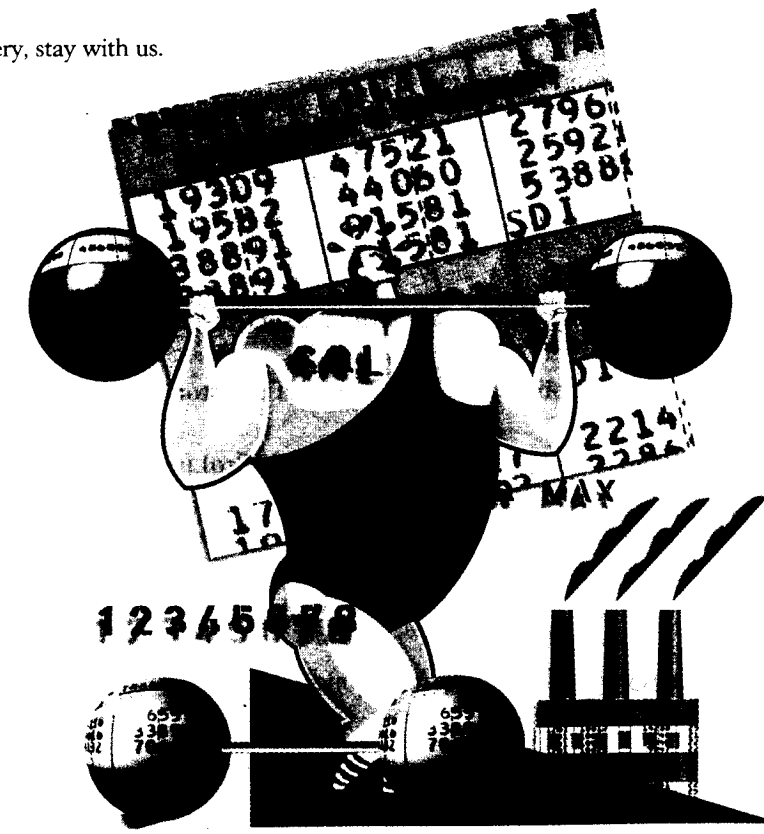
► A data model optimized for transactions and a language that makes business questions hard to answer cause decision support systems to flounder. The answer: dimensional modeling and new SQL functions. *By Ralph Kimball and Kevin Strehlo*

Why Decision Support FAILS and How To FIX It

FORGET EVERYTHING you know about entity relationship data modeling. If your DBA is proud of normalizing all your databases into Fifth Normal Form, tell her to read this article. Or if all you know is that it's difficult to get answers to simple business questions from your decision support system, or it takes hours to get an answer to a few-hundred-record query, stay with us.



Because the truth is that relational database management systems aren't very good at what they were supposed to do–help us get answers from our data–unless we ignore a good deal of relational dogma and use a new approach to data modeling when we're building a decision support database.

The relationally correct data modeling everyone is taught in school is only useful for achieving high performance in on-line transaction processing. The resulting model fragments the data into many tables of relatively equal width and depth to speed transactions along. But using that model with a real-world decision support system almost guarantees failure.

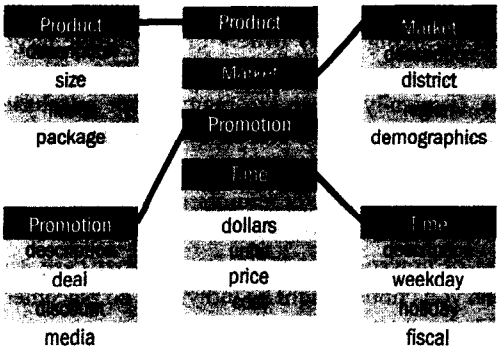
Take the case of a mid-size durable goods manufacturer, one of the smokestacks on the horizon. After a year's effort, IS staffers had a database design, had loaded several hundred megabytes of data from 6 million invoices into a multiprocessor Teradata relational database machine, and users had begun to try to use the system. But there were 50 tables in their fairly typical schema. Asking the question, "Am I profitable in Schenectady?" required 14 con-

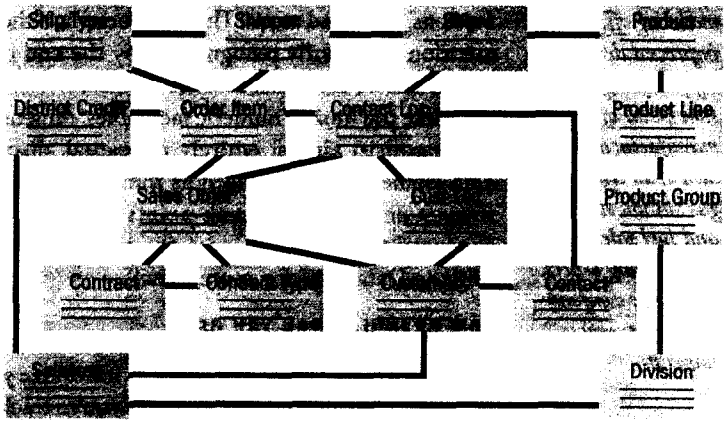
Dimensional Model Stars for Queries...

Unlike Transaction-Oriented Entity-Relation Model

Suppose that a user wanted to build a report tying contracts to products. This is a reasonable business request. In the entity-relation design, how many pathways are there between contract and product? The answer is: many. Each of these pathways will yield a different business answer because the tables are linked together through the underlying data. Although in a technical sense, each possible pathway represents a valid interpretation, it is difficult for humans or even computer software to understand and sift through all the possibilities. The result is a database design that is intimidating and unusable.

In the dimensional model, you create a database using the relevant subset of the full data and model it into large fact tables and small dimensional tables that contain only the constraints needed to produce relevant reports about contracts and products.





straints in an SQL statement that covered two 8½×11-inch single-spaced pages. The $2 million machine was maxed out at 60 queries a day, and IS professionals had to formulate all the queries.

But after the same data was reloaded into a simple five-table design, normal human beings could formulate many questions with a simple query tool, and the machine breezed through 1,000 queries a day.

The technique that saved the day is called dimensional modeling. The key to dimensional modeling is that it distinguishes between two different kinds of tables: a central fact table that is very, very large (read many, many rows) surrounded by small tables that end users will understand as representing the natural dimensions of the business.

Don't worry, there's no extra work involved. Long experience has taught IS that it's impractical to do querying against the live data in a high-volume transaction database anyway. So, while you're extracting the data from the OLTP database into a separate decision support database, recast it from its fully normalized model to a dimensional framework. No sweat. Then you perform queries against that data using an ordinary relational database management system.

Many of the most basic business questions will still go beyond the capabilities of simple query tools, requiring tortuous SQL or hand-stitching answers together in a spreadsheet or similar tool. But dimensional modeling is the only way to go if your business needs to get answers from the 50 megabytes or more of data collected from a typical business process in manufacturing, packaged goods, retailing, and practically every other field.

There are three things you need to stipulate when creating your decision support database: facts, dimensions, and granularity.

KEEP THE FACTS STRAIGHT

Fact tables and dimension tables are deliberately very different.

A fact table is the central table in the database. It consists of the numerical measurements that get entered in your database. For example, the facts might be the variable numbers in a transaction, such as a

Getting the Data Out

Defining the answer you want when using a dimensional design is an intuitive process that involves constraining the facts you want to see using the attributes found in dimension tables. Typical constraints are:

Item Dimension

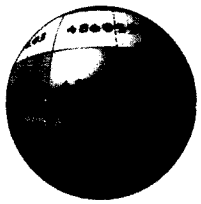
- **Items for which a specific product manager is responsible**
- **Items used by the printing industry**
- **All items except those weighing more than 500 pounds**
- **All items also sold by your East Coast competitors**

Customer Dimension

- **Nonchain customers in Southern states**
- **Customers who carry only your products**
- **New customers in the Eastern sales region**

Time Dimension

- **First fiscal quarter 1994**
- **First fiscal quarter 1995**
- **This month Father's Day sales season**



Dimensional data modeling offers improved decision support for grocery store point of sale, manufacturing shipments, ad tracking, gas station status and fuel sales, pay TV viewing orders, insurance company premiums...you name it.

single point-of-sale purchase by a customer or a line item in an invoice.

Dimension tables, in contrast, are the constraints you'll use when forming subsets of the data. A good dimension table represents a natural dimension in the underlying business, and there's no magic here: the constructs are things like product, markets, promotions, time, and so on. The good news is if the dimension is instantly recognizable by the end user, then it can be used effectively in queries.

Unlike the mostly numeric fact tables, dimension tables consist almost entirely of textual, discrete data elements that are used as the source of constraints and subtotal break points in the reports produced to answer business questions (see "Getting the Data Out" for more detail).

HOW GRANULAR CAN YOU GET?

Granularity is a choice you make to maximize your leverage of the data. For example, if you're tracking sales in a retail chain, you could choose to get answers about hourly sales of items by cash register or daily sales of items per store. Most often, at least some aggregation is required–we don't think the technology is ready for analysis at the individual-item-sold level, not even with dimensional modeling.

Even though we recommend some aggregation in most decision support databases, the fact table ends up being many orders of magnitude larger (that is, many rows deeper) than the dimension tables. Typically, there are many more purchases made than there are stores, or items for sale, or time periods in the database.

Another example of granularity from a different domain might be a database tracking the effectiveness of newspaper advertising. In that case, the grain you choose might be individual ad placement by day and by edition.

There's more to dimensional modeling than just fact tables and dimension tables. The technique makes its own use of keys, as well. Fact tables have a multipart key, which serves as the link between the fact table and the dimension tables. Each element in that multipart key points to (in the jargon of relational data design, is "a foreign key to") one of the dimension tables.

The nonkey fields in the fact table are, what else, facts. As we said, facts are measurements taken at the intersections of the dimensions of the business, i.e., the numbers that matter when you look at the intersection of time and product, or product and market, and so on.

The best facts are numeric, continuously valued, and additive. Additive facts are very powerful in decision support because any subset of them can be summed together to produce a valid result. For example, you can add quantity sold in each discrete fact to find quantity sold over some stretch of time, some product, some marketplace.

Already the alarm bells are going off in the heads of relational purists. But listen, gang, it's okay to be denormalized, even to the radical extent that we're suggesting here. From the users' point of view, the fact that some data in the dimension tables is repeated is great from a readability point of view.

OLTP and Decision Support: Oil and Water

The central accomplishment of RDBMS vendors in the last dozen years has been to elevate "getting the data in" to a level capable of supporting the largest and most demanding production systems. On-line transaction processing can now support 1,000 transactions per second under realistic conditions. Multiprocessing machines can support several times this transaction rate.

But getting the data in is not why the business world has adopted relational databases on such a massive scale. The whole promise of relational database systems has been their flexibility in getting the data out!

The central technique for achieving high-performance transaction processing is called entity-relation modeling. Entity-relation modeling seeks to root out all redundancy in the data. The result is that a typical invoice database will have at least 50, and perhaps as many as 200, separate tables describing all of these logical relationships.

That's fine for transaction processing. When each logical relationship is separately defined, any given transaction becomes extremely simple. A transaction will always update exactly one record in one very narrow, simple table. This is how OLTP has been able to achieve 1,000 transactions per second.

But the entity-relation approach to database design is a disaster for querying. Nor is there a magical GUI that can bail out the entity-relation design for querying. Graphical frosting smoothed over the underlying complexity problem does not make that complexity go away. Ultimately, the user will be faced with the myriad choices such a design represents, and this set of choices is not what the user wants to think about.

The underlying software that tries to optimize such a query is stuck with the same set of problems. Typically, when confronted with a many-table schema, involving complex joins between the desired tables, database software reverts to simplistic behavior. Some DBMSs process queries from left to right. IBM's flagship database product, DB2, actually rejects all queries involving 15 or more tables.

The only answer is dimensional data modeling.

Please note that where the data really matters–the table in the middle–the fact table is completely normalized.

SCRATCHING THE SURFACE

It's too simple? Well, that's right. All we've covered are the basics. For example, most business processes don't have one fact table. Rather, they have multiple fact tables, one for each process central to the business. For example, there might be a production system for orders, shipments, and warehousing.

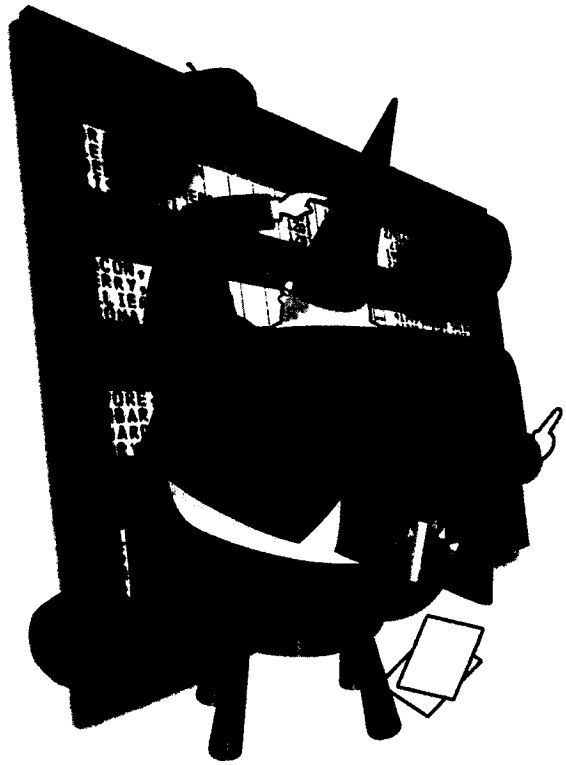
These systems cannot usually be combined into a single fact table because the timing of each transaction is different and the dimensionality of each of the systems is different. For instance, shipments usually have a carrier dimension that orders and warehousing do not share. But these multiple systems can be represented in an information warehouse by building a fact table for each one.

The multiple fact tables are called a Family because they share many of the same dimension tables. The key to a successful Family design is to make sure that, if the fact tables share a dimension, it is exactly the same dimension.

Nor are the schemas of the individual fact and dimension tables as simple as we've depicted here. But it's clear to us that the dimensional approach to data modeling is the right one. Nearly all successful decision support systems that have been implemented to date have gravitated, consciously or unconsciously, to this star-shaped design with a very large central fact table surrounded by four or five small dimensional tables. Most entity-relation designs intended for decision support, on the other hand, have failed.

The dimensional approach to the design of query-processing databases merges nicely with the recent development of a variety of workstation tools for visualizing and manipulating data within a dimensional framework. The major spreadsheet vendors, for instance, are providing "multidimensional spreadsheet views" of data.

Over the past dozen years, the relational database industry has drastically improved the effectiveness of transaction processing and has made transaction processing production-ready. It is now time for the industry to turn its attention to query processing with the goal of achieving the same drastic improvements. We have a framework and a significant amount of pioneering experience that points us in a promising direction. Our book on how to get the most from decision support, due late this year from Butterworth-Heineman, will delve more deeply into all of the issues. But we hope this simple treatment has made it clear that the future of getting the data out lies in dimensional database modeling.



SQL Is OUR Language, Fix It NOW

CONTRARY TO WHAT most of us are led to believe, SQL was not intended for business analysis. As a result, it has a number of glaring weaknesses that make simple analysis difficult or impossible.

The biggest weakness? The inability to do comparisons. Comparisons are the core technique for understanding businesses. If I say that my sales were 289 this year, I am conveying very little information. But if I say that my sales were 10 last year and 289 this year, I have provided a kind of business calibration that carries a number of important connotations.

Remarkable as it may seem, SQL makes it very difficult to express a concept as simple as this year's sales compared with last year's sales, or any equivalent formulation such as the difference or the percent change.

The SQL application programmer trying to do comparisons is faced with four choices, all of them bad. The original SQL comparison mechanism was the self join: this year's sales and last year's sales are retrieved by self-joining the master sales table to itself so that both can be mentioned in the same expression.

While this works, a retailer with 100 million items in the sales table will be very unhappy waiting all day for that table to be joined to itself.

A second technique for solving this simple problem is the use of embedded selects. The outer, or main, query retrieves last year's sales. The inner, or embedded, query must contain programmatic references to the outer query so that it is properly correlated. End-user query tools do not allow you to construct embedded selects. But even good programmers have trouble writing the code, and multiway comparisons (for example, comparing market share over time) are virtually impossible to fathom.

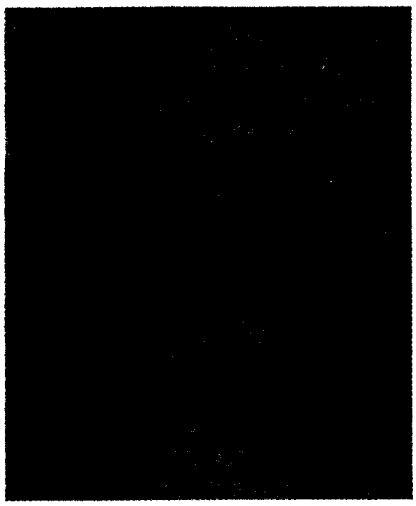
Moreover, relational DBMSs tend to process these complex SQL statements inefficiently.

A third technique is the use of a nonstandard SQL syntax popularized by Oracle, known as DECODE. Unfortunately, the primitive and nonstandard logic of DECODE only allows constants and not expressions to be used as criteria, and this method also fails when it comes to multiway comparisons.

The only remaining choice is to give up on SQL directly, retrieve this year's sales and last year's sales in separate queries, and then stitch the results together outside the database environment. The stitching-together process can be complicated if, for example, you are breaking out the results into this year's products and last year's products. One difficulty: these lists usually have different products in them, which means the business analyst must implement a manual "outer join."

None of these approaches can handle real comparisons. Take a close look at a complex management report. Chances are that the report requires as many as a dozen simultaneous comparisons. We have crossed the line from messy to not implementable.

Amazingly, an even more basic business analysis capability than comparisons is not just impractical in SQL, it's impossible!



SQL does not support sequential computations. This means, for instance, that you can't number your answer set.

The inability to march down the answer set performing some computation really puts a crimp in a number of common report-writing applications. Many reports are nothing more than numerical rankings of various financial measures. Companies that use this style of analysis call these reports "ranking reports." You can't do ranking reports in SQL.

Okay, we admit, the extra work to do ranking isn't great using most query tools. But related problems are tougher. Since you can't march down the answer set, you also can't do a moving sum, or a moving average, or a top 10, or a bottom 25%. You can't process a time series by performing a cumulative sum down the page. You can't do fancy things like label a row by its tertile value (such as high, medium, or low), or by a quartile value.

You also can't create break rows in SQL. A break row is a special total row that occurs when a value in one of your sorted columns changes. Do any of your reports have break rows? If you are typical, every one does. This is another consequence of not being able to march down the answer set doing useful things.

The bottom line is that SQL is only doing half the job for you. Someone, probably you, is going to have to implement all sorts of complicated logic to finish the job.

The lack of sequential processing is a glaring omission when it comes to supporting the business world. End users and application analysts have much better things to do than implement numbering, ranking, moving averages, top 10s, and break rows in every application.

Sequential-processing functions can easily be added to the SQL semantics.

Relational Theory Runs Smack into Reality

Ralph Kimball has learned the hard lessons of decision support systems from his days at Metaphor and Red Brick Systems and from building decision support systems as a consultant for scores of large corporate customers.

While consulting, Kimball realized that even with state-of-the-art front-end tools generating reams of SQL, a natural limit was being reached that thwarted the application developer and made it very difficult to deliver reports and business analyses. The problem was that even simple business questions required absurdly complex SQL. Because the SQL was so complicated, the relational database systems nearly always became bogged down. And finally, when the result came back, SQL had only finished part of the job and left far too much for the application developer to clean up.

In Kimball's consulting practice, he noticed a theme beginning to emerge from those pioneer decision support shops that gradually had built track records of successful installations serving large numbers of users. Without exception, every one of the decision support installations succeeding over the long term had gravitated to a simple, extremely denormalized database design. The most striking finding was that these designs were invariably dimensional.

The database designs all consisted of a central fact table surrounded by dimension tables representing intuitive constructs such as Product, Customer, and Time. When the database had a simple design, the SQL became simpler, and the users and the software both were happier.

Kimball spent the next six years founding Red Brick and designing its relational database system for decision support. The Red Brick DBMS was tightly optimized to process the giant dimensional databases that his consulting customers were building. Although Kimball has turned over the CEO role to Chris Erickson, Red Brick continues with Kimball's focus on dimensional databases for decision support. Kimball continues to be a major Red Brick stockholder, but his consulting applies dimensional modeling to a full range of database products. ***–Kevin Strehlo***

And we don't want to wait for the 1,000-page specification of the SQL 3 standard to clear committee in another two years and finally be implemented around the turn of the century. It is time to stop being purists and fix our database language.

A CALL TO ARMS

We own this language. We bought it, starting 10 years ago when we all committed ourselves to relational databases. The vendors got transaction processing to work just fine, thank you. Now let's do a similar job on queries.

Fortunately, fixing the two glaring business problems, comparisons and sequential processing, does not mean we need to backtrack. Support for both functions can be added to SQL in ways that will be compatible with the current language standard. Doing so will dramatically increase the productivity of SQL application programmers by freeing them from having to implement pick-and-shovel processing in their client applications or programming fiendishly complex SQL with embedded references.

So, let's plunge in. Here's a template for the new SQL functions that we need.

First, imagine supporting comparisons with a simple ALTERNATE keyword. ALTERNATE is associated with a constraint. The ALTERNATE constraint replaces all constraints on the same table in the surrounding query. The value of the ALTERNATE clause is interpreted in the context of each row in the answer set. This context, in effect, supplies the constraints on all other tables.

To see how this works, consider a query that brings back this year's sales:

```
select productname,
 sum(salesdollars)
from sales, product, market, time
where <join-constraints>
 ...and time.year=1994...
group by productname
```

To compute the ratio to last year's sales, we write

```
select productname,
 sum(salesdollars)/sum (ALTERNATE
salesdollars where time.year=1993)
from sales, product, market, time
where <join constraints>
 ...and time.year=1994...
group by productname
```

The italicized part is the only thing that needs to be added. The rest of the SQL statement remains unaffected.

ALTERNATE is a kind of smart correlated subquery. Its advantages are that it is expressed in an in-line notation that is easy to read. Hopefully it is simple enough that query tool vendors will be able to put it in their user interfaces.

The second needed extension to SQL handles sequential processing.

Imagine that after the answer set is ready to be shipped to the user, we are permitted to apply sequential-processing functions. Even purists have to admit that at that point there is a definite order to the answer set. The first record comes out first, and the last record comes out last. We may have even sorted it (SQL's only sop to the need for sequence).

For any given expression in the select list, allow the following new sequential functions:

```
number(),
numberof(),
totalof(expression).
rank(expression),
tertile(expression),
quartile(expression),
movingavg(expression, num),
movingsum(expression, num),
cumulative(expression).
```

In the simplest form, these new functions are used in the select list, with no further modifications to the SQL. For instance, our original simple example of numbering the answer set now reads like:

```
select productname,
 number(), sum(salesdollars)
from sales, product, market, time
where <join constraints>
 ...and time.year=1994...
group by productname
```

Now we have numbers next to our sales results. Again, we have italicized the only part of the SQL that we changed.

Ranks are similar.

We can ask for the top 10 items by allowing the sequential functions to appear in the WHERE clause:

```
select productname,
 sum(salesdollars)
from sales, product, market, time
where <join constraints>
rank(sum(salesdollars))≤10
 ...and time.year=1994...
```

Break rows will require a new BREAK BY clause quite similar to ORDER BY and GROUP BY. Old-time SAS users will recognize this tried and true syntax.

How do we make all this happen? Send us letters to the editor. Talk to your vendor. Join us in fixing our language. We need to get back to work solving business problems and stop wasting so much of our energy doing data processing.

Things SQL Can't Do

SQL can't handle anything requiring sequential processing, including percents, ratios, or differences. It also has trouble doing comparisons. What follows are examples of answers you might want but can't get easily with SQL, not even from the star-shaped dimensional data model.

Item Dimension

- **Sales ratio of Alice's products vs. Julie's products**
- **Sales percent of Alice's products vs. total products**
- **Sales ratio of promoted vs. nonpromoted products**

Customer Dimension

- **Quantity shipped to discount stores vs. department stores**
- **Sales of my products to existing vs. new customers**

Time Dimension

- **Christmas sales this year vs. last year**
- **YTD sales this year vs. YTD sales last year**