が提供する) 操作も、データモデルによっておおよそ定まってきます。歴史的にはさまざまなデータモデルが使われて来ましたが、今日最も広く使われているのは**関係モデル**(relational model) に基づいた**関係データベース**(Relational DataBase, RDB) です。RDB には理論的基盤がしっかりしていて汎用性が高いという特徴があります。以下ではこの RDB を中心に見て行きます。

なお、最近のクラウドサービス (Amazon Simple DB、Google App Engine など) では、データストアは RDB ではなくもっと簡略化された「キーと値の対 (key-value pair)」などを中心とするものになっています。また、トランザクション機能や (後述の)join 機能などにも制約があります。これは、サービスを多数のノードに分散配置するため、RDB のモデルをサポートするのは性能上困難だからとされています。将来もこのような簡略化されたものが使われ続けるかは現時点では分かりません。

もう 1 つの方向としては、RDB のようなきっちりとしたデータモデルを定めてしまうと「曖昧な/構造の明確でない」情報が格納できないという考えから、もっとゆるい構造のままでも情報を格納できるようにしようという動きも有ります。こちらは**半構造データ**(semi-structured data) と呼ばれます。

機構が通信機能経由で (クライアントプログラムとして) サーバに接続してデータにアクセスするという形を取ります (クライアントサーバシステム)。この場合、クライアントはサーバと同じマシンになくてもよいので、遠隔データベースアクセスも可能です (ただし、データ保護のためには通信路の安全性が問題となります)。

さらに DBMS によっては、複数のマシン上で稼働している DBMS 群が相互に通信し合うことで、各マシンに保管されているデータ群を全体として 1 つのデータベースのように扱い、検索や更新が行えるようにするものもあります。

2　関係モデルと RDB

2.1　関係モデルの概念

ではいよいよ、関係モデルと関係データベースについて具体的に見ていくことにしましょう。関係モデルでは、データベース中のあらゆるデータを「表」のようなものとして表します (ちなみに、「関係」というのは、表のようなデータを数学的に定式化する際に使われる言葉です)。関係モデルのデータは次の 3 つの概念から組み立てられます。

- **関係** (relation) — 1 つの「表」のこと。
- **属性** (attribute) — 表の各欄のこと。
- **組** (tuple) — 表中の 1 つの (値が横にならんだ) 行のこと。

だとえば、受注伝票のようなものを扱う簡単な関係データモデルを図 7 に示しました。

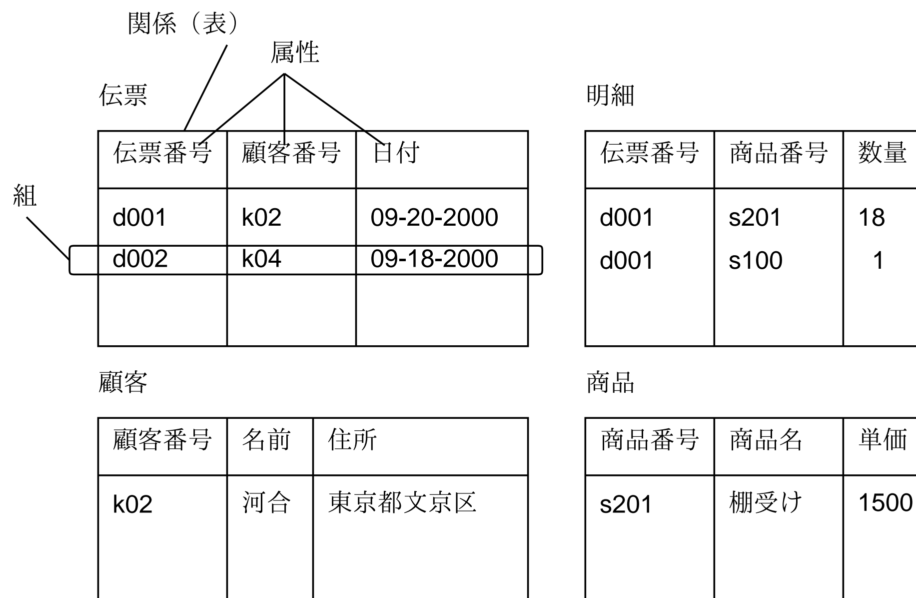

図 7: 関係データベースの例

このように、「表」というのは日常多くの場面で接するものなので、関係データモデルはその点でなじみやすく理解しやすいという長所があります。またその一方で、関係データモデルには表どうしの「演算」が簡潔に定義でき、それを用いて多様なデータ処理/検索が自然に行なえる、という特徴も持っています。

他方、RDB の弱点には、理論的には関係演算により簡潔に記述できる処理でも、実際に実行させようとした場合に多量のデータ操作に対応するため、他のモデルに比べて性能が低い、というものが

みです。RDB のもう一つの弱点は、データとして文字列、数値など基本的なデータ型しか扱わない点です。これに対処するものとしてはオブジェクト指向機能を前提とした DBMS がありますし、RDB にそのような機能を追加する動きもあります。ただ、今日の一般的なデータ処理では RDB が十分有効に使われている、といえるでしょう。

ところで、関係モデルの「表」は、普通の表とは次の点が違っています。

1. 1 つの表の中に全く同じ内容の組が 2 つ以上存在することはない。

2. 表の中の組の順番というのは意味を持たない。

これらは、関係を組の集合として位置付けたためそうなっているのであり、あくまでもモデルとしての考え方の問題です。実際に DBMS を使って処理を進める時には、RDB でも重複した組が現れることがありますし、並べる時に「どんな順に並べろ」と指定することも可能になっています。

2.2　データベースの設計

ところで、図 7 の表は普通の伝票にくらべてやけに細かく分かれているように見えます。たとえば、普通の伝票には顧客の名前や住所まで書いてあるものですが、図 7 では「伝票」の表には「顧客番号」だけ書かれていて、名前や住所は別の「顧客」の表を引かないと分かりません。これでは不便そうに見えますが…なぜこんなふうにしてあるのでしょうか?

その答はこういうことです。もしも「伝票」の表に顧客の名前や住所が書いてあったとすると、たとえばある顧客が 100 枚伝票を切ったとすれば、それに対応して 100 箇所に同じ名前や住所を入れておかなければなりません (記憶領域の無駄)。そしてもしその顧客が引っ越したら、全部の住所を新しいのに更新しなければなりませんし (更新の手間)、万一どれかを更新し忘れると伝票によって顧客の住所が異なるという事態 (データの矛盾) が生じます。さらに困ったことに、ある顧客とたまたま長い間取り引きがなく、データベースからその顧客が切った伝票が全部消えてしまうと…その顧客の名前も住所も、それどころか顧客番号の情報も、全て忘れられてしまいます (情報の逸失)。

このように考えていくと、「伝票」の表に顧客番号が入っている以上、その顧客番号から決まるような情報 (つまり名前や住所) はその表に入れるべきでなく、別の表に分けるのが正しいことが分かります。このような、属性どうしの論理的な依存関係に基づいてそれぞれの関係に含めるべき/べきでない属性を決めて行くことを**関係の正規化**と呼びます。

上の議論をもう少し厳密に整理してみましょう。そのためにはまず、**キー**について定義しておきます。

- ある関係の属性の集合で、その関係中の組を一意的に指定できるような最小のものを**候補キー** (candidate key) と呼ぶ。候補キーのうちからとくに 1 つを選んで**主キー** (primary key)、それ以外の候補キーを代替キー (alternate key) と呼ぶこともある。

多くの場合、候補キーは 1 種類しかなく、従ってこれが主キーとなります。たとえば関係「伝票」では、伝票番号が主キーでした。また、「明細」では伝票番号と商品番号を合わせたものが主キーでした (キーは属性の集合ですからこれでよいわけです。また、この 2 つを合わせることではじめて明細中の各項目が一意に指定できることも明らかです)。

そして、先に述べた「望ましい分け方」を表す指針の 1 つである、**第 3 正規形** (3rd normal form) は次のように定義されます。

- ある関係の主キー以外のすべての属性が主キーにのみ依存して決まる場合に、その関係は第 3 正規形である、という。

キーではない顧客番号に依存して決まるので、上の条件を満たさないことになります。第 3、というところから分かるように、データベースの理論ではほかにもいくつかの正規形が定義されていますが、第 3 正規形がとりあえず覚えておくべき設計指針だと言えます。ただし、このような指針は常に絶対というわけではなく、サイトの都合によってはあえて正規形でない関係を用いることもあるかも知れません。

まとめると、データベースを使う時には、その上で行なう操作、格納の必要がある情報の範囲、正規化、などを考慮して、どのような関係を作り、それぞれにどのような属性を持たせるかを決定する必要があるわけです。この作業を**データベース設計**といいます。後から関係 (表) を増やすくらいならまだしも、一度決めた構造を変更するとなると大量のデータを変換する必要があり、とても大変ですから、最初に十分将来を予見してデータベースを設計することが設計者の腕の見せどころなわけです。

演習 3〜4 名ずつのグループで、「自分たちの行きつけの飲食店とメニューの情報を格納・検索する RDB」を設計するものとします。必須の要件は次の通りです。

a. 地域を指定して、その地域の店の情報を調べられる。
b. 料理/ドリンク名を指定して、それを出している店を調べられる。
c. 「おいしいメニュー」「割安なメニュー」などが調べられる。

組 (tuple) に相当する細長い紙を提供するので、そこに予め設計した属性情報を記入してリレーション単位で並べてみてください (各自が自分のお勧めの店を数店以上、各店についてメニューを数品以上データ提供すること)。その上で、設計したデータベースでさまざまな検索を行えることを確認してください。記録が必要なら紙の束を写真に撮っておくこと。

3 データベースの実際

3.1 PostgreSQL とデータ操作言語 SQL

過去においてはデータベースを扱うには必ずプログラムを書かなければならず、さらに DBMS ごとにプログラムが呼び出す機能名などもバラバラだったのですが、今日の関係データベースでは SQL と呼ばれる**データ操作言語**(Data Manupilation Langauge, DML) が標準化されており、プログラムを書かずに直接 (または適当なインタフェースプログラムから) 処理内容を指定して操作できますし、指定方法もどの DBMS でもほぼ同じになっています。[2]

では実際に SQL を使ってデータベースを作成し、データを投入してみましょう。以下では GSSM のシステム上で稼働しているフリーの DBMS である PostgreSQL を用いて説明しますが、SQL 言語の部分については他の DBMS でも同じです (SQL に規定されていない表示機能などの部分は、DBMS ごとに違いがあります)。PostgreSQL では、Unix 上で各ユーザが自分のデータベースを作成し操作でき、後でそれを他人に公開して共有操作することもできます。とりあえず、自分用に小さいデータベースを用意してその上でいろいろ試してみましょう。[3]

まず、自分用のデータベースを作成するにはコマンド `createdb` を使用します。

```
% createdb
CREATE DATABASE
%
```

[2] プログラムから RDB を操作する場合も、SQL の命令をプログラムから DBMS に送る形で処理内容を指定します。

[3] GSSM で PostgreSQL を使用する作業は `smb` で実行する必要があります。「rlogin smb」で遠隔ログインするか、または `smb` の窓を開いてその中で実行してください。

```
% psql
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

kuno=>
```

ここで最後の「kuno=>」というところが psql のプロンプトで、ここに接続されているデータベース名が表示されます。つまり、createuser は特に指定しなければその人のユーザ名と同じ名前のデータベースを作成し、psql は特に指定しなければその人のユーザ名と同じ名前のデータベースに接続します。[4]

psql に打ち込むコマンドは、「\」で始まるものとそうでないものがあります。「\」で始まるものは psql 固有のコマンドであり、たとえば「\q[RET]」で psql を終了することができます (その他は「\?」でヘルプを実行して見てください)。「\」で始まらないものはすべて SQL の文であり、これによってデータベースを様々に操作できます。

3.2 SQL による関係の定義

ここまでで psql が起動され、SQL コマンドを投入できる状態になりました。次に、SQL で関係を作成するには次の構文を使います。

```
create table 関係名 ( 属性1 型1, 属性2 型2, ... ) ;
```

最後の「;」は SQL の構文の一部ではないのですが、多くの SQL 処理形は「;」「/」など特定の文字が来たときに実際の操作を開始するようになっています (ですから、その前に長いコマンドを何行にも分けて入れることができるわけです)。

さっそく簡単な関係「年齢表」を作ってみましょう。

```
kuno=> create table 年齢表 (名前 char(8), 年齢 int);
CREATE
kuno=>
```

このように、型として文字列を指定する場合はその長さ (文字数[5]) を指定しないといけません。

では次に作成した関係にデータを挿入してみましょう。それには **insert into 命令**を使い、関係名と、そこに入れる組の値を values(...) の中に列挙したもの (属性の並び順は関係を定義したときの順) とを指定します。[6]

---

[4] 名前を指定する場合は次のようにコマンド **createdb**、**psql**、**dropdb** を使用してください。

```
createdb データベース名 — データベースを作成する
qsql -d データベース名 — データベースに接続し SQL を使用開始
dropdb データベース名 — データベースを消去する
```

練習程度であればデータベースを複数持つ必要はないでしょうから、データベース名は省略して自分のユーザ名と同じ名前のデータベース 1 つを使えば済むでしょう (もちろん、1 つのデータベース内に関係はいくつでも入れられます)。

[5] 幸いなことにバイト数ではなく文字数です。

[6] 以下で出て来る例題中の名前やデータはすべて架空のものであり、実在の人物や組織と無関係です。

```
INSERT .....
kuno=> insert into 年齢表 values('大木', 25);
INSERT .....
kuno=>
```

無事にデータが入ったかどうか見たければ、次のようにします (select については後で詳しく説明します)。[7]

```
kuno=> select * from 年齢表;
   名前    | 年齢
----------+------
 久野      |   20
 大木      |   25
 立本      |   30
 尾崎      |   18
 ...
```

create table や copy などのコマンドをいちいち手で打ち込む代りにファイルに入れておいて次の指定により読み込ませて実行させることもできます。

```
kuno=> \i ファイル名
```

これも psql 固有の機能です。psql を終りにする方法ももちろん psql 固有の機能です。

```
kuno=> \q
```

ついでに、不要になった関係を削除する方法も説明しておきます。それは SQL の drop 命令を次のように使います。

```
kuno=> drop table 年齢表 ;
```

3.3 関係データベースと問い合わせ

データベースにおいて重要な機能の 1 つに、自分が知りたい情報を指定してそれをデータベース中から取り出して来ることが挙げられます。これを**問い合わせ** (query) と呼びます。

では、関係データベースの場合に「自分が欲しい情報」というのはどうやったら指定できるかを考えてみましょう。基本的に、関係データベースの中の各関係はあくまでも「組の集合」だということに注意してください。まず、一番最初に思いつくのは多分次のものでしょう。

1. 関係の中から、ある条件を満たす組だけを取り出す (**選択** — selection)。

検索というからには「ある条件のものを探す」というのは自然ですね。ところで、ある関係が属性をたくさん含んでいる場合、当面いらない属性は捨てて表示したいこともあるはずです。

[7] 1 つずつ insert into 命令で挿入していたのでは大変ですからファイルから読み込む方法もあります。そのためには、各欄が「,」字で区切られた次のようなファイル (CSV 形式) を用意します。

```
立本,30
尾崎,18
倉橋,33
西尾,22
```

その上で次のコマンドでデータを取り込めます。

```
kuno=> \copy 年齢表 from ファイル名 using delimiters ','
```

この命令は「\」で始まることからも分かるように SQL ではなく psql 固有のものであり、従って「;」は不要です (その代り 1 行で書く必要があります)。

選択と射影はそれぞれ、表の横の列、縦の列をいくつかずつ取り出す操作だと考えればよいでしょう。

ところで、図 7 のデータベースでは、関係「伝票」には顧客番号しか記録がなく、その氏名は関係「顧客」から同じ顧客番号の組を持ってくると始めて分かるようになっていました。そこで次の操作がとても重要になります。

3. 関係 A の属性 X と関係 B の属性 Y を比べて、互いに値が同じものだけをそれぞれ取り出してくっつけ、幅の広い表にする (**結合** — join)。[8]

これに加えて普通の集合の演算である**和** (union)、**差** (difference)、**積** (product) を使うと、普通データベースに我々が問い合わせたいと思うような事柄は大体指定できます。このような演算体系を**関係代数** (relational algebra) と呼びます。

関係データベース上での問い合わせを定式化するもう 1 つの方法として、**関係論理** (relational calculus) があります。これも、問い合わせの結果がまた表 (関係) の形をしている、という点では関係代数と同じですが、記法としてはだいぶ異なっていて、「どの表とどの表と... から組を持ってきて、この属性とこの属性と... の集まりからなる表を作れ、ただしそれぞれの属性は〜の条件を満たすようなものであること」という書き方をします。つまり、論理式を使って条件を指定するから「関係論理」と呼ぶわけです。実は SQL は関係論理に基づく問い合わせを基本にしています。

もう少し具体的に考えてみましょう。たとえば、関係代数の演算の中に「特定の属性だけを取り出す」射影という演算がありました。関係論理でこれを行いたければ、SQL の記法を借りると次のように書きます。

```
select 属性1, 属性2, ...  from 関係 ;
```

つまり、取り出したい属性だけを指定した **select 文**を使えば、結果として射影が行えるのです。なお、関係が持っているすべての属性を指定した場合には、元の関係と同じものが得られますが、属性名をいちいち書くのは面倒ですから、代わりに「*」と書けば済むようになっています (前節末で関係の内容を表示させた時には、ちょうどこの機能を使っていたわけです)。

一方、「特定の組だけを取り出す」選択演算は

```
select * from 関係 where 条件 ;
```

のようにして、選択したい条件を指定します。条件としては、たとえば「この属性の値がいくつ以上のもの」などと指定するわけです。

では、結合演算はどうでしょう? 実はこれは複数の関係を同時に指定し、条件として 2 つの関係の属性が等しいことを指定すればよいのです。[9]

```
select 関係1.属性1,..  関係2.属性N from 関係1, 関係2
       where 関係1.属性x = 関係2.属性y ;
```

残りの集合演算についても、**where 句**で適切な条件を指定することで実現できます (たとえば、和集合はそれぞれの集合に対応する `where` 条件の `or` を取ればできます)。実は、関係論理と関係代数の問い合わせ記述能力は互いに等しい (つまり、一方で書ける問い合わせは他方でも書ける) ことがわかっています。

[8]厳密には大小関係などに基づく結合も定義できるが、等しいことに基づく結合が最も多く使われる。

[9]最近の SQL では join を指定する特別の記法もありますが、それより「等しい」ことを指定する方が分かりやすいと思いこちらを説明しています。

さて、以下ではSQLの実例を使って実際に問い合わせをしてみましょう。例題としては図7の構造のデータベースを用いて、適当なデータを生成するためのSQL命令群を図8のように用意しました (beginとcommitが何を意味するかは後の方で説明します)。この内容をファイルに (文字コードEUCで) 格納しておいて前に説明した「\i ファイル名」の機能を使って読み込めば準備完了です。

ではまず単純な問い合わせから試してみましょう。

```
kuno=> select 顧客番号, 名前, 住所 from 顧客;
 顧客番号 |   名前   |       住所
----------+----------+------------------
 k02      | 河合     | 東京都文京区
 k03      | 久野     | 東京都目黒区
 k04      | 大木     | 埼玉県和光市
 k01      | 牧本     | 神奈川県川崎市
(4 rows)
```

関係「顧客」にある属性はこの3つで全部なので、先に述べたように属性名を全部列挙する代わりに「*」と指定しても構いません。特定の属性だけ取り出す (つまり射影) 場合は、取り出す属性名を列挙することになります。複数の関係に同じ属性名があるなどして、どの属性かが一意に決まらないような場合には、「関係名.属性名」のように前に関係名をつけることで、どの関係の属性かを明示してください。

次に、where句で条件を指定してみましょう。条件としては次のようなものが書けます。[10]

```
式　比較演算子　式
条件　and　条件
条件　or　条件
式 IN (内側検索)
式 NOT IN (内側検索)
```

「式」は属性名、定数、およびそれらの四則演算 (+、-、*、/) と丸かっこによる組み合わせです (文字列定数は' で囲みます)。比較演算子は=、<>、>、>=、<、<=などです。条件の結合順序は () で指定します。[11]簡単な例を示しましょう。

```
kuno=> select * from 商品 where (単価>2000) and (単価<30000);
 商品番号 |      商品名        | 単価
----------+--------------------+-------
 s200     | スタンド           |  3500
 s203     | サイドデスク       | 27000
(2 rows)
```

次にいよいよ結合を使ってみましょう。この場合は当然fromで複数の関係を指定することになります。

```
kuno=> select 伝票番号, 名前, 日付 from 伝票, 顧客
kuno->   where (伝票.顧客番号 = 顧客.顧客番号) and (伝票.日付 > '10-1-2000');
 伝票番号 |   名前   |    日付
----------+----------+------------
 d008     | 牧本     | 2000-10-10
 d006     | 久野     | 2000-10-02
(2 rows)
```

[10]もっと込み入ったものも書けますがここでは省略しています。

[11]最後の2つは…「内側検索」は通常のselect文で、ただし属性を1つだけ指定します。そうすると、その属性値の集合が得られるので、式がその集合に含まれているかいないかを条件として指定するわけです。

```
insert into 伝票 values('d002', 'k04', '09-18-2000');
insert into 伝票 values('d004', 'k02', '10-01-2000');
insert into 伝票 values('d005', 'k03', '10-01-2000');
insert into 伝票 values('d006', 'k03', '10-02-2000');
insert into 伝票 values('d008', 'k01', '10-10-2000');
create table 明細 (伝票番号 char(4), 商品番号 char(4), 数量 int);
insert into 明細 values('d001', 's201', 18);
insert into 明細 values('d001', 's100', 1);
insert into 明細 values('d001', 's200', 3);
insert into 明細 values('d002', 's203', 1);
insert into 明細 values('d002', 's201', 4);
insert into 明細 values('d004', 's100', 1);
insert into 明細 values('d004', 's201', 5);
insert into 明細 values('d004', 's200', 1);
insert into 明細 values('d005', 's201', 3);
insert into 明細 values('d006', 's100', 2);
insert into 明細 values('d006', 's203', 18);
insert into 明細 values('d006', 's200', 6);
create table 顧客 (顧客番号 char(4), 名前 char(8), 住所 char(16));
insert into 顧客 values('k02', '河合', '東京都文京区');
insert into 顧客 values('k03', '久野', '東京都目黒区');
insert into 顧客 values('k04', '大木', '埼玉県和光市');
insert into 顧客 values('k01', '牧本', '神奈川県川崎市');
create table 商品 (商品番号 char(4), 商品名 char(16), 単価 int);
insert into 商品 values('s201', '棚受け', 1500);
insert into 商品 values('s200', 'スタンド', 3500);
insert into 商品 values('s202', 'ブックエンド', 800);
insert into 商品 values('s203', 'サイドデスク', 27000);
insert into 商品 values('s100', 'ワークデスク', 37000);
commit;
```

図 8: 練習用のデータ

以上の操作をまとめたようすを、図 9 に示しました。

このようにして、関係「伝票」には顧客名は入っていないにも関わらずちゃんと入っているかのような表ができるわけです。ところで、組の順番は前にも述べたように「でたらめ」ですが、見る人にとってはそれは不便です。このため、必要なら「どの項目の昇順/降順で」と指定することで好みの順にならべて表示させられます。そのためには次のような **order by** 句を一番最後に追加してください。

```
... order by 式 asc   ←その式の「昇順」に並べる
... order by 式 desc  ←その式の「降順」に並べる
```

演習 psql を立ち上げ、図 8 のファイルを「\i」で読み込ませてデータベースを用意します (「\z」で関係一覧を表示し確認しなさい)。続いて、これらの関係群に対してここまでに載っている検索例を順次実行してみなさい。納得したら、次の検索課題から 1 つ以上 (できれば全部) やってみなさい。

a. 商品の中で、単価が 10000 円未満のの商品名を打ち出す。できれば、そのような商品で実際に売れたものだけが表示できるとなおよいでしょう。

b. 各明細項目について、伝票番号、商品番号、数量、および金額 (単価×数量) を打ち出す。できれば、商品番号の代わりに商品名が表示できるとなおよいでしょう。

c. 商品のうち、いちどに 4 個以上売れたものについて、商品名と売れた個数と顧客番号を打ち出す。できれば、顧客番号の代わりに顧客名が表示できるとなおよいでしょう。

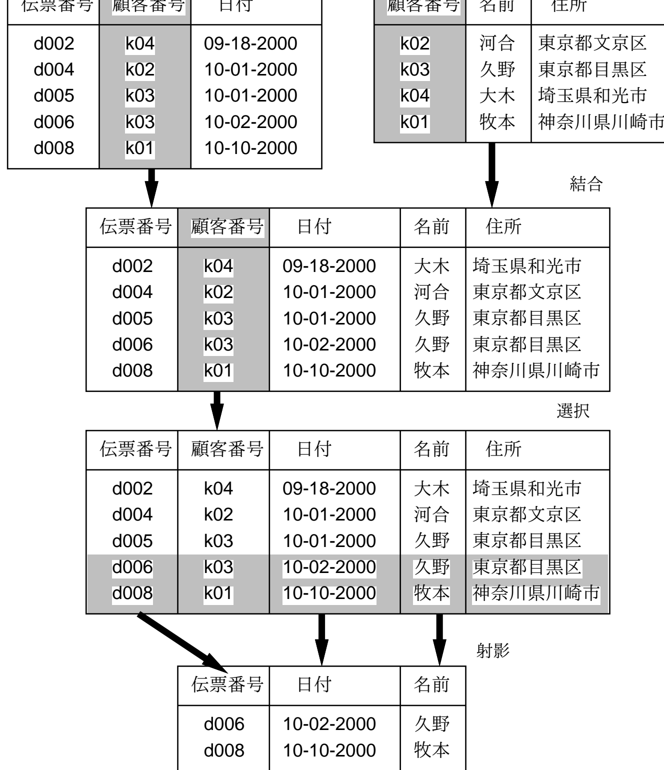

図 9: データベースによる一連の関係操作

3.5 集計関数

さて、ここまで来るとかなり複雑な問い合わせが書けるようになりましたが、個別の値を求めるだけではなく、「平均」や「合計」などの集計も行いたいですね? 実は、`select` の次にくる並びには属性以外に一般の式も書くことができ、しかもこの場所に限り、式の中に次のような**集計関数**(aggregate function) と呼ばれるものが使えます。

- `count(*)` — データの件数
- `sum(式)` — 式の値の合計
- `avg(式)` — 式の値の平均
- `max(式)` — 式の値の最大値
- `min(式)` — 式の値の最小値

これを使った例を挙げておきましょう。

```
kuno=> select count(*), avg(単価), max(単価)-min(単価) from 商品;
 count |        avg         | ?column?
-------+--------------------+----------
     5 | 13960.0000000000 |    36200
(1 row)
```

上では複数の集計結果を並べていましたが、1 つの結果だけを指定すればそれは 1 つの値ですから、SQL における 1 つの式として使うことができます。[12]

[12] つまり、先に「副問い合わせ」が集合である場合を扱いましたが、副問い合わせの結果が 1 つの値である場合もあるわけです。

計とか「顧客ごとの」合計、といったものが欲しいのが普通です。そこで、where 句が終ったあとに「group by 属性名,...」という指定をしておくと、その属性値 (の組) が同じものどうしをグループとしてまとめた後、それぞれについて統計関数を使ってくれます。

```
kuno=> select 伝票番号, count(*), sum(数量) from 明細 group by 伝票番号;
 伝票番号 | count | sum
----------+-------+-----
 d001     |     3 |  22
 d002     |     2 |   5
 d004     |     3 |   7
 d005     |     1 |   3
 d006     |     3 |  26
(5 rows)
```

このように、関係論理に統計関数と group by を組み合わせたことで、かなり複雑なデータ処理まで SQL だけで記述できてしまうのです。

3.6 ビューの定義

ところで、上の例では関係の正規化のため「明細」に商品の値段が入っていなくて、とても不便です。また、「伝票」にも合計金額が入っていて欲しいですね。このような情報は select を使って毎回計算することができますが (だからこそ関係には格納してないわけです)、よく使う場合にはそれを含んだ新しい**ビュー**を定義しておくと便利です。ビューは次のコマンドで定義します。

```
create view ビュー名 as 問い合わせ指定 ;
```

これから分かるように、ビューとは実は select コマンドに名前をつけたものだと考えることができます。では実際に、金額の入った明細を作ってみましょう。

```
kuno=> create view 詳細明細 as
kuno->   select 伝票番号, 明細.商品番号, 単価, 数量, 数量*単価 as 合計価格
kuno->     from 明細, 商品 where 明細.商品番号 = 商品.商品番号 ;
CREATE
kuno=> select * from 詳細明細;
 伝票番号 | 商品番号 | 単価  | 数量 | 合計価格
----------+----------+-------+------+----------
 d001     | s100     | 37000 |    1 |    37000
 d004     | s100     | 37000 |    1 |    37000
 d006     | s100     | 37000 |    2 |    74000
 d001     | s200     |  3500 |    3 |    10500
 d004     | s200     |  3500 |    1 |     3500
 d006     | s200     |  3500 |    6 |    21000
 d001     | s201     |  1500 |   18 |    27000
 d002     | s201     |  1500 |    4 |     6000
 d004     | s201     |  1500 |    5 |     7500
 d005     | s201     |  1500 |    3 |     4500
 d002     | s203     | 27000 |    1 |    27000
 d006     | s203     | 27000 |   18 |   486000
(12 rows)
```

つけるのに使います (こうしないとその欄を名前で指定できないため)。このように、ビューも見ためは普通の関係と変わりがない、という点は、関係モデルの強力な特徴の 1 つです。[13][14]

ビューは、このように情報を組み合わせて参照するのに使うだけでなく、ユーザやプログラムごとに専用のデータの「見えかた」を提供するのにも使われることは前に述べた通りです。このような機能は、たとえば次の場合に活用できます。

- ユーザによってはデータの一部を見えないようにしたい。
- 元データの属性が増えてもプログラムの変更しないようにしたい。

このように、データを仮想化することで汎用性とデータ独立性を得られることが、データベースを利用することの大きな利点です。

演習 上と同じデータベースについて、集計関数を使った次の検索課題から 1 つ以上 (できれば全部) やってみなさい。

a. 商品の中で、単価が平均商品単価を超えるものの一覧表を求める。できれば、平均商品単価よりいくら高いかも併せて示せるとなおよい。
b. 顧客ごとの注文金額合計を示した表を求める。できれば、それぞれの顧客の注文件数も示せるとなおよい。
c. 部品ごとの受注金額合計の表。できれば、部品ごとの注文した顧客数も示せるとなおよい。

3.7 データの更新と削除

前節まででさまざまなデータの「取り出し方」について学びましたが、まだデータの更新方法を説明していませんでした。ただし、関係に組を追加する方法は既に取り上げました。

```
insert into 関係名 values(データ,...) ;
```

一方、組を削除する時は「これこれの条件を満足する組を削除しろ」と言えばいいわけなので、次のように **delete 命令**を使います。

```
delete from 関係名 where 条件… ;
```

では特定のデータを更新 (書き換え) したい時は? 削除と挿入を組み合わせれば理論的には更新になるはずですが、使いにくいし効率も悪そうなので、次のような **update 命令**が用意されています。

```
update 関係名 set 属性1 = 式, 属性2 = 式, ...  from ...  where ...  ;
```

ここで from 以下は select と同様であり、省略も可能です。これを用いれば特定の属性だけを変更できます。たとえば「全部の年齢を 2 割増しにする」には次のように指定すればよいのです。

```
update 年齢表 set 年齢 = 年齢*1.2 ;
```

また、「1 回の取引で合計価格が 3 万を超えた商品について、単価を 2 割引く」という操作は次のようになります。

```
update 商品 set 単価 = 商品.単価*0.8 from 詳細明細
  where (詳細明細.商品番号 = 商品.商品番号) and (詳細明細.合計価格 > 30000);
```

この例はさっき作ったビュー「詳細明細」をうまく活用していることにも注目してください。

[13]ただし、現在の SQL では計算式に基づく欄の使用やビューに対する更新操作などに制約が設けられていて、必ずしも思ったようにできない場合があります。
[14]作ったビューが不要になった場合は「drop view ビュー名 ;」によって消去してください。

ここまでは関係を定義する時に各属性のデータ型しか指定して来ませんでしたが、最初に述べたように、データベースでは「このような性質が保たれること」という条件を指定しておくことで、プログラムが間違っても「悪い」データが入れられてしまうことを (あくまでもある程度ですが) 水際で阻止できるようになっています。代表的な条件としては次のようなものがあります。

- 一意的 (unique) — 「この属性は重複した値を持たない」という性質。
- 非空 (not null) — 「この属性は無効値を持たない」という性質。データは「欠落している」ことも現実によくあるので、データベースでは「無効値」を扱えるようになっていることが普通。属性によっては、これを許さないという指定もしたいわけです。
- 主キー (primary key) — 一意的かつ非空、つまりキーとして使えるという意味。
- 外部キー (foreign key) — 「他の関係の指定した属性値に現われる値のみが値として許される」という指定。たとえば関係「伝票」で属性「商品番号」として正しいものは、関係「商品」に「商品番号」として現われるものに限られるわけです。
- 条件式 — 「常識的に見て単価は百万円以下」など、普通の条件式を使ったチェックもできます。条件として他の関係のデータを参照することもできます。

ただし、すべての「おかしな」データを条件のみで排除できるわけではないので、データベースに「ごみが入って腐る」のを防ぐのは結構難しい問題ではあります。

3.9 データベースの共有制御

さて、ここまででではデータベースを一人でいじってきましたが、複数の人から共有できることがデータベースの本領です。そのために、**grant 命令**で自分のデータベースを他人にも使えるように保護設定を変更できます。

```
grant 権限,... on 関係名,... to ユーザ名
```

ここで権限は `select`(検索できる)、`insert`(組を挿入できる)、`delte`(組を削除できる)、`update`(更新できる)、`rule`(規則を設定できる)、`all`(すべてできる) の組み合わせで指定します。また、ユーザ名の代りに `public` と指定すると「誰でも」指定した権限を持つようになります。また、`grant` で指定した権限を取り除くには **revoke 命令**を使います。

```
revoke 権限,... on 関係名,... from ユーザ名
```

そのパラメタの意味は `grant` と同じです。

3.10 トランザクション

データベースのデータが共有されるようになったとして、それで何も考えずに複数の人 (やプログラム) によるデータ操作がうまく行くという訳には行きません。たとえば、B さんの口座に 1 万円入っていて、A さんから B さんへの 1000 円の送金処理と B さんの 2000 円のクレジット引き落とし処理が「同時に」実行されたとしましょう (図 10)。

1 万円から 3000 円を引き落としたはずなのに残高が 9000 円になってしまいました。このように、データを「同時に更新」することは正しくない結果をもたらす可能性があります。

また、別の事柄として、マシンの故障などに対する耐性の問題も挙げられます。たとえば、「A さんに 1000 円送金」した直後にマシンがダウンして処理が止まってしまったらどうなるでしょうか? A さんの残高は 1000 円増えているはずですが、B さんの残高が更新され損なって 1 万円のままだと、どこかでお金が足りなくなるはずです。

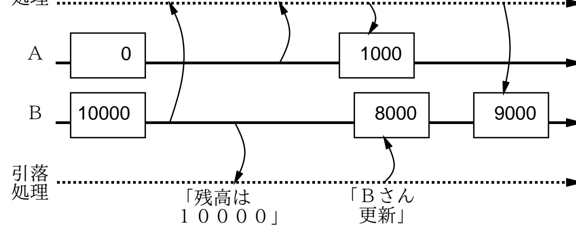

図 10: 競合するアクセスによる矛盾

これらの問題に対処するため、DBMS は**トランザクション**と呼ばれる機能を提供しています。トランザクションを扱う SQL 文は次の 3 つがあります。

- `begin` — トランザクションを開始する
- `commit` — トランザクションを正常完了する
- `abort` — トランザクションを中止する

つまり、上記の引き落としのようなまとまった処理をするときは

```
begin ;
データベース操作
...
commit ; ←またはどこかでうまく行かなければ abort
```

のような形でデータベースを利用することで、その範囲が 1 つのトランザクションとして処理されます。そうするとどういう「いいこと」があるのでしょうか? トランザクションは一般に次のような性質を持ちます (これらの頭文字を取って **ACID 属性**と呼びます)。

- Atomicity (原子性) — トランザクションは「全体として起こる」(commit が成功した場合) か「何も起こらない」(前記以外の場合) かどちらかであり、「途中まで起こる」ということはない。
- Consistency (一貫性) — トランザクションではデータベースを操作している中間的な状態は外から見えず、開始前の (一貫性のある) 状態から、完了後の (一貫性のある) 状態に一瞬で遷移する。たとえば、「B さんの残高を読み、A さんに送金し、B さんの残高を更新する」場合、最後の残高更新が完了するまでその状態は見られることがない。
- Isolation (独立性) — 複数のトランザクション T1、T2 が同時並行的に実行されているとしても、その結果は T1 と T2 が独立に実行された場合 (つまりまず T1 が実行、次に T2 が実行か、あるいはその逆か) と同等の結果をもたらす。
- Durability (耐久性) — commit が成功したらそれ以後は何があっても、そのトランザクションの結果がデータベースに反映されていることを保証する。

これらの性質を実現するため、DBMS はトランザクション内の操作すべてについて「こういう操作をしている」という情報を**安定記憶** (stable storage、システムダウンしてもデータが失われない場所 — だいたいはディスク装置) に書きながら、他のトランザクションとの競合をチェックしていきます (図 11)。そして `commit` した瞬間に「commit した」という記録を書いて、それからデータ本体を本来あるべき形に更新します。途中でシステムダウンした場合は、安定記憶の内容に基づいて操作を再度やり直せばあるべき状態になります。また、競合や abort 文により中止した場合や、システムダウン時に「commit した」がまだ書かれていなかった場合は、単にそこまでの操作を捨てれば何もなかったことになります。

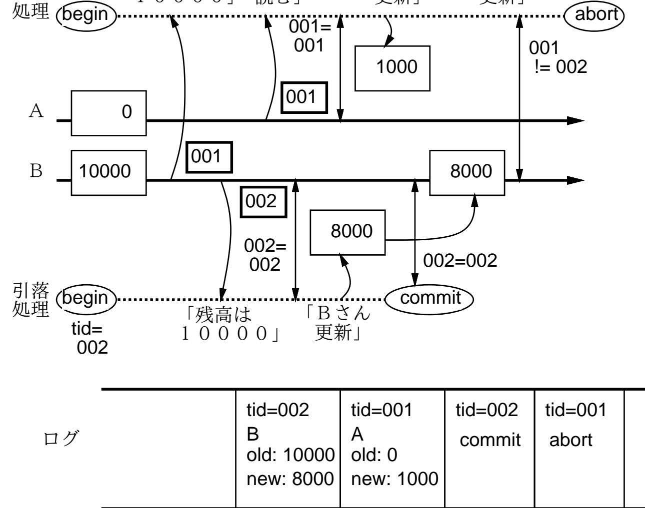

図 11: 楽観的並行制御によるトランザクションの実装

PostgreSQL では個々の SQL 文の操作に対しても 1 つずつトランザクションを作成し、完了したら commit しています。ただし、トランザクションにはオーバヘッドがあるため、実用的にはある程度まとまった操作をすべて begin-commit で囲んでトランザクションにすることが (一貫性のためだけでなく性能のためにも) 望ましいでしょう。図 8 のコマンド群も確かにそのようにしてありました。

演習 2 人以上で協力して、誰かのデータベースに 3.2 節で示したような「年齢表」テーブルを (人数十名以上で) 追加し、このテーブルを他の人に grant で読み書きアクセス可能にすることで、他人から内容が検索できるようにしてみなさい。[15]その上で、次の課題から 1 つ以上 (できれば全部) やってみなさい。

a. 複数人が同時に特定の人の年齢を 1 増やすことを実行し、結果として適切な数だけ増えているかどうか観察する。さらに、「全部の人の年齢を 1 増やす」でも同様であるか観察できるとなおよい。
b. 複数人が begin を実行し、特定の人の年齢を 1 増やし、最後に commit を実行するものとする。この時どのようなことが起きるか観察する。さらに、同じ人でなく別の人を操作した場合はどうかも観察できるとなおよい。
c. 複数人が begin を実行し、特定の人の年齢を別の特定の人の年齢プラス 1 に設定し、最後に commit を実行するものとする。この時どのようなことが起きるか観察する。さらに、参照する人のみが同じ場合、設定する人のみが同じ場合について観察できるとなおよい。

[15]他人の DB をアクセスする場合は「`psql -d` データベース名」を使うのでしたね。