Data Warehousing

40 min. read · View original

In the preceding chapters, you've been unwittingly immersed in the world of on-line transaction processing (OLTP). This world carries with it some assumptions:

1. Only store a piece of information once. If there are N copies of something in the database and you need to change it, you might forget to change it in all N places. Note that only storing information in one spot also enables updates to be fast.
2. It is okay if queries are complex because they are authored infrequently and by professional programmers.
3. Never sequentially scan large tables; reread the tuning chapter if Oracle takes more than one second to perform any operation.

These are wonderful rules to live by if one is booking orders, adding user comments to pages, recording a clickthrough, or seeing if someone is authorized to download a file.

You can probably continue to live by these rules if you want some answers from your data. Write down a list of questions that are important and build some report pages. You might need materialized views to make these reports fast and your queries might be complex, but you don't need to leave the OLTP world simply because business dictates that you answer a bunch of questions.

Why would anyone leave the OLTP world? Data warehousing is useful when you don't know what questions to ask.

What it means to facilitate exploration

Data exploration is only useful when non-techies are able to explore. That means people with very weak skills will be either authoring queries or specifying queries with menus. You can't ask a marketing executive to look at a 600-table data model and pick and choose the relevant columns. You can't ask a salesman to pull the answer to "is this a repeat customer or not?" out of a combination of the `customers` and `orders` tables.

If a data exploration environment is to be useful it must fulfill the following criteria:

- complex questions can be asked with a simple SQL query
- different questions imply very similar SQL query structure
- very different questions require very similar processing time to answer
- exploration can be done from any computer anywhere

The goal is that a business expert can sit down at a Web browser, use a sequence of forms to specify a query, and get a result back in an amount of time that seems reasonable.

It will be impossible to achieve this with our standard OLTP data models. Answering a particular question may require JOINing in four or five extra tables, which could result in a 10,000-fold increase in processing time. Even if a novice user could be guided to specifying a 7-way JOIN from among 600 tables, that person would have no way of understanding or predicting query processing time. Finally there is the question of whether you want novices querying your OLTP tables. If they are only typing SELECTs they might not be doing too much long-term harm but the short-term processing load might result in a system that feels crippled.

It is time to study *data warehousing*.

Classical Retail Data Warehousing

> "Another segment of society that has constructed a language of its own is business. ... [The businessman] is speaking a language that is familiar to him and dear to him. Its portentous nouns and verbs invest ordinary events with high adventure; the executive walks among ink erasers caparisoned like a knight. This we should be tolerant of--every man of spirit wants to ride a white horse. ... A good many of the special words of business seem designed more to express the user's dreams than to express his precise meaning."
> -- last chapter of *The Elements of Style*, Strunk and White

Let's imagine a conversation between the Chief Information Officer of WalMart and a sales guy from Sybase. We've picked these companies for concreteness but they stand for "big Management Information System (MIS) user" and "big relational database management system (RDBMS) vendor".

> Walmart: "I want to keep track of sales in all of my stores simultaneously."
> Sybase: "You need our wonderful RDBMS software. You can stuff data in as sales are rung up at cash registers and simultaneously query data out right here in your office. That's the beauty of concurrency control."

So Walmart buys a $1 million Sun E10000 multi-CPU server and a $500,000 Sybase license. They buy *Database Design for Smarties* and build themselves a normalized SQL data model:

```
create table product_categories (
        product_category_id     integer primary key,
        product_category_name   varchar(100) not null
);

create table manufacturers (
        manufacturer_id         integer primary key,
        manufacturer_name       varchar(100) not null
);

create table products (
        product_id              integer primary key,
        product_name            varchar(100) not null,
        product_category_id     references product_categories,
        manufacturer_id         references manufacturers
);

create table cities (
        city_id                 integer primary key,
        city_name               varchar(100) not null,
        state                   varchar(100) not null,
        population              integer not null
);

create table stores (
        store_id                integer primary key,
        city_id                 references cities,
```

```
        store_location          varchar(200) not null,
        phone_number            varchar(20)
);

create table sales (
        product_id      not null references products,
        store_id        not null references stores,
        quantity_sold   integer not null,
        -- the Oracle "date" type is precise to the second
        -- unlike the ANSI date datatype
        date_time_of_sale       date not null
);

-- put some data in

insert into product_categories values (1, 'toothpaste');
insert into product_categories values (2, 'soda');

insert into manufacturers values (68, 'Colgate');
insert into manufacturers values (5, 'Coca Cola');

insert into products values (567, 'Colgate Gel Pump 6.4 oz.', 1, 68);
insert into products values (219, 'Diet Coke 12 oz. can', 2, 5);

insert into cities values (34, 'San Francisco', 'California', 700000);
insert into cities values (58, 'East Fishkill', 'New York', 30000);

insert into stores values (16, 34, '510 Main Street', '415-555-1212');
insert into stores values (17, 58, '13 Maple Avenue', '914-555-1212');

insert into sales values (567, 17, 1, to_date('1997-10-22 09:35:14', 'YYYY-MM-DD
HH24:MI:SS'));
insert into sales values (219, 16, 4, to_date('1997-10-22 09:35:14', 'YYYY-MM-DD
HH24:MI:SS'));
insert into sales values (219, 17, 1, to_date('1997-10-22 09:35:17', 'YYYY-MM-DD
HH24:MI:SS'));

-- keep track of which dates are holidays
-- the presence of a date (all dates will be truncated to midnight)
-- in this table indicates that it is a holiday
create table holiday_map (
holiday_date            date primary key
);

-- where the prices are kept
create table product_prices (
product_id      not null references products,
from_date       date not null,
price           number not null
);

insert into product_prices values (567,'1997-01-01',2.75);
insert into product_prices values (219,'1997-01-01',0.40);
```

What do we have now?

SALES table

product id	store id	quantity sold	date/time of sale
567	17	1	1997-10-22 09:35:14
219	16	4	1997-10-22 09:35:14
219	17	1	1997-10-22 09:35:17

…

PRODUCTS table

product id	product name	product category	manufacturer id
567	Colgate Gel Pump 6.4 oz.	1	68
219	Diet Coke 12 oz. can	2	5
…			

PRODUCT_CATEGORIES table

product category id	product category name
1	toothpaste
2	soda
…	

MANUFACTURERS table

manufacturer id	manufacturer name
68	Colgate
5	Coca Cola
…	

STORES table

store id	city id	store location	phone number
16	34	510 Main Street	415-555-1212
17	58	13 Maple Avenue	914-555-1212
…			

CITIES table

city id	city name	state	population
34	San Francisco	California	700,000
58	East Fishkill	New York	30,000
…			

After a few months of stuffing data into these tables, a WalMart executive, call her Jennifer Amolucre asks "I noticed that there was a Colgate promotion recently, directed at people who live in small towns. How much Colgate toothpaste did we sell in those towns yesterday? And how much on the same day a month ago?"

At this point, reflect that because the data model is normalized, this information can't be obtained from scanning one table. A normalized data model is one in which all the information in a row depends only on the primary key. For example, the city population is not contained in the `stores` table. That information is stored once per city in the `cities` table and only `city_id` is kept in the `stores` table. This ensures efficiency for transaction processing. If Walmart has to update a city's population, only one record on disk need be touched. As computers get faster, what is more interesting is the consistency of this approach. With the city population kept only in one place, there is no risk that updates will be applied to some records and not to others. If there are multiple stores in the same city, the population will be pulled out of the same slot for all the stores all the time.

Ms. Amolucre's query will look something like this...

```
select sum(sales.quantity_sold)
from sales, products, product_categories, manufacturers, stores, cities
where manufacturer_name = 'Colgate'
and product_category_name = 'toothpaste'
and cities.population < 40000
and trunc(sales.date_time_of_sale) = trunc(sysdate-1)  -- restrict to yesterday
and sales.product_id = products.product_id
and sales.store_id = stores.store_id
and products.product_category_id = product_categories.product_category_id
and products.manufacturer_id = manufacturers.manufacturer_id
```

```
and stores.city_id = cities.city_id;
```

This query would be tough for a novice to read and, being a 6-way JOIN of some fairly large tables, might take quite a while to execute. Moreover, these tables are being updated as Ms. Amolucre's query is executed.

Soon after the establishment of Jennifer Amolucre's quest for marketing information, store employees notice that there are times during the day when it is impossible to ring up customers. Any attempt to update the database results in the computer freezing up for 20 minutes. Eventually the database administrators realize that the system collapses every time Ms. Amolucre's toothpaste query gets run. They complain to Sybase tech support.

> Walmart: "We type in the toothpaste query and our system wedges."
> Sybase: "Of course it does! You built an on-line transaction processing (OLTP) system. You can't feed it a decision support system (DSS) query and expect things to work!"
> Walmart: "But I thought the whole point of SQL and your RDBMS was that users could query and insert simultaneously."
> Sybase: "Uh, not exactly. If you're reading from the database, nobody can write to the database. If you're writing to the database, nobody can read from the database. So if you've got a query that takes 20 minutes to run and don't specify special locking instructions, nobody can update those tables for 20 minutes."
> Walmart: "That sounds like a bug."
> Sybase: "Actually it is a feature. We call it *pessimistic locking*."
> Walmart: "Can you fix your system so that it doesn't lock up?"
> Sybase: "No. But we made this great loader tool so that you can copy everything from your OLTP system into a separate DSS system at 100 GB/hour."

Since you are reading this book, you are probably using Oracle, which is one of the few database management systems that achieves consistency among concurrent users via versioning rather than locking (the other notable example is the free open-source PostgreSQL RDBMS). However, even if you are using Oracle, where readers never wait for writers and writers never wait for readers, you still might not want the transaction processing operation to slow down in the event of a marketing person entering an expensive query.

Basically what IT vendors want Walmart to do is set up another RDBMS installation on a separate computer. Walmart needs to buy another $1 million of computer hardware. They need to buy another RDBMS license. They also need to hire programmers to make sure that the OLTP data is copied out nightly and stuffed into the DSS system--*data extraction*. Walmart is now building the *data warehouse*.

Insight 1

A data warehouse is a separate RDBMS installation that contains copies of data from on-line systems. A physically separate data warehouse is not absolutely necessary if you have a lot of extra computing horsepower. With a DBMS that uses optimistic locking you might even be able to get away with keeping only one copy of your data.

As long as we're copying...

As long as you're copying data from the OLTP system into the DSS system ("data warehouse"), you might as well think about organizing and indexing it for faster retrieval. Extra indices on production tables are bad because they slow down inserts and updates. Every time you add or modify a row to a table, the RDBMS has to update the indices to keep them consistent. But in a data warehouse, the data are static. You build indices once and they take up space and sometimes make queries faster and that's it.

If you know that Jennifer Amolucre is going to do the toothpaste query every day, you can denormalize the data model for her. If you add a `town_population` column to the `stores` table and copy in data from the `cities` table, for example, you sacrifice some cleanliness of data model but now Ms. Amolucre's query only requires a 5-way JOIN. If you add

manufacturer and product_category columns to the sales table, you don't need to JOIN in the products table.

Where does denormalization end?

Once you give up the notion that the data model in the data warehouse need bear some resemblance to the data model in the OLTP system, you begin to think about reorganizing the data model further. Remember that we're trying to make sure that new questions can be asked by people with limited SQL experience, i.e., many different questions can be answered with morphologically similar SQL. Ideally the task of constructing SQL queries can be simplified enough to be doable from a menu system. Also, we are trying to delivery predictable response time. A minor change in a question should not result in a thousand-fold increase in system response time.

The irreducible problem with the OLTP data model is that it is tough for novices to construct queries. Given that computer systems are not infinitely fast, a practical problem is inevitably that the response times of a query into the OLTP tables will vary in a way that is unpredictable to the novice.

Suppose, for example, that Bill Novice wants to look at sales on holidays versus non-holidays with the OLTP model. Bill will need to go look at the data model, which on a production system will contain hundreds of tables, to find out if any of them contain information on whether or not a date is a holiday. Then he will need to use it in a query, something that isn't obvious given the peculiar nature of the Oracle date data type:

```
select sum(sales.quantity_sold)
from sales, holiday_map
where trunc(sales.date_time_of_sale) = trunc(holiday_map.holiday_date)
```

That one was pretty simple because JOINing to the holiday_map table knocks out sales on days that aren't holidays. To compare to sales on non-holidays, he will need to come up with a different query strategy, one that knocks out sales on days that *are* holidays. Here is one way:

```
select sum(sales.quantity_sold)
from sales
where trunc(sales.date_time_of_sale)
not in
(select holiday_date from holiday_map)
```

Note that the morphology (structure) of this query is completely different from the one asking for sales on holidays.

Suppose now that Bill is interested in unit sales just at those stores where the unit sales tended to be high overall. First Bill has to experiment to find a way to ask the database for the big-selling stores. Probably this will involve grouping the sales table by the store_id column:

```
select store_id
from sales
group by store_id
having sum(quantity_sold) > 1000
```

Now we know how to find stores that have sold more than 1000 units total, so we can add this as a subquery:

```
select sum(quantity_sold)
from sales
where store_id in
(select store_id
 from sales
 group by store_id
 having sum(quantity_sold) > 1000)
```

Morphologically this doesn't look very different from the preceding non-holiday query. Bill has had to figure out how to use the GROUP BY and HAVING constructs but otherwise it is a single table query with a subquery. Think about the time to execute, however. The sales table may contain millions of rows. The holiday_map table probably only

contains 50 or 100 rows, depending on how long the OLTP system has been in place. The most obvious way to execute these subqueries will be to perform the subquery for each row examined by the main query. In the case of the "big stores" query, the subquery requires scanning and sorting the entire `sales` table. So the time to execute this query might be 10,000 times longer than the time to execute the "non-holiday sales" query. Should Bill Novice expect this behavior? Should he have to think about it? Should the OLTP system grind to a halt because he didn't think about it hard enough?

Virtually all the organizations that start by trying to increase similarity and predictability among decision support queries end up with a *dimensional data warehouse*. This necessitates a new data model that shares little with the OLTP data model.

Dimensional Data Modeling: First Steps

Dimensional data modeling starts with a *fact table*. This is where we record what happened, e.g., someone bought a Diet Coke in East Fishkill. What you want in the fact table are facts about the sale, ideally ones that are numeric, continuously valued, and additive. The last two properties are important because typical fact tables grow to a billion rows or more. People will be much happier looking at sums or averages than detail. An important decision to make is the granularity of the fact table. If Walmart doesn't care about whether or not a Diet Coke was sold at 10:31 AM or 10:33 AM, recording each sale individually in the fact table is too granular. CPU time, disk bandwidth, and disk space will be needlessly consumed. Let's aggregate all the sales of any particular product in one store on a per-day basis. So we will only have one row in the fact table recording that 200 cans of Diet Coke were sold in East Fishkill on November 30, even if those 200 cans were sold at 113 different times to 113 different customers.

```
create table sales_fact (
        sales_date      date not null,
        product_id      integer,
        store_id        integer,
        unit_sales      integer,
        dollar_sales    number
);
```

So far so good, we can pull together this table with a query JOINing the `sales`, `products`, and `product_prices` (to fill the `dollar_sales` column) tables. This JOIN will group by `product_id`, `store_id`, and the truncated `date_time_of_sale`. Constructing this query will require a professional programmer but keep in mind that this work only need be done once. The marketing experts who will be using the data warehouse will be querying from the `sales_fact` table.

In building just this one table, we've already made life easier for marketing. Suppose they want total dollar sales by product. In the OLTP data model this would have required tangling with the `product_prices` table and its different prices for the same product on different days. With the sales fact table, the query is simple:

```
select product_id, sum(dollar_sales)
from sales_fact
group by product_id
```

We have a *fact table*. In a dimensional data warehouse there will always be just one of these. All of the other tables will define the *dimensions*. Each dimension contains extra information about the facts, usually in a human-readable text string that can go directly into a report. For example, let us define the time dimension:

```
create table time_dimension (
        time_key                integer primary key,
        -- just to make it a little easier to work with; this is
        -- midnight (TRUNC) of the date in question
        oracle_date             date not null,
        day_of_week             varchar(9) not null, -- 'Monday', 'Tuesday'...
        day_number_in_month     integer not null, -- 1 to 31
        day_number_overall      integer not null, -- days from the epoch (first day is 1)
        week_number_in_year     integer not null, -- 1 to 52
        week_number_overall     integer not null, -- weeks start on Sunday
        month                   integer not null, -- 1 to 12
        month_number_overall    integer not null,
        quarter                 integer not null, -- 1 to 4
        fiscal_period           varchar(10),
        holiday_flag            char(1) default 'f' check (holiday_flag in ('t', 'f')),
        weekday_flag            char(1) default 'f' check (weekday_flag in ('t', 'f')),
```

```
        season                  varchar(50),
        event                   varchar(50)
);
```

Why is it useful to define a time dimension? If we keep the date of the sales fact as an Oracle date column, it is still just about as painful as ever to ask for holiday versus non-holiday sales. We need to know about the existence of the `holiday_map` table and how to use it. Suppose we redefine the fact table as follows:

```
create table sales_fact (
        time_key        integer not null references time_dimension,
        product_id      integer,
        store_id        integer,
        unit_sales      integer,
        dollar_sales    number
);
```

Instead of storing an Oracle date in the fact table, we're keeping an integer key pointing to an entry in the time dimension. The time dimension stores, for each day, the following information:

- whether or not the day was a holiday
- into which fiscal period this day fell
- whether or not the day was part of the "Christmas season" or not

If we want a report of sales by season, the query is straightforward:

```
select td.season, sum(f.dollar_sales)
from sales_fact f, time_dimension td
where f.time_key = td.time_key
group by td.season
```

If we want to get a report of sales by fiscal quarter or sales by day of week, the SQL is structurally identical to the above. If we want to get a report of sales by manufacturer, however, we realize that we need another dimension: *product*. Instead of storing the `product_id` that references the OLTP `products` table, much better to use a synthetic product key that references a product dimension where data from the OLTP `products`, `product_categories`, and `manufacturers` tables are aggregated.

Since we are Walmart, a multi-store chain, we will want a *stores* dimension. This table will aggregate information from the `stores` and `cities` tables in the OLTP system. Here is how we would define the stores dimension in an Oracle table:

```
create table stores_dimension (
        stores_key              integer primary key,
        name                    varchar(100),
        city                    varchar(100),
        county                  varchar(100),
        state                   varchar(100),
        zip_code                varchar(100),
        date_opened             date,
        date_remodeled          date,
        -- 'small', 'medium', 'large', or 'super'
        store_size              varchar(100),
        ...
);
```

This new dimension gives us the opportunity to compare sales for large versus small stores, for new and old ones, and for stores in different regions. We can aggregate sales by geographical region, starting at the state level and drilling down to county, city, or ZIP code. Here is how we'd query for sales by city:

```
select sd.city, sum(f.dollar_sales)
from sales_fact f, stores_dimension sd
where f.stores_key = sd.stores_key
group by sd.city
```

Dimensions can be combined. To report sales by city on a quarter-by-quarter basis, we would use the following query:

```
select sd.city, td.fiscal_period, sum(f.dollar_sales)
from sales_fact f, stores_dimension sd, time_dimension td
where f.stores_key = sd.stores_key
and f.time_key = td.time_key
group by sd.stores_key, td.fiscal_period
```

(extra SQL compared to previous query shown in bold).

The final dimension in a generic Walmart-style data warehouse is *promotion*. The marketing folks will want to know how much a price reduction boosted sales, how much of that boost was permanent, and to what extent the promoted product cannibalized sales from other products sold at the same store. Columns in the promotion dimension table would include a promotion type (coupon or sale price), full information on advertising (type of ad, name of publication, type of publication), full information on in-store display, the cost of the promotion, etc.

At this point it is worth stepping back from the details to notice that the data warehouse contains less information than the OLTP system but it can be more useful in practice because queries are easier to construct and faster to execute. Most of the art of designing a good data warehouse is in defining the dimensions. Which aspects of the day-to-day business may be condensed and treated in blocks? Which aspects of the business are interesting?

Real World Example: A Data Warehouse for Levis Strauss

In 1998, ArsDigita Corporation built a Web service as a front end to an experimental custom clothing factory operated by Levi Strauss. Users would visit our site to choose a style of khaki pants, enter their waist, inseam, height, weight, and shoe size, and finally check out with their credit card. Our server would attempt to authorize a charge on the credit card through CyberCash. The factory IT system would poll our server's Oracle database periodically so that it could start cutting pants within 10 minutes of a successfully authorized order.

The whole purpose of the factory and Web service was to test and analyze consumer reaction to this method of buying clothing. Therefore, a data warehouse was built into the project almost from the start.

We did not buy any additional hardware or software to support the data warehouse. The public Web site was supported by a mid-range Hewlett-Packard Unix server that had ample leftover capacity to run the data warehouse. We created a new "dw" Oracle user, GRANTed SELECT on the OLTP tables to the "dw" user, and wrote procedures to copy all the data from the OLTP system into a star schema of tables owned by the "dw" user. For queries, we added an IP address to the machine and ran a Web server program bound to that second IP address.

Here is how we explained our engineering decisions to our customer (Levi Strauss):

```
We employ a standard star join schema for the following reasons:

* Many relational database management systems, including Oracle 8.1,
are heavily optimized to execute queries against these schemata.

* This kind of schema has been proven to scale to the world's
largest data warehouses.
```

```
* If we hired a data warehousing nerd off the street, he or she
would have no trouble understanding our schema.

In a star join schema, there is one fact table ("we sold a pair of
khakis at 1:23 pm to Joe Smith") that references a bunch of dimension
tables.  As a general rule, if we're going to narrow our interest
based on a column, it should be in the dimension table.  I.e., if
we're only looking at sales of grey dressy fabric khakis, we should
expect to accomplish that with WHERE clauses on columns of a product
dimension table.  By contrast, if we're going to be aggregating
information with a SUM or AVG command, these data should be stored in
the columns of the fact table.  For example, the dollar amount of the
sale should be stored within the fact table.  Since we have so few
prices (essentially only one), you might think that this should go in
a dimension.  However, by keeping it in the fact table we're more
consistent with traditional data warehouses.
```

After some discussions with Levi's executives, we designed in the following dimension tables:

- **time**
  for queries comparing sales by season, quarter, or holiday
- **product**
  for queries comparing sales by color or style
- **ship to**
  for queries comparing sales by region or state
- **promotion**
  for queries aimed at determining the relationship between discounts and sales
- **consumer**
  for queries comparing sales by first-time and repeat buyers
- **user experience**
  for queries looking at returned versus exchanged versus accepted items (most useful when combined with other dimensions, e.g., was a particular color more likely to lead to an exchange request)

These dimensions allow us to answer questions such as

- In what regions of the country are pleated pants most popular? (fact table joined with the product and ship-to dimensions)
- What percentage of pants were bought with coupons and how has that varied from quarter to quarter? (fact table joined with the promotion and time dimensions)
- How many pants were sold on holidays versus non-holidays? (fact table joined with the time dimension)

The Dimension Tables

The `time_dimension` table is identical to the example given above.

```
create table time_dimension (
        time_key                integer primary key,
        -- just to make it a little easier to work with; this is
        -- midnight (TRUNC) of the date in question
        oracle_date             date not null,
        day_of_week             varchar(9) not null, -- 'Monday', 'Tuesday'...
        day_number_in_month     integer not null, -- 1 to 31
        day_number_overall      integer not null, -- days from the epoch (first day is 1)
        week_number_in_year     integer not null, -- 1 to 52
        week_number_overall     integer not null, -- weeks start on Sunday
        month                   integer not null, -- 1 to 12
        month_number_overall    integer not null,
        quarter                 integer not null, -- 1 to 4
        fiscal_period           varchar(10),
```

```
        holiday_flag            char(1) default 'f' check (holiday_flag in ('t', 'f')),
        weekday_flag            char(1) default 'f' check (weekday_flag in ('t', 'f')),
        season                  varchar(50),
        event                   varchar(50)
);
```

We populated the `time_dimension` table with a single INSERT statement. The core work is done by Oracle date formatting functions. A helper table, `integers`, is used to supply a series of numbers to add to a starting date (we picked July 1, 1998, a few days before our first real order).

```
-- Uses the integers table to drive the insertion, which just contains
-- a set of integers, from 0 to n.
-- The 'epoch' is hardcoded here as July 1, 1998.

-- d below is the Oracle date of the day we're inserting.
insert into time_dimension
(time_key, oracle_date, day_of_week, day_number_in_month,
 day_number_overall, week_number_in_year, week_number_overall,
 month, month_number_overall, quarter, weekday_flag)
select n, d, rtrim(to_char(d, 'Day')), to_char(d, 'DD'), n + 1,
       to_char(d, 'WW'),
       trunc((n + 3) / 7), -- July 1, 1998 was a Wednesday, so +3 to get the week numbers to
line up with the week
       to_char(d, 'MM'), trunc(months_between(d, '1998-07-01') + 1),
       to_char(d, 'Q'), decode(to_char(d, 'D'), '1', 'f', '7', 'f', 't')
from (select n, to_date('1998-07-01', 'YYYY-MM-DD') + n as d
      from integers);
```

Remember the Oracle date minutia that you learned in the chapter on dates. If you add a number to an Oracle date, you get another Oracle date. So adding 3 to "1998-07-01" will yield "1998-07-04".

There are several fields left to be populated that we cannot derive using Oracle date functions: season, fiscal period, holiday flag, season, event. Fiscal period depended on Levi's choice of fiscal year. The `event` column was set aside for arbitrary blocks of time that were particularly interesting to the Levi's marketing team, e.g., a sale period. In practice, it was not used.

To update the `holiday_flag` field, we used two helper tables, one for "fixed" holidays (those which occur on the same day each year), and one for "floating" holidays (those which move around).

```
create table fixed_holidays (
        month                   integer not null check (month >= 1 and month <= 12),
        day                     integer not null check (day >= 1 and day <= 31),
        name                    varchar(100) not null,
        primary key (month, day)
);

-- Specifies holidays that fall on the Nth DAY_OF_WEEK in MONTH.
-- Negative means count backwards from the end.
create table floating_holidays (
        month                   integer not null check (month >= 1 and month <= 12),
        day_of_week             varchar(9) not null,
        nth                     integer not null,
        name                    varchar(100) not null,
        primary key (month, day_of_week, nth)
);
```

Some example holidays:

```
insert into fixed_holidays (name, month, day)
   values ('New Year''s Day', 1, 1);
insert into fixed_holidays (name, month, day)
```

```
   values ('Christmas', 12, 25);
insert into fixed_holidays (name, month, day)
   values ('Veteran''s Day', 11, 11);
insert into fixed_holidays (name, month, day)
   values ('Independence Day', 7, 4);

insert into floating_holidays (month, day_of_week, nth, name)
   values (1, 'Monday', 3, 'Martin Luther King Day');
insert into floating_holidays (month, day_of_week, nth, name)
   values (10, 'Monday', 2, 'Columbus Day');
insert into floating_holidays (month, day_of_week, nth, name)
   values (11, 'Thursday', 4, 'Thanksgiving');
insert into floating_holidays (month, day_of_week, nth, name)
   values (2, 'Monday', 3, 'President''s Day');
insert into floating_holidays (month, day_of_week, nth, name)
   values (9, 'Monday', 1, 'Labor Day');
insert into floating_holidays (month, day_of_week, nth, name)
   values (5, 'Monday', -1, 'Memorial Day');
```

An extremely clever person who'd recently read *SQL for Smarties* would probably be able to come up with an SQL statement to update the `holiday_flag` in the `time_dimension` rows. However, there is no need to work your brain that hard. Recall that Oracle includes two procedural languages, Java and PL/SQL. You can implement the following pseudocode in the procedural language of your choice:

```
foreach row in "select name, month, day from fixed_holidays"
    update time_dimension
      set holiday_flag = 't'
      where month = row.month and day_number_in_month = row.day;
end foreach

foreach row in "select month, day_of_week, nth, name from floating_holidays"
    if row.nth > 0 then
        # If nth is positive, put together a date range constraint
        # to pick out the right week.
        ending_day_of_month := row.nth * 7
        starting_day_of_month := ending_day_of_month - 6

        update time_dimension
          set holiday_flag = 't'
          where month = row.month
            and day_of_week = row.day_of_week
            and starting_day_of_month <= day_number_in_month
            and day_number_in_month <= ending_day_of_month;
    else
        # If it is negative, get all the available dates
        # and get the nth one from the end.
        i := 0;
        foreach row2 in "select day_number_in_month from time_dimension
                         where month = row.month
                           and day_of_week = row.day_of_week
                         order by day_number_in_month desc"
            i := i - 1;
            if i = row.nth then
                update time_dimension
                  set holiday_flag = 't'
                  where month = row.month
                    and day_number_in_month = row2.day_number_in_month
                break;
            end if
        end foreach
    end if
end foreach
```

The product dimension

The product dimension contains one row for each unique combination of color, style, cuffs, pleats, etc.

```
create table product_dimension (
        product_key     integer primary key,
        -- right now this will always be "ikhakis"
        product_type    varchar(20) not null,
        -- could be "men", "women", "kids", "unisex adults"
        expected_consumers      varchar(20),
        color           varchar(20),
        -- "dressy" or "casual"
        fabric          varchar(20),
        -- "cuffed" or "hemmed" for pants
        -- null for stuff where it doesn't matter
        cuff_state      varchar(20),
        -- "pleated" or "plain front" for pants
        pleat_state     varchar(20)
);
```

To populate this dimension, we created a one-column table for each field in the dimension table and use a multi-table join without a WHERE clause. This generates the cartesian product of all the possible values for each field:

```
create table t1 (expected_consumers varchar(20));
create table t2 (color varchar(20));
create table t3 (fabric varchar(20));
create table t4 (cuff_state varchar(20));
create table t5 (pleat_state varchar(20));

insert into t1 values ('men');
insert into t1 values ('women');
insert into t1 values ('kids');
insert into t1 values ('unisex');
insert into t1 values ('adults');
[etc.]

insert into product_dimension
(product_key, product_type, expected_consumers,
color, fabric, cuff_state, pleat_state)
select
  product_key_sequence.nextval,
  'ikhakis',
  t1.expected_consumers,
  t2.color,
  t3.fabric,
  t4.cuff_state,
  t5.pleat_state
from t1,t2,t3,t4,t5;
```

Notice that an Oracle sequence, `product_key_sequence`, is used to generate unique integer keys for each row as it is inserted into the dimension.

The promotion dimension

The art of building the promotion dimension is dividing the world of coupons into a broad categories, e.g., "between 10 and 20 dollars". This categorization depended on the learning that the marketing executives did not care about the difference between a $3.50 and a $3.75 coupon.

```
create table promotion_dimension (
        promotion_key           integer primary key,
        -- can be "coupon" or "no coupon"
        coupon_state            varchar(20),
        -- a text string such as "under $10"
        coupon_range            varchar(20)
);
```

The separate `coupon_state` and `coupon_range` columns allow for reporting of sales figures broken down into fullprice/discounted or into a bunch of rows, one for each range of coupon size.

The consumer dimension

We did not have access to a lot of demographic data about our customers. We did not have a lot of history since this was a new service. Consequently, our consumer dimension is extremely simple. It is used to record whether or not a sale in the fact table was to a new or a repeat customer.

```
create table consumer_dimension (
        consumer_key            integer primary key,
        -- 'new customer' or 'repeat customer'
        repeat_class            varchar(20)
);
```

The user experience dimension

If we are interested in building a report of the average amount of time spent contemplating a purchase versus whether the purchase was ultimately kept, the `user_experience_dimension` table will help.

```
create table user_experience_dimension (
        user_experience_key     integer primary key,
        -- 'shipped on time', 'shipped late'
        on_time_status          varchar(20),
        -- 'kept', 'returned for exchange', 'returned for refund'
        returned_status         varchar(30)
);
```

The ship-to dimension

Classically one of the most powerful dimensions in a data warehouse, our `ship_to_dimension` table allows us to group sales by region or state.

```
create table ship_to_dimension (
        ship_to_key     integer primary key,
        -- e.g., Northeast
        ship_to_region  varchar(30) not null,
        ship_to_state   char(2) not null
);

create table state_regions (
        state           char(2) not null primary key,
        region          varchar(50) not null
);

-- to populate:
insert into ship_to_dimension
(ship_to_key, ship_to_region, ship_to_state)
select ship_to_key_sequence.nextval, region, state
from state_regions;
```

Notice that we've thrown out an awful lot of detail here. Had this been a full-scale product for Levi Strauss, they would probably have wanted at least extra columns for county, city, and zip code. These columns would allow a regional sales manager to look at sales within a state.

(In a data warehouse for a manufacturing wholesaler, the ship-to dimension would contain columns for the customer's company name, the division of the customer's company that received the items, the sales district of the salesperson who sold the order, etc.)

The Fact Table

The granularity of our fact table is one order. This is finer-grained than the canonical Walmart-style data warehouse as presented above, where a fact is the quantity of a particular SKU sold in one store on one day (i.e., all orders in one day for the same item are aggregated). We decided that we could afford this because the conventional wisdom in the data warehousing business in 1998 was that up to billion-row fact tables were manageable. Our retail price was $40 and it was tough to foresee a time when the factory could make more than 1,000 pants per day. So it did not seem extravagant to budget one row per order.

Given the experimental nature of this project we did not delude ourselves into thinking that we would get it right the first time. Since we were recording one row per order we were able to cheat by including pointers from the data warehouse back into the OLTP database: `order_id` and `consumer_id`. We never had to use these but it was nice to know that if we couldn't get a needed answer for the marketing executives the price would have been some custom SQL coding rather than rebuilding the entire data warehouse.

```
create table sales_fact (
        -- keys over to the OLTP production database
        order_id                integer primary key,
        consumer_id             integer not null,
        time_key                not null references time_dimension,
        product_key             not null references product_dimension,
        promotion_key           not null references promotion_dimension,
        consumer_key            not null references consumer_dimension,
        user_experience_key     not null references user_experience_dimension,
        ship_to_key             not null references ship_to_dimension,
        -- time stuff
        minutes_login_to_order          number,
        days_first_invite_to_order      number,
        days_order_to_shipment          number,
        -- this will be NULL normally (unless order was returned)
        days_shipment_to_intent         number,
        pants_id                integer,
        price_charged           number,
        tax_charged             number,
        shipping_charged        number
);
```

After defining the fact table, we populated it with a single insert statement:

```
-- find_product, find_promotion, find_consumer, and find_user_experience
-- are PL/SQL procedures that return the appropriate key from the dimension
-- tables for a given set of parameters

insert into sales_fact
 select o.order_id, o.consumer_id, td.time_key,
        find_product(o.color, o.casual_p, o.cuff_p, o.pleat_p),
        find_promotion(o.coupon_id),
        find_consumer(o.pants_id),
        find_user_experience(o.order_state, o.confirmed_date, o.shipped_date),
        std.ship_to_key,
        minutes_login_to_order(o.order_id, usom.user_session_id),
        decode(sign(o.confirmed_date - gt.issue_date), -1, null, round(o.confirmed_date -
gt.issue_date, 6)),
        round(o.shipped_date - o.confirmed_date, 6),
        round(o.intent_date - o.shipped_date, 6),
        o.pants_id, o.price_charged, o.tax_charged, o.shipping_charged
 from khaki.reportable_orders o, ship_to_dimension std,
      khaki.user_session_order_map usom, time_dimension td,
      khaki.addresses a, khaki.golden_tickets gt
 where o.shipping = a.address_id
        and std.ship_to_state = a.usps_abbrev
        and o.order_id = usom.order_id(+)
        and trunc(o.confirmed_date) = td.oracle_date
        and o.consumer_id = gt.consumer_id;
```

As noted in the comment at top, most of the work here is done by PL/SQL procedures such as `find_product` that dig up the right row in a dimension table for this particular order.

The preceding insert will load an empty data warehouse from the on-line transaction processing system's tables. Keeping the data warehouse up to date with what is happening in OLTP land requires a similar INSERT with an extra restriction WHERE clause limiting orders to only those order ID is larger than the maximum of the order IDs currently in the warehouse. This is a safe transaction to execute as many times per day as necessary--even two simultaneous INSERTs would not corrupt the data warehouse with duplicate rows because of the primary key constraint on `order_id`. A daily update is traditional in the data warehousing world so we scheduled one every 24 hours using the Oracle `dbms_job` package (http://www.oradoc.com/ora816/server.816/a76956/jobq.htm#750).

Sample Queries

We have (1) defined a star schema, (2) populated the dimension tables, (3) loaded the fact table, and (4) arranged for periodic updating of the fact table. Now we can proceed to the interesting part of our data warehouse: getting information back out.

Using only the `sales_fact` table, we can ask for

- the total number of orders, total revenue to date, tax paid, shipping costs to date, the average price paid for each item sold, and the average number of days to ship:

```
select count(*) as n_orders,
       round(sum(price_charged)) as total_revenue,
       round(sum(tax_charged)) as total_tax,
       round(sum(shipping_charged)) as total_shipping,
       round(avg(price_charged),2) as avg_price,
       round(avg(days_order_to_shipment),2) as avg_days_to_ship
from sales_fact;
```

- the average number of minutes from login to order (we exclude user sessions longer than 30 minutes to avoid skewing the results from people who interrupted their shopping session to go out to lunch or sleep for a few hours):

```
select round(avg(minutes_login_to_order), 2)
from sales_fact
where minutes_login_to_order < 30
```

- the average number of days from first being invited to the site by email to the first order (excluding periods longer than 2 weeks to remove outliers):

```
select round(avg(days_first_invite_to_order), 2)
from sales_fact
where days_first_invite_to_order < 14
```

Joining against the `ship_to_dimension` table lets us ask how many pants were shipped to each region of the United States:

```
select ship_to_region, count(*) as n_pants
from sales_fact f, ship_to_dimension s
where f.ship_to_key = s.ship_to_key
group by ship_to_region
order by n_pants desc
```

Region	Pants Sold
New England Region	612
NY and NJ Region	321
Mid Atlantic Region	318
Western Region	288
Southeast Region	282
Southern Region	193

Great Lakes Region	177
Northwestern Region	159
Central Region	134
North Central Region	121

Note: these data are based on a random subset of orders from the Levi's site and we have also made manual changes to the report values. The numbers are here to give you an idea of what these queries do, not to provide insight into the Levi's custom clothing business.

Joining against the `time_dimension`, we can ask how many pants were sold for each day of the week:

```
select day_of_week, count(*) as n_pants
from sales_fact f, time_dimension t
where f.time_key = t.time_key
group by day_of_week
order by n_pants desc
```

Day of Week	Pants Sold
Thursday	3428
Wednesday	2823
Tuesday	2780
Monday	2571
Friday	2499
Saturday	1165
Sunday	814

We were able to make pants with either a "dressy" or "casual" fabric. Joining against the `product_dimension` table can tell us how popular each option was as a function of color:

```
select color, count(*) as n_pants, sum(decode(fabric,'dressy',1,0)) as n_dressy
from sales_fact f, product_dimension p
where f.product_key = p.product_key
group by color
order by n_pants desc
```

Color	Pants Sold	% Dressy
dark tan	486	100
light tan	305	49
dark grey	243	100
black	225	97
navy blue	218	61
medium tan	209	0
olive green	179	63

Note: 100% and 0% indicate that those colors were available only in one fabric.

Here is a good case of how the data warehouse may lead to a practical result. If these were the real numbers from the Levi's warehouse, what would pop out at the manufacturing guys is that 97% of the black pants sold were in one fabric style. It might not make sense to keep an inventory of casual black fabric if there is so little consumer demand for it.

Query Generation: The Commercial Closed-Source Route

The promise of a data warehouse is not fulfilled if all users must learn SQL syntax and how to run SQL*PLUS. From being exposed to 10 years of advertising for query tools, we decided that the state of forms-based query tools must be truly advanced. We thus suggested to Levi Strauss that they use Seagate Crystal Reports and Crystal Info to analyze their data. These packaged tools, however, ended up not fitting very well with what Levi's wanted to accomplish. First, constructing queries was not semantically simpler than coding SQL. The Crystal Reports consultant that we brought in said that most of his clients ended up having a programmer set up the report queries and the business people would simply run the report every day against new data. If professional programmers had to construct queries, it seemed just as easy just to write more admin pages using our standard Web development tools, which required about 15 minutes per page. Second, it was impossible to ensure availability of data warehouse

queries to authorized users anywhere on the Internet. Finally there were security and social issues associated with allowing a SQL*Net connection from a Windows machine running Crystal Reports out through the Levi's firewall to our Oracle data warehouse on the Web.

Not knowing if any other commercial product would work better and not wanting to disappoint our customer, we extended the ArsDigita Community System with a data warehouse query module that runs as a Web-only tool. This is a free open-source system and comes with the standard ACS package that you can download from http://www.arsdigita.com/download/.

Query Generation: The Open-Source ACS Route

The "dw" module in the ArsDigita Community System is designed with the following goals:

1. naive users can build simple queries by themselves
2. professional programmers can step in to help out the naive users
3. a user with no skill can re-execute a saved query

We keep one row per query in the `queries` table:

```
create table queries (
        query_id        integer primary key,
        query_name      varchar(100) not null,
        query_owner     not null references users,
        definition_time date not null,
        -- if this is non-null, we just forget about all the query_columns
        -- stuff; the user has hand-edited the SQL
        query_sql       varchar(4000)
);
```

Unless the `query_sql` column is populated with a hand-edited query, the query will be built up by looking at several rows in the `query_columns` table:

```
-- this specifies the columns we we will be using in a query and
-- what to do with each one, e.g., "select_and_group_by" or
-- "select_and_aggregate"

-- "restrict_by" is tricky; value1 contains the restriction value, e.g., '40'
-- or 'MA' and value2 contains the SQL comparion operator, e.g., "=" or ">"

create table query_columns (
        query_id        not null references queries,
        column_name     varchar(30),
        pretty_name     varchar(50),
        what_to_do      varchar(30),
        -- meaning depends on value of what_to_do
        value1          varchar(4000),
        value2          varchar(4000)
);

create index query_columns_idx on query_columns(query_id);
```

The `query_columns` definition appears strange at first. It specifies the name of a column but not a table. This module is predicated on the simplifying assumption that we have one enormous view, `ad_hoc_query_view`, that contains all the dimension tables' columns alongside the fact table's columns.

Here is how we create the view for the Levi's data warehouse:

```
create or replace view ad_hoc_query_view
as
select minutes_login_to_order, days_first_invite_to_order,
       days_order_to_shipment, days_shipment_to_intent, pants_id,
       price_charged, tax_charged, shipping_charged,
       oracle_date, day_of_week,
```

```
       day_number_in_month, week_number_in_year, week_number_overall,
       month, month_number_overall, quarter, fiscal_period,
       holiday_flag, weekday_flag, season, color, fabric, cuff_state,
       pleat_state, coupon_state, coupon_range, repeat_class,
       on_time_status, returned_status, ship_to_region, ship_to_state
from sales_fact f, time_dimension t, product_dimension p,
     promotion_dimension pr, consumer_dimension c,
     user_experience_dimension u, ship_to_dimension s
where f.time_key = t.time_key
and f.product_key = p.product_key
and f.promotion_key = pr.promotion_key
and f.consumer_key = c.consumer_key
and f.user_experience_key = u.user_experience_key
and f.ship_to_key = s.ship_to_key;
```

At first glance, this looks like a passport to sluggish Oracle performance. We'll be doing a seven-way JOIN for every data warehouse query, regardless of whether we need information from some of the dimension tables or not.

We can test this assumption as follows:

```
-- tell SQL*Plus to turn on query tracing
set autotrace on

-- let's look at how many pants of each color
-- were sold in each region

SELECT ship_to_region, color, count(pants_id)
FROM ad_hoc_query_view
GROUP BY ship_to_region, color;
```

Oracle will return the query results first...

ship_to_region	color	count(pants_id)
Central Region	black	46
Central Region	dark grey	23
Central Region	dark tan	39
..		
Western Region	medium tan	223
Western Region	navy blue	245
Western Region	olive green	212

... and then explain how those results were obtained:

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=181 Card=15 Bytes=2430)
   1    0   SORT (GROUP BY) (Cost=181 Card=15 Bytes=2430)
   2    1     NESTED LOOPS (Cost=12 Card=2894 Bytes=468828)
   3    2       HASH JOIN (Cost=12 Card=885 Bytes=131865)
   4    3         TABLE ACCESS (FULL) OF 'PRODUCT_DIMENSION' (Cost=1 Card=336 Bytes=8400)
   5    3         HASH JOIN (Cost=6 Card=885 Bytes=109740)
   6    5           TABLE ACCESS (FULL) OF 'SHIP_TO_DIMENSION' (Cost=1 Card=55 Bytes=1485)
   7    5           NESTED LOOPS (Cost=3 Card=885 Bytes=85845)
   8    7             NESTED LOOPS (Cost=3 Card=1079 Bytes=90636)
   9    8               NESTED LOOPS (Cost=3 Card=1316 Bytes=93436)
  10    9                 TABLE ACCESS (FULL) OF 'SALES_FACT' (Cost=3 Card=1605 Bytes=93090)
  11    9                 INDEX (UNIQUE SCAN) OF 'SYS_C0016416' (UNIQUE)
  12    8               INDEX (UNIQUE SCAN) OF 'SYS_C0016394' (UNIQUE)
  13    7             INDEX (UNIQUE SCAN) OF 'SYS_C0016450' (UNIQUE)
  14    2       INDEX (UNIQUE SCAN) OF 'SYS_C0016447' (UNIQUE)
```

As you can see from the table names in bold face, Oracle was smart enough to examine only tables relevant to our query: `product_dimension`, because we asked about color; `ship_to_dimension`, because we asked about region; `sales_fact`, because we asked for a count of pants sold. Bottom line: Oracle did a 3-way JOIN instead of the 7-way JOIN specified by the view.

To generate a SQL query into `ad_hoc_query_view` from the information stored in `query_columns` is most easily done with a function in a procedural language such as Java, PL/SQL, Perl, or Tcl (here is pseudocode):

```
proc generate_sql_for_query(a_query_id)
    select_list_items list;
    group_by_items list;
    order_clauses list;

    foreach row in "select column_name, pretty_name
                    from query_columns
                    where query_id = a_query_id
                      and what_to_do = 'select_and_group_by'"]
        if row.pretty_name is null then
            append_to_list(group_by_items, row.column_name)
        else
            append_to_list(group_by_items, row.column_name || ' as "' || row.pretty_name ||
'"'
        end if
    end foreach

    foreach row in "select column_name, pretty_name, value1
                    from query_columns
                    where query_id = a_query_id
                      and what_to_do = 'select_and_aggregate'"
         if row.pretty_name is null then
            append_to_list(select_list_items, row.value1 || row.column_name)
         else
            append_to_list(select_list_items, row.value1 || row.column_name || ' as "' ||
row.pretty_name || '"'
         end if
    end foreach

    foreach row in "select column_name, value1, value2
                    from query_columns
                    where query_id = a_query_id
                      and what_to_do = 'restrict_by'"
        append_to_list(where_clauses, row.column_name || ' ' || row.value2 || ' ' ||
row.value1)
    end foreach

    foreach row in "select column_name
                    from query_columns
                    where query_id = a_query_id
                      and what_to_do = 'order_by'"]
        append_to_list(order_clauses, row.column_name)
    end foreach

    sql := "SELECT " || join(select_list_items, ', ') ||
           " FROM ad_hoc_query_view"

    if list_length(where_clauses) > 0 then
        append(sql, ' WHERE ' || join(where_clauses, ' AND '))
    end if

    if list_length(group_by_items) > 0 then
        append(sql, ' GROUP BY ' || join(group_by_items, ', '))
    end if

    if list_length(order_clauses) > 0 then
        append(sql, ' ORDER BY ' || join(order_clauses, ', '))
    end if
```

```
    return sql
end proc
```

How well does this work in practice? Suppose that we were going to run regional advertisements. Should the models be pictured where pleated or plain front pants? We need to look at recent sales by region. With the ACS query tool, a user can use HTML forms to specify the following:

- pants_id : select and aggregate using count
- ship_to_region : select and group by
- pleat_state : select and group by

The preceding pseudocode turns that into

```
SELECT ship_to_region, pleat_state, count(pants_id)
FROM ad_hoc_query_view
GROUP BY ship_to_region, pleat_state
```

which is going to report sales going back to the dawn of time. If we weren't clever enough to anticipate the need for time windowing in our forms-based interface, the "hand edit the SQL" option will save us. A professional programmer can be grabbed for a few minutes to add

```
SELECT ship_to_region, pleat_state, count(pants_id)
FROM ad_hoc_query_view
WHERE oracle_date > sysdate - 45
GROUP BY ship_to_region, pleat_state
```

Now we're limiting results to the last 45 days:

ship_to_region	pleat_state	count(pants_id)
Central Region	plain front	8
Central Region	pleated	26
Great Lakes Region	plain front	14
Great Lakes Region	pleated	63
Mid Atlantic Region	plain front	56
Mid Atlantic Region	pleated	162
NY and NJ Region	plain front	62
NY and NJ Region	pleated	159
New England Region	plain front	173
New England Region	pleated	339
North Central Region	plain front	7
North Central Region	pleated	14
Northwestern Region	plain front	20
Northwestern Region	pleated	39
Southeast Region	plain front	51
Southeast Region	pleated	131
Southern Region	plain front	13
Southern Region	pleated	80
Western Region	plain front	68
Western Region	pleated	120

If we strain our eyes and brains a bit, we can see that plain front pants are very unpopular in the Great Lakes and South but more popular in New England and the West. It would be nicer to see percentages within region, but standard SQL does not make it possible to combine results to values in surrounding rows. We will need to refer to the "SQL for Analysis" chapter in the Oracle data warehousing documents to read up on extensions to SQL that makes this possible:

```
SELECT
  ship_to_region,
  pleat_state,
  count(pants_id),
  ratio_to_report(count(pants_id))
       over (partition by ship_to_region) as percent_in_region
FROM ad_hoc_query_view
WHERE oracle_date > sysdate - 45
GROUP BY ship_to_region, pleat_state
```

We're asked Oracle to window the results ("partition by ship_to_region") and compare the number of pants in each row to the sum across all the rows within a regional group. Here's the result:

ship_to_region	pleat_state	count(pants_id)	percent_in_region
...			
Great Lakes Region	plain front	14	.181818182
Great Lakes Region	pleated	63	.818181818
...			
New England Region	plain front	173	.337890625
New England Region	pleated	339	.662109375
...			

This isn't quite what we want. The "percents" are fractions of 1 and reported with far too much precision. We tried inserting the Oracle built-in `round` function in various places of this SQL statement but all we got for our troubles was "ERROR at line 5: ORA-30484: missing window specification for this function". We had to add an extra layer of SELECT, a view-on-the-fly, to get the report that we wanted:

```
select ship_to_region, pleat_state, n_pants, round(percent_in_region*100)
from
(SELECT
   ship_to_region,
   pleat_state,
   count(pants_id) as n_pants,
   ratio_to_report(count(pants_id))
        over (partition by ship_to_region) as percent_in_region
 FROM ad_hoc_query_view
 WHERE oracle_date > sysdate - 45
 GROUP BY ship_to_region, pleat_state)
```

returns

ship_to_region	pleat_state	count(pants_id)	percent_in_region
...			
Great Lakes Region	plain front	14	18
Great Lakes Region	pleated	63	82
...			
New England Region	plain front	173	34
New England Region	pleated	339	66
...			

What if you're in charge of the project?

If you are in charge of a data warehousing project, you need to assemble the necessary tools. Do not be daunted by this prospect. The entire Levi Strauss system described above was implemented in three days by two programmers.

The first tool that you need is intelligence and thought. If you pick the right dimensions and put the required data into them, your data warehouse will be useful. If you don't get your dimensions right, you won't even be able to ask the interesting questions. If you're not smart or thoughtful, probably the best thing to do is find a boutique consulting firm with expertise in building data warehouses for your industry. Get them to lay out the initial star schema. They won't get it right but it should be close enough to live with for a few months. If you can't find an expert, *The Data Warehouse Toolkit* (Ralph Kimball 1996) contains example schemata for 10 different kinds of businesses.

You will need some place to store your data and query parts back out. Since you are using SQL your only choice is a relational database management system. There are specialty vendors that have historically made RDBMSes with enhanced features for data warehousing, such as the ability to compute a value based on information from the current row compared to information from a previously output row of the report. This gets away from the strict unordered set-theoretic way of looking at the world that E.F. Codd sketched in 1970 but has

proven to be useful. Starting with version 8.1.6, Oracle has added most of the useful third-party features into their standard product. Thus all but the very smallest and very largest modern data warehouses tend to be built using Oracle (see the "SQL for Analysis" chapter in the *Oracle8i Data Warehousing Guide* volume of the Oracle documentation).

Oracle contains two features that may enable you to construct and use your data warehouse without investing in separate hardware. First is the optimistic locking system that Oracle has employed since the late 1980s. If someone is doing a complex query it will not affect transactions that need to update the same tables. Essentially each query runs in its own snapshot of the database as it existed when the query was started. The second Oracle feature is *materialized views* or *summaries*. It is possible to instruct the database to keep a summary of sales by quarter, for example. If someone asks for a query involving quarterly sales, the small summary table will be consulted instead of the comprehensive sales table. This could be 100 to 1000 times faster.

One typical goal of a data warehousing project is to provide a unified view of a company's disparate information systems. The only way to do this is to extract data from all of these information systems and clean up those data for consistency and accuracy. This is purportedly a challenging task when RDBMSes from different vendors are involved, though it might not seem so on the surface. After all, every RDBMS comes with a C library. You could write a C program to perform queries on the Brand X database and do inserts on the Brand Y database. Perl and Tcl have convenient facilities for transforming text strings and there are db connectivity interfaces from these scripting languages to DBMS C libraries. So you could write a Perl script. Most databases within a firm are accessible via the Web, at least within a company's internal network. Oracle includes a Java virtual machine and Java libraries to fetch Web pages and parse XML. So you could write a Java or PL/SQL program running inside your data warehouse Oracle installation to grab the foreign information and bring it back (see the chapter on foreign and legacy data).

If you don't like to program or have a particularly knotty connectivity problem involving an old mainframe, various companies make software that can help. For high-end mainframe stuff, Oracle Corporation itself offers some useful layered products. For low-end "more-convenient-than-Perl" stuff, Data Junction (www.datajunction.com) is useful.

Given an already-built data warehouse, there are a variety of useful query tools. The theory is that if you've organized your data model well enough, a non-technical user will be able to navigate around via a graphic user interface or a Web browser. The best known query tool is Crystal Reports (www.seagatesoftware.com), which we tried to use in the Levi Strauss example. See http://www.arsdigita.com/doc/dw for details on the free open-source ArsDigita Community System data warehouse query module.

Is there a bottom line to all of this? If you can think sufficiently clearly about your organization and its business to construct the correct dimensions and program SQL reasonably well, you will be successful with the raw RDBMS alone. Extra software tools can potentially make the project a bit less painful or a bit shorter but they won't be of critical importance.

More Information

The construction of data warehouses is a guild-like activity. Most of the expert knowledge is contained within firms that specialize not in data warehousing but in data warehousing for a particular kind of company. For example, there are firms that do nothing but build data warehouses for supermarkets. There are firms that do nothing but build data warehouses for department stores. Part of what keeps this a tight guild is the poor quality of textbooks and journal articles on the subject. Most of the books on data warehousing are written by and for people who do not know SQL. The books focus on (1) stuff that you can buy from a vendor, (2) stuff that you can do from a graphical user interface after the data warehouse is complete, and (3) how to navigate around a large organization to get all the other suits to agree to give you their data, their money, and a luxurious schedule.

The only worthwhile introductory book that we've found on data warehousing in general is Ralph Kimball's *The Data Warehouse Toolkit*. Kimball is also the author of an inspiring book on clickstream data warehousing: *The Data Webhouse Toolkit*. The latter book is good if you are interested in applying classical dimensional data warehousing techniques to user activity analysis.

It isn't exactly a book and it isn't great for beginners but the *Oracle8i Data Warehousing Guide* volume of the official Oracle server documentation is extremely useful.

Data on consumer purchasing behavior are available from A.C. Nielsen (www.acnielsen.com), Information Resources Incorporated (IRI; www.infores.com), and a bunch of other companies listed in http://dir.yahoo.com/Business_and_Economy/Business_to_Business/Marketing_and_Advertising/Market_Resear

Reference

Next: Foreign and Legacy Data

---

philg@mit.edu

Reader's Comments

> I really like that Walmart/Sybase example, because Walmart is actually running the largest commercial data warehouse in the world including 2 years of detail data with tens of billions of detail rows. Of course, it's not using an OLTP system like Sybase/Oracle, it's a decision support database, Teradata.
>
> -- Dieter Noeth, May 14, 2003

> I would dispute that: http://www.wintercorp.com/vldb/2003_TopTen_Survey/TopTenWinners.asp Shows that France Telecom has the largest DSS system in Oracle. Walmart is not in the top-ten list, and surprise, surprise, the squashed competitor Kmart is.
>
> -- what ever, January 7, 2004

> Your comments about Sybase are naive and incorrect. Perhaps you had a bad run dealing with Sybase support, not sure if I can influence it otherwise. Sybase IQ for years now has been the bane some of the World's Largest data warehouses. Query performance and scalability are notably the highlights of all Sybase IQ implementations. Sybase customer, comScore Networks, received the Grand Prize in the 2003 Winter Corporation TopTen Program for Largest Database Size and Most Rows/Records for Microsoft Windows-based systems using Sybase IQ, the highly scalable analytics engine. Other Winter Corporation TopTen award-winning Sybase customers in UNIX categories include Nielsen Media Research and Korean-based customers Health Insurance Review Agency (HIRA), LG Card, Samsung Card and Chohung Bank. http://www.sybase.be/belgium/press/20031111-ipg-IQwintercorp.jsp I appreciate that you have given an opportunity to comment on this page.
>
> -- Subraya Pai, May 9, 2004

> Crystal Reports is not the reporting tool usually choosen for ad-hoc querying of datawarehouses, so maybe that's the reason your end customers weren't very happy about it. Tools better suited to this task are BusinessObjects (who also aquired CrystalReports a couple of month ago), Brio (aquired by Hyperion about 1 year ago), Microstrategy, Oracle Discoverer and Cognos. All of them allow you to build metadata about the datamodel of the datawarehouse and present the end user with the world in terms known to them (no criptic database table column names, predefined filter conditions and so on). For the end-user it's really only a matter of dragging and dropping the "objects" in their report and "pressing" a button. The tool will the generate the

proper SQL, query the database (some of them even rewrite the query if you have aggregate tables, allow you to "join" in the report query results from different queries and databases or take a stored procedure as the data source) allow you to do simple computations (excel-like) on the result set etc.

Regards, Georg

-- Georg Breazu, November 15, 2004

this article considered as a good article in data warehousing.

I want to say that there is a query language called multidimensional Expression MDX .It is now the standard query language for OLAP.it is like Sql but with more capabilities in Grouping and more functions to facilitate OLAP operations.The most common function are ROLLUP and CUBE>>>>>>>>>>>>>>>>>>>>>>>>>>>>etc. thanks to all.....

-- drtech dr, December 22, 2008

good n useful article..tnx for sharing..

-- mitesh trivedi, February 9, 2010

Add a comment

Related Links

- Optimize Data Warehouse- Optimize performance and usability of large data warehouses using parallelism, partitions, and aggregate summaries across the complete application stack comprising of ETL, databases, and reporting. (contributed by Jag Singh)
- DBMS 2 on data warehousing- Data warehouse technology, developments, and trends, from what is now the industry-leading source of database management news and analysis. (contributed by Curt Monash)

Add a link