INTRODUCTION TO COMPUTING
TOPIC 10: DATABASES

PAUL L. BAILEY

1. Terminology

A *database* is a (possible huge) set of information, organized in a fashion to facilitate retrieval of specific pieces of the information. The software which stores and retrieves information from the database is known as a *database management system* (DBMS). Familiar example of DBMS's may include Oracle or Microsoft SQL Server, but there are many, many others.

The database is typically divided into groups of information about specific *objects*. Each object is described by its *attributes*.

For example, consider a simple address book database. The main type of object in this database is a person. The attributes of a person we wish to store may include his/her

- Name
- Sex
- Date of Birth
- Address
- City
- State
- Zip
- Phone
- Email

The set of information regarding a specific object is called a *record*. The record is subdivided into *fields*, where each field contains the value of a specific attribute of the object.

Continuing the example, one record of our database may contain

- John Smith
- Male
- 12/25/1980
- 123 Nowhereland Ave
- Magnolia
- Arkansas
- 12345
- 870-234-5678
- jzsmith@gmail.com

Date: April 24, 2008.

In order to manage this information effectively, the system needs to know what type of information to expect in each field. This is known as the *field type*, or *data type*.

The most basic types are

- integer
- floating point number
- string (sequence of ASCII characters)
- blob (binary data of unspecified length)

This types refer to the physical manner in which the data is stored.

In our example, the data types corresponding to the given fields may be

- string (25 characters)
- string (6 characters)
- integer (date stored as number of days since a base date)
- string (40 characters)
- string (25 characters)
- string (20 characters)
- integer (store zip as an integer)
- string (12 characters)
- string (25 characters)

Some database management systems include additional types, built from the basic types, which in addition to saying how the data is physically stored, also say how it is to be interpreted. In fact, the DBMS may allow the database designer to define new types, by specifying the nature of this interpretation. For example, the DBMS may include separate types like

- Name (always stored as "LAST,FIRST" in up to 25 characters)
- Date (always stored as an integer containing the number of days since a constant base date)
- Time (always stored as an integer containing the number of seconds since midnight)

The interpretation of the data gives the system knowledge of how to handle the data. Storing the name as above allows the system to sort by last name. Storing date as above allows the system to do automatic calculations, such as finding the age of a person.

Each object of the same type stores the same set of attributes, and each attribute has a name and a data type. The list of field names and types is typically known as the *record layout* for the object type. In our example, the record layout is

```
Name            string (25 characters)
Sex             string (6 characters)
Date of Birth   date
Address         string (40 characters)
City            string (25 characters)
State           string (20 characters)
Zip             integer
Phone           string (12 characters)
Email           string (25 characters)
```

2. Physical Implementation of Records

There are two basic historical styles of storing information; fixed-length records, and delimited records.

2.1. **Fixed Length Records.** With fixed length records, the maximum field length is allocated for each field, and the record length is the sum of these field lengths. If the entire field length is not used, the field is padded with spaces. In this way, every record has the same length.

For example, consider a DMV database which stores licensed drivers. Its main type of object is a driver (person), and stores information which goes on the driver's license. Suppose our record layout is

```
Name        string (25 characters)
Sex         string (1 character)
Birthdate   string (10 characters)
Height      string (6 characters)
Weight      string (4 characters)
Hair        string (7 characters)
Eyes        string (7 characters)
```

The total record length is 60. We give some example records, with the end indicated with a vertical bar.

```
SMITH,JOHN               M11/25/19606'0"  175 BLACK  BROWN  |
RASKOLNIKOV,CYNTHIA      F01/01/19705'10" 135 BLONDE HAZEL  |
```

The system knows where fields begin and end by there length.

2.2. **Delimited Records.** It is also possible to store variable length fields. Each field is separated by a specific character, such as a tab, comma, or semicolon. This separating character is known as a *delimiter*.

The examples above, using semicolons as delimiters, look like this:

```
SMITH,JOHN;M;11/25/1960;6'0";175;BLACK;BROWN
RASKOLNIKOV,CYNTHIA;F;01/01/1970;5'10";135;BLONDE;HAZEL
```

Modern systems use complex mixtures of these techniques.

2.3. **String Manipulation Functions.** Programming languages typically supply functions which search and take apart strings, so that field can be extracted from records. Such functionally usually includes:

- `Find` finds a substring in a string
  Example: `Find("This is a test","is") = 3`, because the first occurrence of `"is"` in the string `"This is a test"` starts at the third character in the string.
- `Extract` pulls out the characters between to positions
  Example: `Extract("This is a test",3,6) = "is i"`
  Extract may be used to pull fields out of a fixed length record.
- `Piece` pulls out a delimited field
  Example: `Piece("This;is;a;test",";",4) = "test"`
  Piece may be used to pull fields out of a delimited record.

3. Indices

Consider our DMV database using fixed length records. Suppose we store all of the drivers in one file. The records occur consecutively in the file. The sequence number of the record is its *record number*. Thus, the fifth record in the file has record number 5.

We know the exact byte position of the file where the n^{th} records begins; it is in byte position

$$(\text{bytes per records}) \times (n-1).$$

Here, the first byte of the file is numbered 0.

Now the system can randomly access the n^{th} byte of a file; it does not have to read the first $n-1$ bytes to get to the n^{th} byte. Thus, the system can go straight to a given record, if we know the number of the record we wish.

However, how does the system know which record is the record for a given person?

The records of a database typically are stored in the order they are entered, and there could be millions of them. In order to find the record for (let's say) John Smith, we do not wish to sequentially search through millions of records looking for records which have a name of John Smith.

Index files facilitate finding the desired information in a database. An index on a particular field stores the values for that field, together with the number of the record it comes from. The entries in the index are sorted, which makes it easier to zero in on a specific value.

Each time a record is added, or an indexed field value is changed, the index is updated.

For example, suppose we indexed the name field in the above example. We look up SMITH,JOHN in the index, and find that this name occurs in record number 3000. Then we go directly to record 3000 to get the rest of the information.

4. B-Trees

A *B-tree* is an efficient way to store indices; its goal is to find a given piece of information with a minimal number of "disk hits".

The phrase B-tree is said to stand for "balanced tree"; this refers to the fact that the data is obtained by descending a sequence of branches, and that all of the ultimate information is stored at the same depth.

A file is divided into blocks of equal size. These blocks are allocated to store either name/data pairs or name/pointer pairs. Directory blocks store name/pointer pairs, and data blocks store name/data pairs.

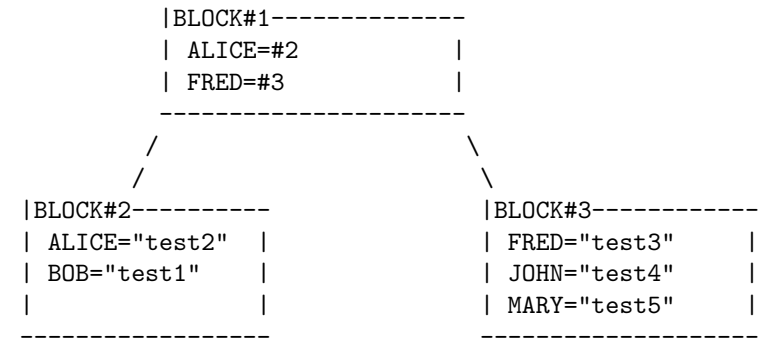
A name/data pair consists of the name of a variable, together with its value. A name/point pair consists of the name of a variable, together with the number of a block where the system goes to continue to look for the name/data pair.

We describe how a B-tree works by explaining how it grows. The "directory block", or root, is at the top of the tree. Initially, this contains name/data pairs. The system inserts new pairs, in order of the names, until the block is full. When the block is full, it is split.

Say, for example, a block can hold four pairs. We set BOB="test1", ALICE="test2", FRED="test3", and JOHN="test4". The block stores these entries, sorted by name, and now looks like this:

```
|BLOCK#1-----------
| ALICE="test2"   |
| BOB="test1"     |
| FRED="test3"    |
| JOHN=test4"     |
-------------------
```

The block is full. We set MARY="test5". The system splits the block, and creates a new pointer block, whose entries are name/pointer pairs which tell the system the block numbers of where to continue looking.



Now, to find JOHN, the system looks in the directory block, says that JOHN is greater than FRED, so JOHN must be in block #3. The system continues to add blocks to the tree structure, and splitting them when they get full. Each time a data block splits, a new pointer is added to the pointer block above it, until the pointer block needs to split, and so forth.

The amount of information stored at the data level is exponentially related to the number of levels between the data level and the top directory block.

5. MUMPS Globals

The ANSI standard programming language MUMPS is still common among hospital database applications. The MUMPS language has an embedded database scheme, implemented using global variables.

A *local variable* is stored in memory, is accessible by the single program which created it, and is temporary; when the program quits, the variable and its value are forgotten. A *global variable* is stored permanently on the disk, and is accessible to all users in a multi-user application. All of MUMPS's globals are stored in an enormous B-tree, which is a way of storing sorted data for efficient disk access.

Local variable start with a letter, and global variables start with a caret:

```
A=1, B="SMITH,JOHN", C("TEST")=23          local variables
^A=1, ^B="SMITH,JOHN", ^C("TEST")=23       global variables
```

MUMPS programmers create ad hoc databases using delimited records. By ad hoc, we mean that no database management system is used. The "file design" is kept as a separate text document, which is referred to and edited by programmers as they proceed. Filers which create indices are hand coded.

Records may be stored as follows. Let `^PAT` denote the patient database. The records are stored sequentially:

```
^PAT(1)="SMITH,JOHN;M;12/25/1960"
^PAT(2)="JONES,TOM;M;01/01/1950"
...
^PAT(1000)="THOMAS,CLARENCE;M;05/05/1940"
```

If `^NAM` is an index by name, it might contain

```
^NAM("THOMAS,CLARENCE")=1000
```

So we could look here to find that Clarence Thomas's record was number 1000.

6. ISAM Files

ISAM (Indexed Sequential Access Method) is a simple database management scheme for fixed length records.

The record is laid out as a sequence of fixed length fields. A huge file contains all of the fixed length records. Since all of the records have the same length, a record can be randomly accessed, as long as the record number is known.

Some of the fields are indicated as "indexed". Each indexed field supports its own index file, stored in a B-tree. When a new record is filed, the index files are built automatically by the DBMS.

To look up a record by an indexed field, the corresponding index is accessed to find the record number of the indexed record.

7. Relational Databases

A *relational database* combines the underlying ideas of a ISAM database, together with the mathematical concept of a relation, to organize and empower the ability to select a subset of the desired records.

7.1. **Relations.** All modern mathematics is built on the notion of sets and functions. The symbol $\mathbb{R}$ denotes the set of real numbers. Recall the mathematical concept of the Cartesian plane; it is the set of all ordered pairs of real numbers:

$$\mathbb{R} \times \mathbb{R} = \{(x, y) \mid x \text{ and } y \text{ are real numbers}\}.$$

A *relation* between real numbers can be described as a subset of $\mathbb{R} \times \mathbb{R}$. For example, consider the inequality $x \leq y$. One way to understand this statement would be to list all pairs (x, y) such that $x \leq y$. There are infinitely many real numbers, so we couldn't actually list the pairs; yet we can still consider the set

$$\{(x, y) \mid x \leq y\}.$$

In some way, this set *defines* what it means for x to be less than or equal to y.

Given any two sets A and B, we can form their *cartesian product* $A \times B$, which is the set of all ordered pairs whose first entry is from the set A and whose second entry is from the set B:

$$A \times B = \{(a, b) \mid a \text{ is in } A \text{ and } b \text{ is in } B\}.$$

Now a relation between members of the sets A and B can be described as a subset of $A \times B$.

For example, consider the case where A and B are sets of people, and the relation is "is the son of". We consider the set

$$\{(a, b) \mid a \text{ is the son of } b.$$

This set captures the idea behind the phrase "is the son of", by listing all of the pairs of people such that the first is the son of the second.

We can equally define the cartesian product of three sets $A \times B \times C$, whose entries are *ordered triples*:

$$A \times B \times C = \{(a, b, c) \mid a \text{ is in } A,\ b \text{ is in } B,\ c \text{ is in } C\}.$$

Again, consider the case where A,B, and C are sets of people. We could from the relation

$$\{(\text{father}, \text{mother}, \text{child})\},$$

listing all of the combinations of three people such that the first is the father of the third, and the second is the mother of the third. We call this set a relation./

In general, given n sets A_1, A_2, ..., A_n, we consider the set of all *ordered n-tuples* with first entry from A_1, second entry from A_2, etc., up to the n^{th} entry from A_n. This is the *cartesian product* of the sets:

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) \mid a_i \text{ is in the set } A_i.$$

A *relation* on these sets is a subset of $A_1 \times \cdots \times A_n$; it is a set of ordered tuples whose i^{th} entry is from the set A_i. The sets A_1, dots, A_n may be called the *attributes* of the relation.

Relations are sometimes classified as *one-to-one*, *one-to-many*, *many-to one*, or *many-to-many*. We give familiar examples of these:

(a) (husband,wife) is one-to-one
(b) (father,child) is one-to-many
(c) (child,mother) is many-to-one
(c) (brother,sister) is many-to-many

A *functional relation* is a relation in which the value of second entry is completely determined by the value of the first. These relations are one-to-one or many-to-one. For example, if we know who the child is, we should be able to determine the mother.

7.2. **Relational Databases.** In a relational database, the data is viewed as being organized into relations.

For example, let A be the set of all names, $B = \{M, F\}$, and C is the set of all dates. A relation on $A \times B \times C$ would be a collection of ordered tuples of the form (name,sex,birthdate). The relation describes what is stored; a particular tuple in the relation corresponds to a record in the database. The sets A, B, and C which constitute the relation correspond to the attributes, or fields, of the database.

Since it is common to list the tuples in rows and line up the attributes in columns, the relation itself is often referred to as a *table*. In fact, there are three common sets of words describing this aspect of a relational database:

Math Model	Relation	Tuples	Attributes
Programmer	File	Records	Fields
User	Table	Rows	Columns

A relational database consists of multiple tables. The tables are linked by the fact that the same attribute may exist in more than one table.

A *key* is a group of one or more attributes which uniquely identifies a row in a table. By storing a key in one table, that table may point to an entry in another table.

7.3. **Algebra of Tables.** *Algebra* regards the process of taking two things and using them to produce a third, as in adding or multiplying numbers.

There are various operations which can be performed on tables in a relational database, taking two tables an producing a third. These operations are set-theoretical in nature.

Two tables T_1 and T_2 are *union-compatible* if they have the same columns. The first three operations require that the tables T_1 and T_2 be union compatible.

(a) *Union*: The union of two tables T_1 and T_2 is formed by taking all the rows from either table and putting them into one table. Duplicate rows are discarded.

(b) *Intersection*: The intersection of table T_1 and table T_2 consists of the rows which are in both T_1 and T_2.

(c) *Difference*: The difference between table T_1 and table T_2 is a table whose rows are those of T_1 which are *not* in T_2.

We given an example.

```
T1 =
Name    Sex     Eyes
--------------------
John    M       Blue
Mary    F       Brown
Fred    M       Hazel
```

```
T2 =
Name    Sex     Eyes
--------------------
Tim     M       Brown
John    M       Blue
Sue     F       Blue
```

```
T1 union T2 =
Name    Sex     Eyes
--------------------
John    M       Blue
Mary    F       Brown
Fred    M       Hazel
Tim     M       Brown
Sue     F       Blue
```

```
T1 intersect T2 =
Name    Sex     Eyes
--------------------
John    M       Blue
```

```
T1 difference T2 =
Name    Sex     Eyes
--------------------
Mary    F       Blue
Fred    M       Hazel
```

Other tables are built by creating new tables with columns from both tables.

(d) *Product:* The product of T_1 and T_2 is the table which has all of the columns of T_1 and T_2, and whose rows are every combination of the rows of T_1 with the rows of T_2.

```
T1 =                                T2 =
Name    Sex     Eyes                Dept    Course
--------------------                --------------------
John    M       Blue                MATH    1023
Mary    F       Brown               ENGL    1113
Fred    M       Hazel

T1 product T2 =
Name    Sex     Eyes    Dept    Course
-------------------------------------
John    M       Blue    MATH    1023
John    M       Blue    ENGL    1113
Mary    F       Brown   MATH    1023
Mary    F       Brown   ENGL    1113
Fred    M       Hazel   MATH    1023
Fred    M       Hazel   ENGL    1113
```

We can build the tables we want by taking subtables of the product.

(e) *Projection*: chooses a subset of the columns

(f) *Selection*: chooses a subset of the rows

(g) *Join*: a combination of product, selection, and projection; typically, the joined table contains tuples such that the value of a column in table T_1 equals the value of a corresponding table in T_2.

More specifically, the *natural join* of T_1 and T_2 along a common column C has as columns all the columns of T_1 and of T_2, with C not repeated, and has as rows every combination of rows from T_1 and T_2 which have a common value in column C.

```
T1 =                            T2 =
Student     Course              Course      CourseName
--------------------------      --------------------------
Fred        MATH1023            MATH1023    College Algebra
Tom         ENGL1113            ENGL1113    Composition I
Mary        ENGL1113            HIST1013    History of Civ
Mary        MATH1023
Sue         PSYC2003

T1 join T2 at Course =
Student     Course      CourseName
---------------------------------
Fred        MATH1023    College Algebra
Tom         ENGL1113    Composition I
Mary        ENGL1113    Composition I
Mary        MATH1023    College Algebra
```

Department of Mathematics & Computer Science, Southern Arkansas University
E-mail address: plbailey@saumag.edu