Data Modeling

23 min. read · View original

Data modeling is the hardest and most important activity in the RDBMS world. If you get the data model wrong, your application might not do what users need, it might be unreliable, it might fill up the database with garbage. Why then do we start a SQL tutorial with the most challenging part of the job? Because you can't do queries, inserts, and updates until you've defined some tables. And defining tables is *data modeling*.

When data modeling, you are telling the RDBMS the following:

- what elements of the data you will store
- how large each element can be
- what kind of information each element can contain
- what elements may be left blank
- which elements are constrained to a fixed range
- whether and how various tables are to be linked

Three-Valued Logic

Programmers in most computer languages are familiar with Boolean logic. A variable may be either true or false. Pervading SQL, however, is the alien idea of *three-valued logic*. A column can be true, false, or NULL. When building the data model you must affirmatively decide whether a NULL value will be permitted for a column and, if so, what it means.

For example, consider a table for recording user-submitted comments to a Web site. The publisher has made the following stipulations:

- comments won't go live until approved by an editor
- the admin pages will present editors with all comments that are pending approval, i.e., have been submitted but neither approved nor disapproved by an editor already

Here's the data model:

```
create table user_submitted_comments (
        comment_id              integer primary
key,
        user_id                 not null
references users,
        submission_time         date default
sysdate not null,
        ip_address              varchar(50) not
null,
        content                 clob,
        approved_p              char(1)
check(approved_p in ('t','f'))
);
```

Implicit in this model is the assumption that `approved_p` can be NULL and that, if not explicitly set during the INSERT, that is what it will default to. What about the check constraint? It would seem to restrict `approved_p` to values of "t" or "f". NULL, however, is a special value and if we wanted to prevent `approved_p` from taking on NULL we'd have to add an explicit `not null` constraint.

How do NULLs work with queries? Let's fill `user_submitted_comments` with some sample data and see:

```
insert into user_submitted_comments
```

```
(comment_id, user_id, ip_address, content)
values
(1, 23069, '18.30.2.68', 'This article reminds me
of Hemingway');

Table created.

SQL> select first_names, last_name, content,
user_submitted_comments.approved_p
from user_submitted_comments, users
where user_submitted_comments.user_id =
users.user_id;

FIRST_NAMES  LAST_NAME      CONTENT
APPROVED_P
------------ -------------- --------------------
---------------- -----------
Philip       Greenspun      This article reminds
me of Hemingway
```

We've successfully JOINed the `user_submitted_comments` and `users` table to get both the comment content and the name of the user who submitted it. Notice that in the select list we had to explicitly request **`user_submitted_comments`**`.approved_p`. This is because the `users` table also has an `approved_p` column.

When we inserted the comment row we did not specify a value for the `approved_p` column. Thus we expect that the value would be NULL and in fact that's what it seems to be. Oracle's SQL*Plus application indicates a NULL value with white space.

For the administration page, we'll want to show *only* those comments where the `approved_p` column is NULL:

```
SQL> select first_names, last_name, content,
```

```
user_submitted_comments.approved_p
from user_submitted_comments, users
where user_submitted_comments.user_id =
users.user_id
and user_submitted_comments.approved_p = NULL;

no rows selected
```

"No rows selected"? That's odd. We know for a fact that we have one row in the comments table and that is `approved_p` column is set to NULL. How to debug the query? The first thing to do is simplify by removing the JOIN:

```
SQL> select * from user_submitted_comments where
approved_p = NULL;

no rows selected
```

What is happening here is that any expression involving NULL evaluates to NULL, including one that effectively looks like "NULL = NULL". The WHERE clause is looking for expressions that evaluate to true. What you need to use is the special test IS NULL:

```
SQL> select * from user_submitted_comments where
approved_p is NULL;

COMMENT_ID    USER_ID SUBMISSION_T IP_ADDRESS
---------- ---------- ------------ ----------
CONTENT                              APPROVED_P
------------------------------------ ------------
         1      23069 2000-05-27   18.30.2.68
This article reminds me of Hemingway
```

An adage among SQL programmers is that the only time you can use "= NULL" is in an UPDATE statement (to set a column's value to NULL). It never makes sense to use "= NULL" in a WHERE clause.

The bottom line is that as a data modeler you will have to decide which columns can be NULL and what that value will mean.

Back to the Mailing List

Let's return to the mailing list data model from the introduction:

```
create table mailing_list (
        email           varchar(100) not null
primary key,
        name            varchar(100)
);

create table phone_numbers (
        email           varchar(100) not null
references mailing_list,
        number_type     varchar(15) check
(number_type in ('work','home','cell','beeper')),
        phone_number    varchar(20) not null
);
```

This data model locks you into some realities:

- You will not be sending out any physical New Year's cards to folks on your mailing list; you don't have any way to store their addresses.
- You will not be sending out any electronic mail to folks who work at companies with elaborate Lotus Notes configurations; sometimes Lotus Notes results in email addresses that are longer than `100` characters.
- You are running the risk of filling the database with garbage since you have not constrained phone numbers in any way. American users could add or delete digits by mistake. International users could mistype country codes.
- You are running the risk of not being able to serve rich people because the `number_type` column may be too constrained. Suppose William H. Gates the Third wishes to record some extra phone numbers with types of "boat", "ranch", "island", and "private_jet". The `check (number_type in`

`('work','home','cell','beeper'))` statement prevents Mr. Gates from doing this.
- You run the risk of having records in the database for people whose name you don't know, since the `name` column of `mailing_list` is free to be NULL.
- Changing a user's email address won't be the simplest possible operation. You're using `email` as a key in two tables and therefore will have to update both tables. The `references mailing_list` keeps you from making the mistake of only updating `mailing_list` and leaving orphaned rows in `phone_numbers`. But if users changed their email addresses frequently, you might not want to do things this way.
- Since you've no provision for storing a password or any other means of authentication, if you allow users to update their information, you run a minor risk of allowing a malicious change. (The risk isn't as great as it seems because you probably won't be publishing the complete mailing list; an attacker would have to guess the names of people on your mailing list.)

These aren't necessarily bad realities in which to be locked. However, a good data modeler recognizes that every line of code in the .sql file has profound implications for the Web service.

Papering Over Your Mistakes with Triggers

Suppose that you've been using the above data model to collect the names of Web site readers who'd like to be alerted when you add new articles. You haven't sent any notices for two months. You want to send everyone who signed up in the last two months a "Welcome to my Web service; thanks for signing up; here's what's new" message. You want to send the older subscribers a simple "here's what's new" message. But you can't do this because you didn't store a registration date. It is easy enough to fix the table:

```
alter table mailing_list add (registration_date
```

```
date);
```

But what if you have 15 different Web scripts that use this table? The ones that query it aren't a problem. If they don't ask for the new column, they won't get it and won't realize that the table has been changed (this is one of the big selling features of the RDBMS). But the scripts that update the table will all need to be changed. If you miss a script, you're potentially stuck with a table where various random rows are missing critical information.

Oracle has a solution to your problem: *triggers*. A trigger is a way of telling Oracle "any time anyone touches this table, I want you to execute the following little fragment of code". Here's how we define the trigger `mailing_list_registration_date`:

```
create trigger mailing_list_registration_date
before insert on mailing_list
for each row
when (new.registration_date is null)
begin
 :new.registration_date := sysdate;
end;
```

Note that the trigger only runs when someone is trying to insert a row with a NULL registration date. If for some reason you need to copy over records from another database and they have a registration date, you don't want this trigger overwriting it with the date of the copy.

A second point to note about this trigger is that it runs `for each row`. This is called a "row-level trigger" rather than a "statement-level trigger", which runs once per transaction, and is usually not what you want.

A third point is that we're using the magic Oracle procedure `sysdate`, which will return the current time. The Oracle `date` type is precise to the second even though the default is to display only the day.

A fourth point is that, starting with Oracle 8, we could have done this more cleanly by adding a `default sysdate` instruction to the column's definition.

The final point worth noting is the `:new.` syntax. This lets you refer to the new values being inserted. There is an analogous :old. feature, which is useful for update triggers:

```
create or replace trigger mailing_list_update
before update on mailing_list
for each row
when (new.name <> old.name)
begin
  -- user is changing his or her name
  -- record the fact in an audit table
  insert into mailing_list_name_changes
    (old_name, new_name)
  values
    (:old.name, :new.name);
end;
/
show errors
```

This time we used the `create or replace` syntax. This keeps us from having to `drop trigger mailing_list_update` if we want to change the trigger definition. We added a comment using the SQL comment shortcut "--". The syntax `new.` and `old.` is used in the trigger definition, limiting the conditions under which the trigger runs. Between the `begin` and `end`, we're in a

PL/SQL block. This is Oracle's procedural language, described later, in which `new.name` would mean "the `name` element from the record in `new`". So you have to use `:new` instead. It is obscurities like this that lead to competent Oracle consultants being paid $200+ per hour.

The "/" and `show errors` at the end are instructions to Oracle's SQL*Plus program. The slash says "I'm done typing this piece of PL/SQL, please evaluate what I've typed." The "show errors" says "if you found anything to object to in what I just typed, please tell me".

The Discussion Forum -- philg's personal odyssey

Back in 1995, I built a threaded discussion forum, described *ad nauseum* in http://philip.greenspun.com/wtr/dead-trees/53013.htm. Here's how I stored the postings:

```
create table bboard (
        msg_id          char(6) not null primary
key,
        refers_to       char(6),
        email           varchar(200),
        name            varchar(200),
        one_line        varchar(700),
        message         clob,
        notify          char(1) default 'f' check
(notify in ('t','f')),
        posting_time    date,
        sort_key        varchar(600)
);
```

German order reigns inside the system itself: messages are uniquely keyed with `msg_id`, refer to each other (i.e., say "I'm a response to msg X") with `refers_to`, and a thread can be displayed conveniently by using the `sort_key` column.

Italian chaos is permitted in the `email` and `name` columns; users could remain anonymous, masquerade as "president@whitehouse.gov" or give any name.

This seemed like a good idea when I built the system. I was concerned that it work reliably. I didn't care whether or not users put in bogus content; the admin pages made it really easy to remove such postings and, in any case, if someone had something interesting to say but needed to remain anonymous, why should the system reject their posting?

One hundred thousand postings later, as the moderator of the photo.net Q&A forum, I began to see the dimensions of my data modeling mistakes.

First, anonymous postings and fake email addresses didn't come from Microsoft employees revealing the dark truth about their evil bosses. They came from complete losers trying and failing to be funny or wishing to humiliate other readers. Some fake addresses came from people scared by the rising tide of spam email (not a serious problem back in 1995).

Second, I didn't realize how the combination of my email alert systems, fake email addresses, and Unix mailers would result in my personal mailbox filling up with messages that couldn't be delivered to "asdf@asdf.com" or "duh@duh.net".

Although the solution involved changing some Web scripts, fundamentally the fix was add a column to store the IP address from which a post was made:

```
alter table bboard add (originating_ip
varchar(16));
```

Keeping these data enabled me to see that most of the anonymous posters were people who'd been using the forum for some time, typically from the same IP address. I just sent them mail and asked them to stop, explaining the problem with bounced email.

After four years of operating the photo.net community, it became apparent that we needed ways to

- display site history for users who had changed their email addresses
- discourage problem users from burdening the moderators and the community
- carefully tie together user-contributed content in the various subsystems of photo.net

The solution was obvious to any experienced database nerd: a canonical users table and then content tables that reference it. Here's a simplified version of the data model, taken from a toolkit for building online communities, describe in http://philip.greenspun.com/panda/community:

```
create table users (
        user_id                 integer not null
primary key,
        first_names             varchar(100) not
null,
        last_name               varchar(100) not
null,
        email                   varchar(100) not
null unique,
```

```
        ..
);

create table bboard (
        msg_id          char(6) not null primary
key,
        refers_to       char(6),
        topic           varchar(100) not null
references bboard_topics,
        category        varchar(200),   -- only
used for categorized Q&A forums
        originating_ip  varchar(16),    -- stored
as string, separated by periods
        user_id         integer not null
references users,
        one_line        varchar(700),
        message         clob,
        -- html_p - is the message in html or not
        html_p          char(1) default 'f' check
(html_p in ('t','f')),
        ...
);

create table classified_ads (
        classified_ad_id        integer not null
primary key,
        user_id                 integer not null
references users,
        ...
);
```

Note that a contributor's name and email address no longer appear in the `bboard` table. That doesn't mean we don't know who posted a message. In fact, this data model can't even represent an anonymous posting: `user_id integer not null references users` requires that each posting be associated with a user ID and that there actually be a row in the `users` table with that ID.

First, let's talk about how much fun it is to move a live-on-the-Web 600,000 hit/day service from one data model to another. In this case, note that the original `bboard` data model had a single `name` column. The community system has separate columns for first and last names. A conversion script can easily split up "Joe Smith" but what is it to do with William Henry Gates III?

How do we copy over anonymous postings? Remember that Oracle is not flexible or intelligent. We said that we wanted every row in the `bboard` table to reference a row in the `users` table. Oracle will abort any transaction that would result in a violation of this integrity constraint. So we either have to drop all those anonymous postings (and any non-anonymous postings that refer to them) or we have to create a user called "Anonymous" and assign all the anonymous postings to that person. The technical term for this kind of solution is *kludge*.

A more difficult problem than anonymous postings is presented by long-time users who have difficulty typing and or keeping a job. Consider a user who has identified himself as

1. Joe Smith; jsmith@ibm.com
2. Jo Smith; jsmith@ibm.com (typo in name)
3. Joseph Smith; jsmth@ibm.com (typo in email)
4. Joe Smith; cantuseworkaddr@hotmail.com (new IBM policy)
5. Joe Smith-Jones; joe_smithjones@hp.com (got married, changed name, changed jobs)
6. Joe Smith-Jones; jsmith@somedivision.hp.com (valid but not canonical corporate email address)

7. Josephina Smith; jsmith@somedivision.hp.com (sex change; divorce)
8. Josephina Smith; josephina_smith@hp.com (new corporate address)
9. Siddhartha Bodhisattva; josephina_smith@hp.com (change of philosophy)
10. Siddhartha Bodhisattva; thinkwaitfast@hotmail.com (traveling for awhile to find enlightenment)

Contemporary community members all recognize these postings as coming from the same person but it would be very challenging even to build a good semi-automated means of merging postings from this person into one user record.

Once we've copied everything into this new *normalized* data model, notice that we can't dig ourselves into the same hole again. If a user has contributed 1000 postings, we don't have 1000 different records of that person's name and email address. If a user changes jobs, we need only update one column in one row in one table.

The `html_p` column in the new data model is worth mentioning. In 1995, I didn't understand the problems of user-submitted data. Some users will submit plain text, which seems simple, but in fact you can't just spit this out as HTML. If user A typed < or > characters, they might get swallowed by user B's Web browser. Does this matter? Consider that "<g>" is interpreted in various online circles as an abbreviation for "grin" but by Netscape Navigator as an unrecognized (and therefore ignore) HTML tag. Compare the meaning of

> "We shouldn't think it unfair that Bill Gates has more wealth than the 100 million poorest Americans *combined*. After all, he invented the personal computer, the graphical user interface, and the Internet."

with

> "We shouldn't think it unfair that Bill Gates has more wealth than the 100 million poorest Americans *combined*. After all, he invented the personal computer, the graphical user interface, and the Internet. <g>"

It would have been easy enough for me to make sure that such characters never got interpreted as markup. In fact, with AOLserver one can do it with a single call to the built-in procedure `ns_quotehtml`. However, consider the case where a nerd posts some HTML. Other users would then see

> "For more examples of my brilliant thinking and modesty, check out my home page."

I discovered that the only real solution is to ask the user whether the submission is an HTML fragment or plain text, show the user an approval page where the content may be previewed, and then remember what the user told us in an `html_p` column in the database.

Is this data model perfect? Permanent? Absolutely. It will last for at least... Whoa! Wait a minute. I didn't know that Dave Clark was replacing his original Internet Protocol, which the world has been running since around 1980, with IPv6 (http://www.faqs.org/rfcs/rfc2460.html). In the near future, we'll have IP addresses that are

128 bits long. That's 16 bytes, each of which takes two hex characters to represent. So we need 32 characters plus at least 7 more for periods that separate the hex digits. We might also need a couple of characters in front to say "this is a hex representation". Thus our brand new data model in fact has a crippling deficiency. How easy is it to fix? In Oracle:

```
alter table bboard modify (originating_ip
varchar(50));
```

You won't always get off this easy. Oracle won't let you shrink a column from a maximum of 50 characters to 16, even if no row has a value longer than 16 characters. Oracle also makes it tough to add a column that is constrained `not null`.

Representing Web Site Core Content

Free-for-all Internet discussions can often be useful and occasionally are compelling, but the anchor of a good Web site is usually a set of carefully authored extended documents. Historically these have tended to be stored in the Unix file system and they don't change too often. Hence I refer to them as *static pages*. Examples of static pages on the photo.net server include this book chapter, the tutorial on light for photographers at http://www.photo.net/making-photographs/light.

We have some big goals to consider. We want the data in the database to

- help community experts figure out which articles need revision and which new articles would be most valued by the community at large.
- help contributors work together on a draft article or a new version of an old article.
- collect and organize reader comments and discussion, both for presentation to other

readers but also to assist authors in keeping content up-to-date.
- collect and organize reader-submitted suggestions of related content out on the wider Internet (i.e., links).
- help point readers to new or new-to-them content that might interest them, based on what they've read before or based on what kind of content they've said is interesting.

The big goals lead to some more concrete objectives:

- We will need a table that holds the static pages themselves.
- Since there are potentially many comments per page, we need a separate table to hold the user-submitted comments.
- Since there are potentially many related links per page, we need a separate table to hold the user-submitted links.
- Since there are potentially many authors for one page, we need a separate table to register the author-page many-to-one relation.
- Considering the "help point readers to stuff that will interest them" objective, it seems that we need to store the category or categories under which a page falls. Since there are potentially many categories for one page, we need a separate table to hold the mapping between pages and categories.

```
create table static_pages (
        page_id         integer not null primary
key,
        url_stub        varchar(400) not null
unique,
        original_author integer references
users(user_id),
        page_title      varchar(4000),
        page_body       clob,
        obsolete_p      char(1) default 'f' check
```

```
(obsolete_p in ('t','f')),
        members_only_p  char(1) default 'f' check
(members_only_p in ('t','f')),
        price           number,
        copyright_info  varchar(4000),
        accept_comments_p       char(1) default
't' check (accept_comments_p in ('t','f')),
        accept_links_p          char(1) default
't' check (accept_links_p in ('t','f')),
        last_updated            date,
        -- used to prevent minor changes from
looking like new content
        publish_date            date
);

create table static_page_authors (
        page_id         integer not null
references static_pages,
        user_id         integer not null
references users,
        notify_p        char(1) default 't' check
(notify_p in ('t','f')),
        unique(page_id,user_id)
);
```

Note that we use a generated integer `page_id` key for this table. We could key the table by the `url_stub` (filename), but that would make it very difficult to reorganize files in the Unix file system (something that should actually happen very seldom on a Web server; it breaks links from foreign sites).

How to generate these unique integer keys when you have to insert a new row into `static_pages`? You could

- lock the table
- find the maximum `page_id` so far
- add one to create a new unique `page_id`
- insert the row
- commit the transaction (releases the table lock)

Much better is to use Oracle's built-in sequence generation facility:

```
create sequence page_id_sequence start with 1;
```

Then we can get new page IDs by using `page_id_sequence.nextval` in INSERT statements (see the Transactions chapter for a fuller discussion of sequences).

Reference

Here is a summary of the data modeling tools available to you in Oracle, each hyperlinked to the Oracle documentation. This reference section covers the following:

- data types
- statements for creating, altering, and dropping tables
- constraints

Data Types

For each column that you define for a table, you must specify the data type of that column. Here are your options:

Character Data

char(n) A fixed-length character string, e.g., `char(200)` will take up 200 bytes regardless of how long the string actually is. This works well when the data truly are of fixed size, e.g., when you are recording a user's sex as "m" or "f". This works badly when the data are of variable length. Not only does it waste space on the disk and in the memory cache, but it makes comparisons fail. For example, suppose you insert "rating" into a `comment_type` column of type `char(30)` and then your Tcl program queries the database. Oracle sends this column value back to procedural language clients padded with enough spaces to make up 30 total characters. Thus if you have a comparison within Tcl of whether `$comment_type == "rating"`, the comparison will fail because `$comment_type` is actually "rating" followed by 24 spaces.

	The maximum length char in Oracle8 is 2000 bytes.
varchar(n)	A variable-length character string, up to 4000 bytes long in Oracle8. These are stored in such a way as to minimize disk space usage, i.e., if you only put one character into a column of type `varchar(4000)`, Oracle only consumes two bytes on disk. The reason that you don't just make all the columns `varchar(4000)` is that the Oracle indexing system is limited to indexing keys of about 700 bytes.
clob	A variable-length character string, up to 4 gigabytes long in Oracle8. The CLOB data type is useful for accepting user input from such applications as discussion forums. Sadly, Oracle8 has tremendous limitations on how CLOB data may be inserted, modified, and queried. Use `varchar(4000)` if you can and prepare to suffer if you can't. In a spectacular demonstration of what happens when companies don't follow the lessons of *The Mythical Man Month*, the regular string functions don't work on CLOBs. You need to call identically named functions in the DBMS_LOB package. These functions take the same arguments but in different orders. You'll never be able to write a working line of code without first reading the DBMS_LOB section of the *Oracle8 Server Application Developer's Guide*.
nchar, nvarchar, nclob	The n prefix stands for "national character set". These work like char, varchar, and clob but for multi-byte characters (e.g., Unicode; see http://www.unicode.org).

Numeric Data	
number	Oracle actually only has one internal data type that is used for storing numbers. It can handle 38 digits of precision and exponents from -130 to +126. If you want to get fancy, you can specify precision and scale limits. For example, `number(3,0)` says "round everything to an integer [scale 0] and accept numbers than range from -999 to +999". If you're American and commercially minded, `number(9,2)` will probably work well for storing prices in dollars and cents (unless you're selling stuff to Bill Gates, in which case the billion dollar limit imposed by the precision of 9 might prove constraining). If you *are* Bill Gates, you might not want to get distracted by insignificant numbers: Tell Oracle to round everything to the nearest million with `number(38,-6)`.
integer	In terms of storage consumed and behavior, this is not any different from `number(38)` but I think it reads better and it is more in line with ANSI SQL (which would be a standard if anyone actually implemented it).
Dates and Date/Time Intervals (Version 9i and newer)	
timestamp	A point in time, recorded with sub-second precision. When creating a column you specify the number of digits of precision beyond one second from 0 (single second precision) to 9 (nanosecond precision). Oracle's calendar can handle dates between between January 1, 4712 BC and December 31, 9999 AD. You can put in values with the `to_timestamp` function and query them out using the `to_char` function. Oracle offers several variants of this datatype for coping with data aggregated across multiple timezones.
interval year to month	An amount of time, expressed in years and months.
interval day to second	An amount of time, expressed in days, hours, minutes, and seconds. Can be precise down to the nanosecond if desired.
Dates and Date/Time Intervals (Versions 8i and earlier)	
date	Obsolete as of version 9i. A point in time, recorded with one-second precision, between January 1, 4712 BC and December 31, 4712 AD.

	You can put in values with the `to_date` function and query them out using the `to_char` function. If you don't use these functions, you're limited to specifying the date with the default system format mask, usually 'DD-MON-YY'. This is a good recipe for a Year 2000 bug since January 23, 2000 would be '23-JAN-00'. On better-maintained systems, this is often the ANSI default: 'YYYY-MM-DD', e.g., '2000-01-23' for January 23, 2000.
number	Hey, isn't this a typo? What's `number` doing in the date section? It is here because this is how Oracle versions prior to 9i represented date-time intervals, though their docs never say this explicitly. If you add numbers to dates, you get new dates. For example, tomorrow at exactly this time is `sysdate+1`. To query for stuff submitted in the last hour, you limit to `submitted_date > sysdate - 1/24`.
	Binary Data
blob	BLOB stands for "Binary Large OBject". It doesn't really have to be all that large, though Oracle will let you store up to 4 GB. The BLOB data type was set up to permit the storage of images, sound recordings, and other inherently binary data. In practice, it also gets used by fraudulent application software vendors. They spend a few years kludging together some nasty format of their own. Their MBA executive customers demand that the whole thing be RDBMS-based. The software vendor learns enough about Oracle to "stuff everything into a BLOB". Then all the marketing and sales folks are happy because the application is now running from Oracle instead of from the file system. Sadly, the programmers and users don't get much because you can't use SQL very effectively to query or update what's inside a BLOB.
bfile	A binary file, stored by the operating system (typically Unix) and kept track of by Oracle. These would be useful when you need to get to information both from SQL (which is kept purposefully ignorant about what goes on in the wider world) and from an application that can only read from standard files (e.g., a typical Web server). The bfile data type is pretty new but to my mind it is already

> obsolete: Oracle 8.1 (8i) lets external applications view content in the database as though it were a file on a Windows NT server. So why not keep everything as a BLOB and enable Oracle's Internet File System?

Despite this plethora of data types, Oracle has some glaring holes that torture developers. For example, there is no Boolean data type. A developer who needs an `approved_p` column is forced to use `char(1) check(this_column in ('t','f'))` and then, instead of the clean query `where approved_p` is forced into `where approved_p = 't'`.

Oracle8 includes a limited ability to create your own data types. Covering these is beyond the scope of this book. See Oracle8 Server Concepts, User-Defined Datatypes.

Tables

The basics:

```
CREATE TABLE your_table_name (
       the_key_column        key_data_type
PRIMARY KEY,
       a_regular_column      a_data_type,
       an_important_column   a_data_type NOT
NULL,
       ... up to 996 intervening columns in
Oracle8 ...
       the_last_column       a_data_type
);
```

Even in a simple example such as the one above, there are few items worth noting. First, I like to define the key column(s) at the very top. Second, the `primary key` constraint has some powerful effects. It forces `the_key_column` to be non-null. It causes the creation of an index on `the_key_column`, which will slow down updates to `your_table_name` but improve the speed of access when someone queries for a row with a particular value of `the_key_column`. Oracle checks this index when inserting any new row and aborts the transaction if there is already a row with the same value for `the_key_column`. Third, note that there is no comma following the definition of the last row. If you are careless and leave the comma in, Oracle will give you a very confusing error message.

If you didn't get it right the first time, you'll probably want to

```
alter table your_table_name add (new_column_name
a_data_type any_constraints);
```

or

```
alter table your_table_name modify
(existing_column_name new_data_type
new_constraints);
```

In Oracle 8i you can drop a column:

```
alter table your_table_name drop column
existing_column_name;
```

(see http://www.oradoc.com/keyword/drop_column).

If you're still in the prototype stage, you'll probably find it easier to simply

```
drop table your_table_name;
```

and recreate it. At any time, you can see what you've got defined in the database by querying Oracle's *Data Dictionary*:

```
SQL> select table_name from user_tables order by
table_name;

TABLE_NAME
------------------------------
ADVS
ADV_CATEGORIES
ADV_GROUPS
```

```
ADV_GROUP_MAP
ADV_LOG
ADV_USER_MAP
AD_AUTHORIZED_MAINTAINERS
AD_CATEGORIES
AD_DOMAINS
AD_INTEGRITY_CHECKS
BBOARD
...
STATIC_CATEGORIES
STATIC_PAGES
STATIC_PAGE_AUTHORS
USERS
...
```

after which you will typically type `describe table_name_of_interest` in SQL*Plus:

```

SQL> describe users;
 Name                            Null?    Type
 ------------------------------- -------- ----
 USER_ID                         NOT NULL
NUMBER(38)
 FIRST_NAMES                     NOT NULL
VARCHAR2(100)
 LAST_NAME                       NOT NULL
VARCHAR2(100)
 PRIV_NAME
NUMBER(38)
 EMAIL                           NOT NULL
VARCHAR2(100)
 PRIV_EMAIL
NUMBER(38)
 EMAIL_BOUNCING_P                         CHAR(1)
 PASSWORD                        NOT NULL
VARCHAR2(30)
 URL
VARCHAR2(200)
 ON_VACATION_UNTIL                        DATE
 LAST_VISIT                               DATE
 SECOND_TO_LAST_VISIT                     DATE
 REGISTRATION_DATE                        DATE
```

```
 REGISTRATION_IP
VARCHAR2(50)
 ADMINISTRATOR_P                                         CHAR(1)
 DELETED_P                                               CHAR(1)
 BANNED_P                                                CHAR(1)
 BANNING_USER
NUMBER(38)
 BANNING_NOTE
VARCHAR2(4000)
```

Note that Oracle displays its internal data types rather than the ones you've given, e.g., `number(38)` rather than `integer` and `varchar2` instead of the specified `varchar`.

Constraints

When you're defining a table, you can constrain single rows by adding some magic words after the data type:

- `not null`; requires a value for this column
- `unique`; two rows can't have the same value in this column (side effect in Oracle: creates an index)
- `primary key`; same as `unique` except that no row can have a null value for this column and other tables can refer to this column
- `check`; limit the range of values for column, e.g., `rating   integer check(rating > 0 and rating <= 10)`
- `references`; this column can only contain values present in another table's primary key column, e.g., `user_id           not null references users` in the `bboard` table forces the `user_id` column to only point to valid users. An interesting twist is that you don't have to give a data type for `user_id`; Oracle assigns this column to whatever data type the foreign key has (in this case `integer`).

Constraints can apply to multiple columns:

```
create table static_page_authors (
```

```
        page_id         integer not null
references static_pages,
        user_id         integer not null
references users,
        notify_p        char(1) default 't' check
(notify_p in ('t','f')),
        unique(page_id,user_id)
);
```

Oracle will let us keep rows that have the same `page_id` and rows that have the same `user_id` but not rows that have the same value in both columns (which would not make sense; a person can't be the author of a document more than once). Suppose that you run a university distinguished lecture series. You want speakers who are professors at other universities or at least PhDs. On the other hand, if someone controls enough money, be it his own or his company's, he's in. Oracle stands ready:

```

create table distinguished_lecturers (
        lecturer_id             integer primary
key,
        name_and_title          varchar(100),
        personal_wealth         number,
        corporate_wealth        number,
        check (instr(upper(name_and_title),'PHD')
<> 0
               or
instr(upper(name_and_title),'PROFESSOR') <> 0
               or (personal_wealth +
corporate_wealth) > 1000000000)
);

insert into distinguished_lecturers
values
(1,'Professor Ellen Egghead',-10000,200000);

1 row created.

insert into distinguished_lecturers
values
(2,'Bill Gates,
innovator',75000000000,18000000000);
```

```
1 row created.

insert into distinguished_lecturers
values
(3,'Joe Average',20000,0);

ORA-02290: check constraint
(PHOTONET.SYS_C001819) violated
```

As desired, Oracle prevented us from inserting some random average loser into the `distinguished_lecturers` table, but the error message was confusing in that it refers to a constraint given the name of "SYS_C001819" and owned by the PHOTONET user. We can give our constraint a name at definition time:

```
create table distinguished_lecturers (
        lecturer_id             integer primary
key,
        name_and_title          varchar(100),
        personal_wealth         number,
        corporate_wealth        number,
        constraint ensure_truly_distinguished
        check (instr(upper(name_and_title),'PHD')
<> 0
               or
instr(upper(name_and_title),'PROFESSOR') <> 0
               or (personal_wealth +
corporate_wealth) > 1000000000)
);

insert into distinguished_lecturers
values
(3,'Joe Average',20000,0);

ORA-02290: check constraint
(PHOTONET.ENSURE_TRULY_DISTINGUISHED) violated
```

Now the error message is easier to understand by application programmers.

Creating More Elaborate Constraints with Triggers

The default Oracle mechanisms for constraining data are not always adequate. For example, the ArsDigita Community System auction module has a table called `au_categories`. The `category_keyword` column is a unique shorthand way of referring to a category in a URL. However, this column may be NULL because it is not the primary key to the table. The shorthand method of referring to the category is optional.

```
create table au_categories (
        category_id             integer primary
key,
        -- shorthand for referring to this
category,
        -- e.g. "bridges", for use in URLs
        category_keyword        varchar(30),
        -- human-readable name of this category,
        -- e.g. "All types of bridges"
        category_name           varchar(128) not
null
);
```

We can't add a UNIQUE constraint to the `category_keyword` column. That would allow the table to only have one row where `category_keyword` was NULL. So we add a trigger that can execute an arbitrary PL/SQL expression and raise an error to prevent an INSERT if necessary:

```
create or replace trigger au_category_unique_tr
before insert
on au_categories
for each row
declare
        existing_count integer;
begin
        select count(*) into existing_count from
au_categories
            where category_keyword =
:new.category_keyword;
        if existing_count > 0
        then
            raise_application_error(-20010,
'Category keywords must be unique if used');
```

```
        end if;
end;
```

This trigger queries the table to find out if there are any matching keywords already inserted. If there are, it calls the built-in Oracle procedure `raise_application_error` to abort the transaction.

The True Oracle Religion

Next: queries

---

philg@mit.edu

Add a comment