# Kimball Data Warehouse Toolkit

## Ch 3 - Retail Sales Case Study

* Best way to understand principles of dimensional modeling = work through a series of tangible examples
* By visualizing real cases, you hold the particular design challenges + solutions in your mind more effectively than if presented abstractly
* Universities, insurance companies, banks, + airlines alike surely need the techniques developed in this retail chapter
* Besides, thinking about someone else’s business is refreshing
* It is too easy to let historical complexities derail you when dealing with data from your company
* By stepping outside your organization and then returning with a well-understood design principle (or two), it is easier to remember the spirit of the design principles as you descend into the intricate details of your business

### 4-step Dimensional Design Process

#### Step 1: Select the Business Process

* **Business process** **= a low-level activity performed by an organization** (taking orders, invoicing, receiving payments, handling service calls, registering students, performing a medical procedure, or processing claims)
* To identify an organization’s business processes, helpful to understand several common characteristics:
* **Frequently expressed as action verbs** because they represent activities the business performs
* Companion **dimensions describe *descriptive context*** associated with each business process event
* Typically **supported by an operational system** (ex: the billing or purchasing system)
* **Generate or capture KPI’s**
* Sometimes the metrics are a *direct* result of the business process, + measurements are derivations at other times
* Analysts invariably want to scrutinize + evaluate these metrics by a seemingly limitless combination of filters and constraints
* **Usually triggered by an input and result in output metrics**
* In many organizations, there’s a series of processes in which the outputs from one process become the inputs to the next
* In the parlance of a dimensional modeler, **this series of processes results in a series of fact tables**
* Need to listen carefully to a business to identify the organization’s business processes because business users can’t readily answer the question, “What business process are you interested in?”
* **Performance measurements users want to analyze in the DW/BI system result from business process events**
* Sometimes business users talk about strategic business *initiatives* instead of business *processes*
* These initiatives are typically broad enterprise plans championed by executive leadership to deliver competitive advantage
* **In order to tie a business *initiative* to a business *process*** representing a project-sized unit of work for a DW/BI team, **need to decompose a business initiative into the underlying processes**
* This means digging a bit deeper to understand the data + operational systems that support the initiative’s analytic requirements
* It’s also worth noting what a business process is *not*
* **Organizational business departments or functions do NOT equate to business processes**
* By focusing on processes, rather than on functional departments, consistent information is delivered more economically throughout the organization
* If you design departmentally-bound dimensional models, you inevitably duplicate data with different labels and data values
* **The best way to ensure consistency is to publish the data once**

#### Step 2: Declare the Grain

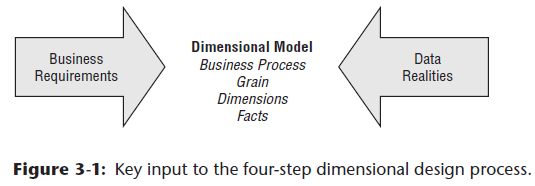
* Declaring the **grain**means **specifying exactly what an individual fact table row represents**
* Grain **conveys the level of detail associated with the fact table measurements + provides the answer to the question, “How do you describe a single row in the fact table?”**
* Grain is **determined by the *physical* realities of the operational system that captures the business process’s events**
* Example grain declarations include:
* One row per scan of an individual product on a customer’s sales transaction
* One row per line item on a bill from a doctor
* One row per individual boarding pass scanned at an airport gate
* One row per daily snapshot of the inventory levels for each item in a warehouse
* One row per bank account each month
* These **grain declarations are expressed in *business terms***
* Perhaps you were expecting the grain to be a traditional declaration of the fact table’s PK
* Although the grain ultimately *is* equivalent to the PK, it’s a **mistake to list a set of dimensions + then assume this list is the grain declaration**
* **Whenever possible, express the grain in business terms**
* Dimensional modelers sometimes try to bypass this seemingly unnecessary step of the 4-step design process
* **Don’t: Declaring the grain is a critical step that can’t be taken lightly**
* In debugging thousands of dimensional designs over the years, the most frequent error is not declaring the grain of the fact table at the beginning of the design process
* If the grain isn’t clearly defined, the whole design rests on quicksand, discussions about candidate dimensions go around in circles, + rogue facts sneak into the design
* An inappropriate grain haunts a DW/BI implementation
* Extremely important that everyone on the design team reaches agreement on the fact table’s granularity
* However, you **may discover in steps 3 or 4 of the design process that the grain statement is wrong**
* **This is okay,** but then you ***must* return to step 2**, restate the grain correctly, + revisit steps 3 and 4 again

#### Step 3: Identify the Dimensions

* **Dimensions** fall out of the question, **“How do business people describe the data resulting from the business process measurement events?”**
* **Need to decorate fact tables with a robust set of dimensions representing all possible descriptions that take on single values in the context of each measurement**
* **If clear about the grain, dimensions typically can easily be identified as they represent the “who, what, where, when, why, and how” associated with the event**
* Common dimensions: date, product, customer, employee, facility
* **With the choice of each dimension, then list all discrete, text-like attributes that flesh out each dimension table**

#### Step 4: Identify the Facts

* **Facts** are determined by answering the question, **“What is the process measuring?”**
* Business users are keenly interested in analyzing these performance metrics
* **All candidate facts in a design *must* be true to the grain** defined in step 2
* **Facts that clearly belong to a different grain must be in a *separate* fact table**
* Typical facts: numeric additive figures, such as quantity ordered or dollar cost amount.
* Need to consider both your business users’ requirements + the realities of your source data in tandem to make decisions regarding the four steps, as illustrated below



* **Resist the temptation to model the data by looking at source data alone**
* May be less intimidating to dive into the data rather than interview a business person
* However, the **data is no substitute for business user input**
* Unfortunately, many organizations have attempted this path-of-least-resistance data-driven approach but without much success

### Retail Case Study

* Patterns discussed in the context of this case study are relevant to virtually every dimensional model regardless of the industry
* Imagine you work in the HQ of a large grocery chain w/ 100 grocery stores spread across 5 states
* Each store has a full complement of departments (grocery, frozen foods, dairy, meat, produce, bakery, floral, and health/beauty aids)
* Each store has ~60,000 individual products (stock keeping units, or SKUs), on its shelves
* Data is collected at several interesting places in a grocery store
* Some of the most useful data is collected at the cash registers as customers purchase products.
* The POS system scans product barcodes at the cash register, measuring consumer takeaway at the front door of the grocery store
* Other data is captured at the store’s back door where vendors make deliveries
* At a grocery store, management is concerned with the logistics of ordering, stocking, + selling products while maximizing profit
* Profit ultimately comes from charging as much as possible for each product, lowering costs for product acquisition + overhead, + at the same time attracting as many customers as possible in a highly competitive environment
* Some of the most significant management decisions have to do with pricing + promotions
* Both store management and HQ marketing spend a great deal of time tinkering w/ pricing + promotions
* Such promotions include temporary price reductions, ads in newspapers + newspaper inserts, displays in the grocery store, + coupons.
* Most direct + effective way to create a surge in the volume of product sold is to lower price dramatically
* A 50-cent reduction in the price of paper towels, especially when coupled w/ an ad + display, can cause the sale of the paper towels to jump by a factor of 10
* Unfortunately, such a big price reduction usually is not sustainable because the towels probably are being sold at a loss
* As a result of these issues, the visibility of all forms of promotion is an important part of analyzing the operations of a grocery store

#### Step 1: Select the Business Process

* **1st step in the design = decide what business process to model by combining an understanding of the business requirements w/ an understanding of the available source data**
* **NOTE**: The **1st DW/BI project** should focus on the **business process that is both the most critical** **to business users**, as well as **the most feasible**
* **Feasibility** covers a range of considerations, including **data availability** **+** **quality**, as well as **organizational readiness**.
* In our retail case study, management wants to better understand customer purchases as captured by the POS system
* Thus, the business process you’re modeling is POS retail sales transactions
* This data enables the business users to analyze which products are selling in which stores on which days under what promotional conditions in which transactions

#### Step 2: Declare the Grain

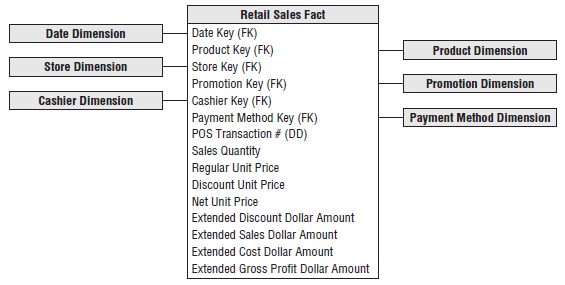
* After the business process has been identified, the design team faces a serious decision about the granularity
* ***What level of data detail should be made available in the dimensional model?***
* **Tackling data at its lowest atomic grain makes sense for many reasons**
* **Atomic data is highly dimensional 🡪 The more detailed + atomic a fact measurement, the more things you know for sure**
* **All those things you know for sure translate into dimensions**
* In this regard, **atomic data is a perfect match for the dimensional approach**
* **Atomic data provides maximum analytic flexibility because it can be constrained + rolled up in every way possible**
* Detailed data in a dimensional model is poised + ready for the ad hoc attack by business users
* **NOTE:** **Develop dimensional models representing the most detailed, atomic information captured by a business process**
* You *could* declare a more summarized granularity representing an aggregation of atomic data
* However, as soon as you select a **higher-level grain**, you **limit yourself to fewer and/or potentially less detailed dimensions**
* The less-granular model is **immediately vulnerable to unexpected user requests to drill down into the details**
* Users inevitably run into an analytic wall when not given access to the atomic data
* Although aggregated data plays an important role for performance tuning, it is *not* a substitute for giving users access to the lowest level details
* Users can easily summarize atomic data, but it’s impossible to create details from summary data
* Unfortunately, some industry pundits remain confused about this point
* They claim dimensional models are only appropriate for summarized data + criticize the dimensional modeling approach for its supposed need to anticipate the business question
* This misunderstanding goes away when detailed, atomic data is made available in a dimensional model
* In our case study, the **most granular data is an individual product on a POS transaction,** assuming the POS system rolls up all sales for a given product within a shopping cart into a single line item
* Although users probably are not interested in analyzing single items associated with a specific POS transaction, you **can’t predict all the ways they’ll want to cull through that data**
* Ex: They may want to understand the difference in sales on Monday vs. Sunday, assess whether it’s worthwhile to stock so many individual sizes of certain brands, understand how many shoppers took advantage of the 50c -off promotion on shampoo, or determine the impact of decreased sales when a competitive diet soda product was promoted heavily
* Although none of these queries calls for data from one specific transaction, they are broad questions that require detailed data sliced in precise ways
* None of them could have been answered if you elected to provide access only to summarized data
* **NOTE**: A **DW/BI system almost *always* demands data expressed at the lowest possible grain**, not because queries want to see individual rows but because **queries need to cut through the details in very precise ways**

#### Step 3: Identify the Dimensions

* After the grain of the fact table has been chosen, the choice of dimensions is straightforward.
* The product and transaction fall out immediately
* **Within the framework of the primary dimensions, you can ask whether other dimensions can be attributed to the POS measurements**, such as date of the sale, store where a sale occurred, promotion under which a product is sold, cashier who handled a sale, + potentially method of payment
* **NOTE**: **A careful grain statement determines the primary dimensionality of the fact table**
* **Add more dimensions to the fact table if additional dimensions naturally take on only *one* value under each combination of the primary dimensions**
* **If the additional dimension violates the grain by causing additional fact rows to be generated, the dimension needs to be disqualified or the grain statement needs to be revisited**
* The following descriptive dimensions apply to this use case: date, product, store, promotion, cashier, + method of payment
* In addition, the POS transaction ticket number is included as a special dimension (a **degenerate dimension**)
* Before fleshing out the dimension tables with descriptive attributes, complete the final step of the four-step process 🡪 don’t want to lose sight of the forest for the trees at this stage of the design

#### Step 4: Identify the Facts

* Final step in the design = make a careful determination of which facts will appear in the fact table
* Again, the **grain declaration helps anchor your thinking**
* Simply put, **the facts must be true to the grain**
* The individual product line item on the POS transaction in this case
* **When considering potential facts, may again discover adjustments need to be made to either earlier grain assumptions or choice of dimensions.**
* The facts collected by the POS system include sales quantity, per unit regular, discount, net paid prices, and extended discount + sales dollar amounts
* Extended sales dollar amount equals sales quantity multiplied by net unit price
* Likewise, extended discount dollar amount is sales quantity multiplied by unit discount amount
* Some sophisticated POS systems also provide a standard dollar cost for the product as delivered to the store by the vendor
* Presuming this cost fact is readily available + doesn’t require a heroic activity-based costing initiative, you can include the extended cost amount in the fact table
* The fact table begins to take shape below



* 4 of the facts (sales quantity + the extended discount, sales, + cost dollar amounts) are beautifully additive across all the dimensions
* Can slice and dice the fact table by the dimension attributes with impunity, and every sum of these 4 facts is valid and correct

##### Derived Facts

* You can **compute gross profit by subtracting extended cost dollar amount from extended sales dollar amount, or revenue**
* **Although computed**, gross profit is also **perfectly additive across all the dimensions**
* Can calculate gross profit of any combo of products sold in any set of stores on any set of days
* **Dimensional modelers sometimes question whether a calculated derived fact (calculated from other facts) should be stored in the database**
* Generally recommended to be stored physically
* In this case study, a gross profit calculation is straightforward, but *storing it* means it’s **computed consistently in the ETL process, eliminating the possibility of user calculation errors**
* *Cost of a user incorrectly representing gross profit overwhelms the minor incremental storage cost*
* **Storing it also ensures all users + BI reporting applications refer to gross profit consistently**
* Because gross profit can be calculated from adjacent data w/in a single fact table row, some argue you should perform the calculation in a **view** that is indistinguishable from the table
* This is reasonable if all users access the data via the view and no users with ad hoc query tools can sneak around the view to get at the physical table
* **Views are a reasonable way to minimize user error while saving on storage, but the DBA must allow no exceptions to accessing the data through the view**
* Likewise, some organizations want to perform calculations in the BI tool
* **Again, this works if all users access the data using a common tool, which is seldom the case in experience**
* However, **sometimes non-additive metrics on a report such as %’s or ratios *must* be computed in the BI application because the calculation cannot be precalculated and stored in a fact table**
* **OLAP cubes excel in these situations**

##### Non-Additive Facts

* **Gross margin can be calculated by dividing gross profit by extended sales dollar revenue**
* Gross margin is a **non-additive fact**because it **can’t be summarized along any dimension**
* Can calculate the gross margin of any set of products, stores, or days by remembering to sum the revenues and costs respectively *before dividing*
* **NOTE**: **%’s and ratios, such as gross margin, are non-additive**
* The **numerator and denominator should be stored in the fact table**
* The **ratio can then be calculated in a BI tool for any slice of the fact table by remembering to calculate the ratio of the sums, NOT the sum of the ratios**
* **Unit price is another non-additive fact**
* Unlike the extended amounts in the fact table, summing unit price across any of the dimensions results in a meaningless, nonsensical number
* Ex: You sold 1 widget at a unit price of $1.00 and 4 widgets at a unit price of 50c each
* *Could* sum the sales quantity to determine that 5 widgets were sold
* Likewise, could sum the sales dollar amounts ($1.00 and $2.00) to arrive at a total sales amount of $3.00
* However, ***can’t* sum the unit prices ($1.00 and 50c) and declare that the total unit price is $1.50**
* Similarly, **shouldn’t announce that the average unit price is 75c**
* Properly weighted average unit price should be calculated by taking the total sales amount ($3.00) and dividing by the total quantity (5 widgets) to arrive at a 60c average unit price
* You’d never arrive at this conclusion by looking at the unit price for each transaction line in isolation
* To analyze the average price, you must add up the sales dollars and sales quantities *before* dividing the total dollars by the total quantity sold
* Fortunately, many BI tools perform this function correctly
* **Some question whether non-additive facts should be physically stored in a fact table**
* This is a legitimate question given their **limited analytic value**, aside from printing individual values on a report or applying a filter directly on the fact, which are both atypical
* In some situations, a fundamentally non-additive fact such as a temperature is supplied by the source system
* These **non-additive facts may be averaged carefully over many records, if the business analysts agree that this makes sense**

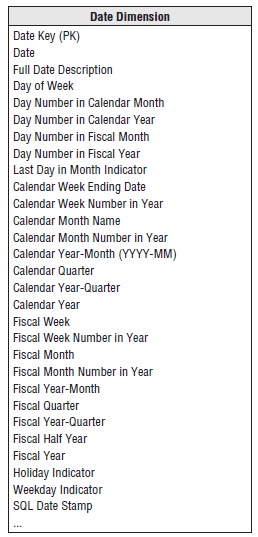
##### Transaction Fact Tables

* **Transactional business processes are the most common**
* **Fact tables representing these processes share several characteristics**:
* The **grain** of atomic transaction fact tables **can be succinctly expressed in the context of the transaction**, such as 1 row per transaction or 1 row per transaction line.
* Because these fact tables record a transactional event, they are **often sparsely populated**
* In our case study, we certainly wouldn’t sell every product in every shopping cart
* Even though transaction fact tables are unpredictably + sparsely populated, they **can be enormous**
* Most billion and trillion row tables in a DW are transaction fact tables
* Transaction fact tables **tend to be highly dimensional**
* **Metrics resulting from transactional events are typically additive** **as long as they have been extended by the quantity amount, rather than capturing per unit metrics.**
* At this early stage of the design, it is **often helpful to estimate the number of rows in your largest table, the fact table**
* In this case study, it simply may be a matter of talking w/ a source system expert to understand how many POS transaction line items are generated on a periodic basis
* Retail traffic fluctuates significantly from day to day, so you need to understand the transaction activity over a reasonable period of time
* Alternatively, you could estimate the number of rows added to the fact table annually by dividing the chain’s annual gross revenue by the average item selling price
* Assuming gross revenues are $4B/year + that average price of an item on a customer ticket is $2.00, you can calculate that there are approximately 2B transaction line items/year
* This is a typical engineer’s estimate that gets you surprisingly close to sizing a design directly from your armchair
* **As designers, you always should be triangulating to determine whether your calculations are reasonable**

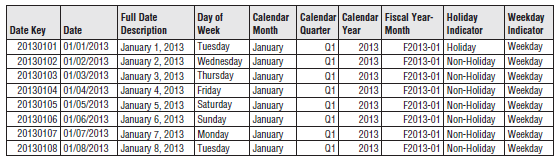
### Dimension Table Details

#### Date Dimension

* This is a **special dimension b/c it is the 1 dimension nearly guaranteed to be in *every* dimensional model since virtually every business process captures a time series of performance metrics**
* In fact, **date is usually the 1st dimension in the underlying partitioning scheme of the database so that the successive time interval data loads are placed into virgin territory on the disk**
* **Unlike most other dimensions, you can build a date dimension table in advance**
* May put 10 or 20 years of rows representing individual days in the table, so you can cover the history you have stored, as well as several years in the future.
* Even 20 years’ worth of days is only ~7,300 rows, which is a relatively small dimension table
* For a daily date dimension table in a retail environment, we recommend the partial list of columns shown below:



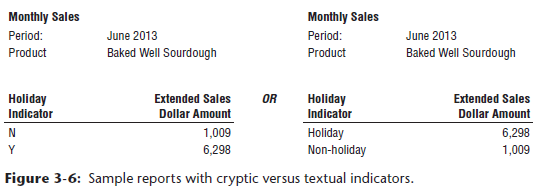
* **Each column in the date dimension table is defined by the particular day that the row represents**
* The day-of-week column contains the day’s name, such as Monday, + would be used to create reports comparing Monday business with Sunday business
* The day number in a calendar month column starts with 1 at the start of each month + runs to 28, 29, 30, or 31 depending on the month, + is useful for comparing the same day each month
* Similarly, you could have a month number in year (1, ..., 12)
* **All these integers support simple date arithmetic across year and month boundaries**
* **For reporting, should include both long and abbreviated labels**
* Ex: You’d want a month name attribute with values such as January
* In addition, a year-month (YYYY-MM) column is useful as a report column header
* You likely also want a quarter number (Q1, . . ., Q4), as well as a year-quarter, such as 2013-Q1
* You’d include similar columns for the fiscal periods if they diff er from calendar periods
* Sample rows containing several date dimension columns are illustrated below:



* Some designers pause at this point to ask **why an explicit date dimension table is needed**
* They reason that if the date key in the fact table is a date type column, then any SQL query can directly constrain on the fact table date key and use natural SQL date semantics to filter on month or year while avoiding a supposedly expensive JOIN
* This reasoning falls apart for several reasons
* 1) **If your RDBMS can’t handle an efficient JOIN to the date dimension table, you’re in deep trouble**
* Most database optimizers are quite efficient at resolving dimensional queries; it is not necessary to avoid JOINs like the plague.
* 2) **Since the average business user is not versed in SQL date semantics, they’d be unable to request typical calendar groupings**
* 3) SQL date functions do not support filtering by attributes such as weekdays versus weekends, holidays, fiscal periods, or seasons
* **Presuming the business needs to slice data by these nonstandard date attributes, then an explicit date dimension table is essential**
* **Calendar logic belongs in a dimension table, not in the application code**
* **NOTE**: **Dimensional models always need an explicit date dimension table**
* There are many date attributes not supported by the SQL date function, including week numbers, fiscal periods, seasons, holidays, + weekends
* **Rather than attempting to determine these nonstandard calendar calculations in a query, look them up in a date dimension table**

##### Flags and Indicators as Textual Attributes

* Like many operational flags + indicators, the date dimension’s holiday indicator is a simple indicator with 2 potential values
* Because dimension table attributes serve as report labels and values in pull-down query filter lists, **this indicator should be populated with meaningful values such as Holiday or Non-holiday instead of the cryptic Y/N, 1/0, or True/False**
* Imagine a report comparing holiday vs. non-holiday sales for a product like below



* **More meaningful domain values for this indicator translate into a more meaningful, self-explanatory report**
* Rather than decoding flags into understandable labels in the BI application, **store decoded values in the database so they’re consistently available to all users regardless of their BI reporting environment or tools**
* Similar argument holds true for the weekday indicator with a value of Weekday or Weekend
* Saturdays and Sundays obviously would be assigned the weekend value
* Of course, multiple date table attributes can be jointly constrained, so you can easily compare weekday holidays with weekend holidays

##### Current and Relative Date Attributes

* **Most date dimension attributes are not subject to updates** (June 1, 2013 will always roll up to June, Calendar Q2, and 2013)
* However, **there *are* attributes you can add to the basic date dimension that will change over time,** including IsCurrentDay, IsCurrentMonth, IsPrior60Days, and so on
* IsCurrentDay obviously must be updated each day + is useful for generating reports that always run for today
* A nuance to consider is the day that IsCurrentDay refers to
* Most DWs load data daily, so IsCurrentDay would refer to yesterday (or more accurately, the most recent day loaded)
* **Might also add attributes to the date dimension that are unique to your corporate calendar**, such as IsFiscalMonthEnd
* **Some date dimensions include updated lag attributes**
* The lag day column would take the value 0 for today, –1 for yesterday, +1 for tomorrow, and so on
* **Could easily be a computed column rather than physically stored**
* Might be useful to set up similar structures for month, quarter, and year
* **Many BI tools include functionality to do prior period calculations, so these lag columns may be unnecessary**

##### Time-of-Day as a Dimension or Fact

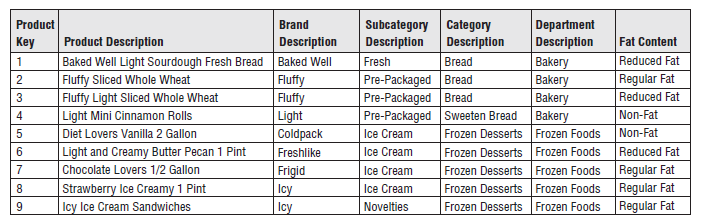
* Although date and time are comingled in an operational date/time stamp, **time-of-day is typically separated from the date dimension to avoid a row count explosion in the date dimension**
* As noted earlier, a date dimension with 20 years of history contains ~7,300 rows
* If you changed the grain of this dimension to 1 row per minute in a day, you’d end up w/ > 10 million rows to accommodate the 1,440 minutes per day
* If you tracked time to the second, you’d have > 31 million rows per year!
* **Because the date dimension is likely the most frequently constrained dimension in a schema, it should be kept as small and manageable as possible**
* **If you want to filter or roll up time periods based on summarized day part groupings,** such as activity during 15-minute intervals, hours, shifts, lunch hour, or prime time, **time-of-day would be treated as a full-fledged dimension table with 1 row per discrete time period,** such as 1 row per minute within a 24-hour period resulting in a dimension with 1,440 rows.
* If there’s **no need to roll up or filter on time-of-day groupings, time-of-day should be handled as a simple date/time fact in the fact table**
* By the way, **business users are often more interested in time lags**, such as the transaction’s duration, **rather** **than discreet start and stop times**
* Time lags can easily be computed by taking the difference between date/time stamps
* These date/time stamps also allow an application to determine the time gap between 2 transactions of interest, even if these transactions exist in different days, months, or years.

#### Product Dimension

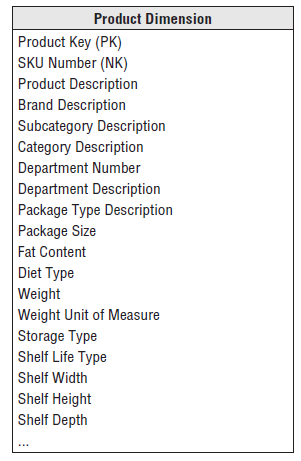
* The product dimension describes every SKU in the grocery store
* Although a typical store may stock 60,000 SKUs, when you account for different merchandising schemes + historical products that are no longer available, the product dimension may have 300,000 or more rows
* **The product dimension is almost always sourced from the operational product master file**
* Most retailers administer their product master file at HQ and download a subset to each store’s POS system at frequent intervals
* It is HQ’s responsibility to define the appropriate product master record (+ unique SKU number) for each new product

##### Flatten Many-to-One Hierarchies

* The product dimension represents the many descriptive attributes of each SKU
* The **merchandise hierarchy** is an important **group of attributes**
* Typically, individual SKUs roll up to brands, brands roll up to categories, + categories roll up to departments
* **Each of these is a many-to-one relationship**
* This merchandise hierarchy + additional attributes are shown for a subset of products below



* **For each SKU, all levels of the merchandise hierarchy are well defined**
* Some attributes, such as SKU description, are unique (In this case, there’re 300,000 different values)
* At the other extreme, there are only perhaps 50 distinct values of the department attribute
* Thus, on average, there are **6,000 repetitions of each unique value in the department attribute**
* **This is perfectly acceptable**! **You do not need to separate these repeated values into a second normalized table to save space**
* Remember **dimension table space requirements pale in comparison with fact table space considerations**
* **NOTE:** **Keeping the repeated low cardinality values in the *primary* dimension table is a fundamental dimensional modeling technique**
* **Normalizing these values into separate tables defeats the primary goals of simplicity + performance**
* Many of the attributes in the product dimension table are *not* part of the merchandise
* Hierarchy
* Package type attribute might have values such as Bottle, Bag, Box, or Can, + any SKU in any department could have one of these values
* It **often makes sense to combine a constraint on this attribute with a constraint on a merchandise hierarchy attribute**
* Ex: You could look at all SKUs in the Cereal category packaged in Bags
* Put another way, you can **browse among dimension attributes regardless of whether they belong to the merchandise hierarchy**
* **Product dimension tables typically have more than one explicit hierarchy**.
* A recommended partial product dimension for a retail grocery dimensional model is shown below:



##### Attributes with Embedded Meaning

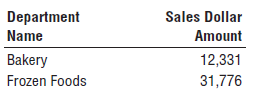
* Often **operational** product codes, identified in the dimension table by the **NK notation for natural key**, **have embedded meaning with different parts of the code representing significant characteristics of the product**
* ***In this case***, the **multipart attribute should be both preserved in its entirety within the dimension table, as well as broken down into its component parts, which are handled as separate attributes**
* Ex: If the 5TH through 9TH characters in the operational code identify the manufacturer, the manufacturer’s name should also be included as a dimension table attribute

##### Numeric Values as Attributes or Facts

* **Will sometimes encounter numeric values that don’t clearly fall into either the fact or dimension attribute categories**
* Ex: The standard list price for a product 🡪 definitely a numeric value, so the initial instinct is to place it in the fact table
* But typically, the standard price changes infrequently, unlike most facts that are often differently valued on every measurement event
* **If the numeric value is used primarily for calculation purposes, it likely belongs in the fact table**
* Because standard price is **non-additive**, you might multiply it by the quantity for an **extended amount** which *would be* **additive**
* Alternatively, if the standard price is used primarily for price variance analysis, perhaps the variance metric should be stored in the fact table instead
* **If the stable numeric value is used predominantly for filtering and grouping, it should be treated as a product dimension attribute**
* **Sometimes numeric values serve *both* calculation and filtering/grouping functions**
* **In these cases, store the value in *both* the fact and dimension tables**
* Perhaps the standard price in the fact table represents the valuation at the time of the sales transaction, whereas the dimension attribute is labeled to indicate it’s the current standard price
* **NOTE:** **Data elements that are used both for fact calculations and dimension constraining, grouping, + labeling should be stored in both locations**, even though a clever programmer could write applications that access these data elements from a single location
* It is **important** that **dimensional models be as consistent as possible** and **application development be predictably simple**
* **Data involved in calculations should be in fact tables and data involved in constraints, groups and labels should be in dimension tables**

##### Drilling Down on Dimension Attributes

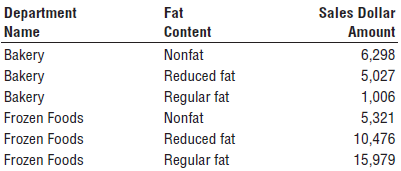
* A reasonable product dimension table can have 50+ descriptive attributes
* **Each attribute is a rich source for constraining and constructing row header labels**
* **Drilling down**= nothing more than asking for a row header from a dimension that provides more information
* Ex: You have a simple report summarizing the sales dollar amount by department



* To drill down, you can drag any other attribute, such as brand, from the product dimension into the report next to department, + automatically drill down to this next level of detail



* Could even drill down by the fat content attribute, *even though it isn’t in the merchandise hierarchy rollup*



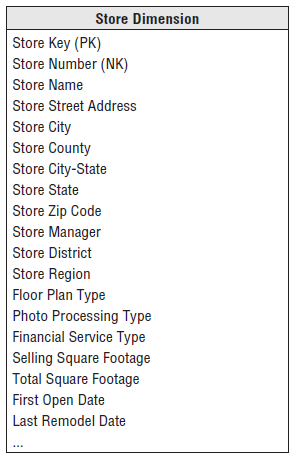
* **NOTE**: **Drilling down in a dimensional model is nothing more than adding row header attributes from the dimension tables**
* **Drilling up**= *removing* row headers
* You **can drill down or up on attributes from *more than one* explicit hierarchy *and* with attributes that are part of *no* hierarchy**
* The product dimension is a common dimension in many dimensional models.
* Great care should be taken to fill this dimension with as many descriptive attributes as possible
* **A robust and complete set of dimension attributes translates into robust and complete analysis capabilities for the business users**

#### Store Dimension

* This describes every store in the grocery chain
* Unlike the product master file that is almost guaranteed to be available in every large grocery business, there may not be a comprehensive store master file
* POS systems may simply supply a store number on the transaction records
* In these cases, project teams must **assemble the necessary components** of the store dimension **from *multiple* operational sources**
* Often there will be a store real estate department at HQ who will help define a detailed store master file

##### Multiple Hierarchies in Dimension Tables

* The store dimension is the case study’s primary *geographic* dimension
* Each store can be thought of as a location + you can roll stores up to any geographic attribute, such as ZIP, county, + state in the US
* **Contrary to popular belief, cities and states within the United States are not a hierarchy**
* Since many states have identically named cities, you’ll want to include a City-State attribute in the store dimension
* Stores likely also roll up an internal *organization* hierarchy consisting of store districts and regions
* These two different store hierarchies are both easily represented in the dimension because both the geographic and organizational hierarchies are well defined for a single store row
* **NOTE**: It is **not uncommon to represent multiple hierarchies in a dimension table**
* The **attribute names and values should be unique across the multiple hierarchies**
* A recommended retail store dimension table is shown below:



* Floor plan type, photo processing type, + finance services type are all short text descriptors that describe the particular store
* These should not be one-character codes but rather should be 10- to 20-character descriptors that make sense when viewed in a pull-down filter list or used as a report label
* The column describing selling square footage is numeric and theoretically additive across stores
* Might be tempted to place it in the fact table
* However, it is **clearly a constant attribute of a store and is used as a constraint or label more often than it is used as an additive element in a summation**
* For these reasons, selling square footage belongs in the store dimension table

##### Dates Within Dimension Tables

* The first open date and last remodel date in the store dimension could be date type columns
* However, if users want to group and constrain on nonstandard calendar attributes (like the open date’s fiscal period), they are typically JOIN keys to copies of the date dimension table
* These date dimension copies are declared in SQL by the **view** construct + are semantically distinct from the primary date dimension
* The view declaration would look like the following:

create view first\_open\_date (first\_open\_day\_number, first\_open\_month,

...)

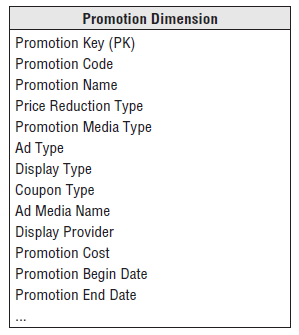
as select day\_number, month, ...

from date

* Now the system acts as if there is another physical copy of the date dimension table called FIRST\_OPEN\_DATE
* Constraints on this new date table have *nothing* to do with constraints on the primary date dimension joined to the fact table
* The first open date view is a permissible **outrigger (tables or entities that are shared by more than one dimension)** to the store dimension
* Notice we have carefully relabeled all the columns in the view so they cannot be confused with columns from the primary date dimension
* These **distinct logical views on a single physical date dimension** are an example of dimension **role playing**

#### Promotion Dimension

* The promotion dimension is potentially the most interesting dimension in the retail sales schema
* The promotion dimension describes the promotion conditions under which a product is sold
* Promotion conditions include temporary price reductions, end aisle displays, newspaper ads, + coupons
* This dimension is often called a **causal dimension**because it **describes factors thought to cause a change in product sales**
* Business analysts at both HQ and the stores are interested in determining whether a promotion is effective, + promotions are judged on 1+ of the following factors:
* Whether the **products under promotion experienced a gain in sales**, called **lift**, during the promotional period
* *Lift can be measured only if the store can* ***agree on what the baseline sales of the promoted products would have been*** *without the promotion*
* **Baseline** values can be **estimated from prior sales history** and, in some cases, with the help of **sophisticated models**.
* Whether the products under promotion showed a **drop in sales just prior to or after the promotion, canceling the gain** in sales during the promotion (**time shifting**)
* In other words, did you transfer sales from regularly priced products to temporarily reduced priced products?
* Whether the products under promotion showed a **gain in sales** but **other products** nearby on the shelf showed a **corresponding sales decrease** (**cannibalization**)
* Whether ***all* the products** in the promoted category of products **experienced a net overall gain** in sales, taking into account the time periods before, during, and after the promotion (**market growth**)
* Whether the promotion was **profitable**
* Usually the profit of a promotion is taken to be the incremental gain in profit of the promoted category over the baseline sales, taking into account time shifting + cannibalization, as well as the costs of the promotion
* The causal conditions potentially affecting a sale are not necessarily tracked directly by the POS system
* The transaction system keeps track of price reductions + markdowns, + the presence of coupons also typically is captured w/ the transaction b/c the customer either presents coupons at the time of sale or does not
* Ads and in-store display conditions may need to be linked from other sources
* **The various possible causal conditions are highly correlated**
* A temporary price reduction usually is associated with an ad and perhaps an end aisle display
* **For this reason, it makes sense to create 1 row in the promotion dimension for each combination of promotion conditions that occurs**
* Over the course of a year, there may be 1,000 ads, 5,000 temporary price reductions, + 1,000 end aisle displays, but may be only 10,000 combos of these 3 conditions affecting any particular product
* Ex: In a given promotion, most of the stores would run all 3 promotion mechanisms simultaneously, but a few of the stores may not deploy the end aisle displays
* In this case, 2 separate promotion condition rows would be needed, one for the normal price reduction + ad + display and one for the price reduction + ad only
* A recommended promotion dimension table is shown below



* From a purely logical point of view, you could record similar information about the promotions by separating the 4 causal mechanisms (price reductions, ads, displays, coupons) into separate dimensions rather than combining them into one dimension
* **Ultimately, this choice is the designer’s prerogative**
* Trade-offs in favor of keeping the 4 dimensions together include the following:
* If the 4 causal mechanisms are **highly correlated**, the **combined single dimension is not much larger than any one of the separated dimensions would be**
* The **combined single dimension can be browsed efficiently** to see how the various price reductions, ads, displays, + coupons are used together
* However, this browsing only shows the possible promotion combinations
* Browsing in the dimension table does *not* reveal which stores or products were affected by the promotion (*this information is found in the fact table*)
* Trade-offs in favor of separating the causal mechanisms into 4 distinct dimension tables include:
* The separated dimensions **may be more understandable to the business community** if users think of these mechanisms separately
* This would be *revealed during the business requirement interviews*
* **Administration of the separate dimensions may be more straightforward** than administering a combined dimension
* ***Keep in mind there is no difference in the content between these two choices***
* **NOTE:** The **inclusion of promotion cost attribute in the promotion dimension should be done with careful thought**
* This attribute can be used for constraining + grouping
* However, **this cost should *NOT* appear in the POS transaction fact table representing individual product sales *because it is at the wrong grain***
* This cost would have to reside in a fact table whose grain is the overall promotion

##### Null Foreign Keys, Attributes, and Facts

* Typically, many sales transactions include products that are not being promoted (hopefully)
* **The promotion dimension must include a row, with a unique key such as 0 or –1, to identify this no promotion condition and avoid a null promotion key in the fact table**
* **Referential integrity is violated if you put a null in a fact table column declared as a FK to a dimension table**
* In addition to the referential integrity alarms, **null keys are the source of great confusion to users because they can’t JOIN on null keys.**
* **WARNING**: **Avoid null keys in the fact table**
* A proper design includes a row in the corresponding dimension table to identify that the dimension is not applicable to the measurement
* We **sometimes encounter nulls as dimension attribute values**, which usually result when a given dimension row has not been fully populated, or when there are attributes that are not applicable to all the dimension’s rows
* In either case, **it’s recommended to substitute a descriptive string, such as Unknown or Not Applicable, in place of the null value**
* Null values essentially disappear in pull-down menus of possible attribute values or in report groupings, + special syntax is required to identify them
* If users sum up facts by grouping on a fully populated dimension attribute, and then alternatively, sum by grouping on a dimension attribute with null values, they’ll get different query results, + you’ll get a phone call because the data doesn’t appear to be consistent
* **Rather than leaving the attribute null, or substituting a blank space or a period, it’s best to label the condition, + users can then purposely decide to exclude the Unknown or Not Applicable from their query**
* It’s worth noting that some OLAP products prohibit null attribute values, so this is one more reason to avoid them
* Finally, we **can also encounter nulls as metrics in the fact table**
* Generally leave these null so that they’re properly handled in aggregate functions such as SUM, MIN, MAX, COUNT, and AVG which do the “right thing” with nulls
* **Substituting a zero instead would improperly skew these aggregated calculations**
* Data mining tools may use different techniques for tracking nulls
* You may need to do some additional transformation work beyond the above recommendations if creating an observation set for data mining

#### Other Retail Sales Dimensions

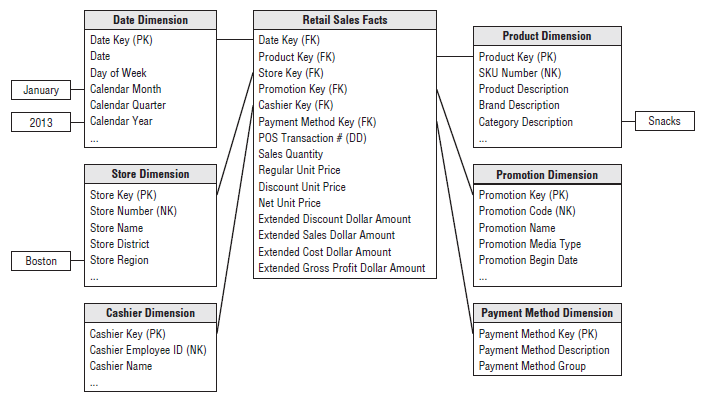
* ***Any* descriptive attribute that takes on a single value in the presence of a fact table measurement event is a good candidate to be added to an existing dimension or be its own dimension**
* The **decision whether a dimension should be associated with a fact table should be a binary yes/no based on the fact table’s declared grain**
* Ex: There’s probably a cashier identified for each transaction
* The corresponding cashier dimension would likely contain a small subset of nonprivate employee attributes
* Like the promotion dimension, the cashier dimension will likely have a No Cashier row for transactions that are processed through self-service registers
* A trickier situation unfolds for the payment method
* Perhaps the store has rigid rules and only accepts one payment method per transaction
* This would make life as a dimensional modeler easier b/c you’d attach a simple payment method dimension to the sales schema that would likely include a payment method description, along w/ perhaps a grouping of payment methods into either cash equivalent or credit payment types
* In real life, payment methods often present a more complicated scenario
* **If multiple payment methods are accepted on a single POS transaction, the payment method does not take on a single value at the declared grain**
* Rather than altering the declared grain to be something unnatural such as one row per payment method per product on a POS transaction, you would **likely capture the payment method in a *separate fact table* with a granularity of either 1 row per transaction (then the various payment method options would appear as separate facts) or 1 row per payment method per transaction (which would require a separate payment method dimension to associate with each row)**

#### Degenerate Dimension for Transaction Numbers

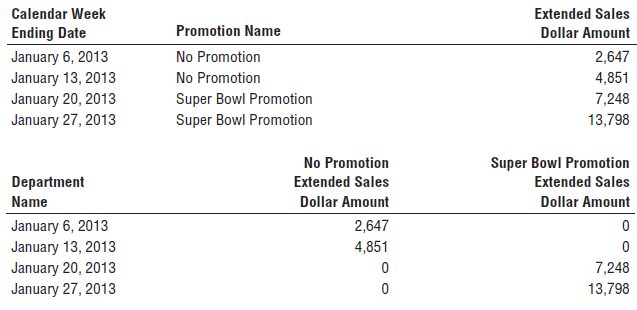
* **The retail sales fact table includes the POS transaction number on every line item row**
* In an **operational parent/child database, the POS transaction number would be the key to the transaction header record, containing all the information valid for the transaction as a whole, such as the transaction date and store identifier**
* **However, in the dimensional model, you have already extracted this interesting header information into other dimensions**
* The POS transaction number is still useful because it serves as the grouping key for pulling together all the products purchased in a single market basket transaction + also potentially enables you to link back to the operational system
* Although the POS transaction number looks like a dimension key in the fact table, the **descriptive items that might otherwise fall in a POS transaction dimension have been stripped off**
* **Because the resulting dimension is empty, we refer to the POS transaction number as a degenerate dimension****(dimensions that have no attributes other than a key, like transaction, invoice, or ticket numbers)**
* The ***natural* operational ticket number, such as the POS transaction number, sits by itself in the fact table without joining to a dimension table**
* **Degenerate dimensions are very common when the grain of a fact table represents a single transaction or transaction line because the degenerate dimension represents the unique identifier of the parent**
* **Order numbers, invoice numbers, and bill-of-lading numbers almost always appear as degenerate dimensions in a dimensional model.**
* **Degenerate dimensions often play an integral role in the fact table’s PK**
* In our case study, the PK of the retail sales fact table consists of the degenerate POS transaction number and product key, assuming scans of identical products in the market basket are grouped together as a single line item
* **NOTE**: **Operational transaction control numbers such as order numbers, invoice numbers, and bill-of-lading numbers usually give rise to empty dimensions and are represented as degenerate dimensions in transaction fact tables**
* **The degenerate dimension is a dimension key without a corresponding dimension table.**
* If, for some reason, **one or more attributes are legitimately left over after all the other dimensions have been created and seem to belong to this header entity**, you’d **simply create a normal dimension row with a normal join**
* *However, you would* ***no longer have a degenerate dimension***

### Retail Schema in Action

* With our retail POS schema designed, let’s illustrate how it’d be put to use in a query environment
* A business user might be interested in better understanding weekly sales dollar volume by promotion for the snacks category during January 2013 for stores in the Boston district
* For this, you’d place **query constraints** on month and year in the date dimension, district in the store dimension, and category in the product dimension



* If the query tool summed the sales dollar amount grouped by week ending date and promotion, the SQL query results would look similar to those below



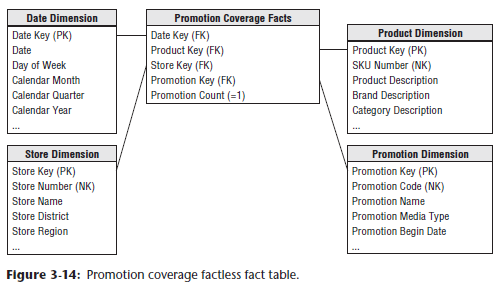
* You can plainly see the relationship between the dimensional model + the associated query
* **High-quality dimension attributes are crucial because they are the source of query constraints and report labels**
* If you use a BI tool with more functionality, the results would likely appear as a cross-tabular “pivoted” report, which may be more appealing to business users than the columnar data resulting from an SQL statement

### Retail Schema Extensibility

* Let’s turn our attention to **extending the initial dimensional design**
* Several years after the rollout of the retail sales schema, the retailer implements a frequent shopper program
* Rather than knowing an unidentified shopper purchased 26 items on a cash register receipt, you can now identify the specific shopper
* Imagine business user interest in analyzing shopping patterns by a multitude of geographic, demographic, behavioral, + other differentiating shopper characteristics
* The handling of this new frequent shopper information is relatively straightforward 🡪 **create a new frequent shopper dimension table and add another FK in the fact table**
* Because you can’t ask shoppers to bring in all their old cash register receipts to tag their historical sales transactions with their new frequent shopper number, you’d **substitute a default shopper dimension surrogate key (unique identifier, possibly generated by the system), corresponding to a Prior to Frequent Shopper Program dimension row, on the historical fact table rows**
* Likewise, not everyone who shops at the grocery store will have a frequent shopper card, so you’d also want to **include a Frequent Shopper Not Identified row in the shopper dimension**
* As discussed earlier w/ promotion dimensions, you **can’t have a null frequent shopper key in the fact table**
* Our **original schema gracefully extends to accommodate this new dimension largely because the POS transaction data was initially modeled at its most granular level**
* The **addition of dimensions applicable at that granularity did not alter the existing dimension keys or facts + all existing BI applications continue to run without any changes**
* If the grain was originally declared as daily retail sales (transactions summarized by day, store, product, + promotion) rather than the transaction line detail, you’d not have been able to incorporate the frequent shopper dimension
* **Premature summarization or aggregation inherently limits the ability to add supplemental dimensions because the additional dimensions often don’t apply at the higher grain**
* The **predictable symmetry of dimensional models enables them to absorb some rather significant changes in source data and/or modeling assumptions without invalidating existing BI applications, including:**
* **New dimension *attributes***
* If you discover new textual descriptors of a dimension, you can add these attributes as new columns
* *All existing applications will be oblivious to the new attributes and continue to function*
* If the new attributes are available only after a specific point in time, then Not Available or its equivalent should be populated in the old dimension rows
* Be forewarned that this scenario is more complicated if the business users want to track historical changes to this newly identified attribute
* If this is the case, pay close attention to the **slowly changing dimensions**
* **New dimensions**
* As just discussed, you can add a dimension to an existing fact table by adding a new FK column and populating it correctly with values of the PK from the new dimension.
* **New measured facts**
* If new measured facts become available, you can add them gracefully to the fact table
* The simplest case is when new facts are available in the same measurement event and at the same grain as the existing facts
* In this case, the fact table is altered to add the new columns, + the values are populated into the table
* If the new facts are only available from a point in time forward, null values need to be placed in the older fact rows.
* More complex situation = when new measured facts occur naturally *at a different grain*
* **If the new facts cannot be allocated or assigned to the original grain of the fact table, the new facts belong in their own fact table because it’s a mistake to mix grains in the same fact table**

### Fact-less Fact Tables

* There is one important question that cannot be answered by the previous retail sales schema: *What products were on promotion but did not sell?*
* The sales fact table records only the SKUs *actually sold*
* **There are no fact table rows with 0 facts** for SKUs that didn’t sell **because doing so would enlarge the fact table enormously**
* **In the relational world, a promotion coverage or event fact table is needed to answer the question concerning what *didn’t* happen**
* The promotion coverage fact table keys would be: date, product, store, + promotion, in this case study
* This obviously looks similar to the sales fact table we just designed; however, the **grain would be significantly different**
* In the case of the promotion coverage fact table, you’d **load one row for each product** on promotion in a store each day (or week, if retail promotions are a week in duration) **regardless of whether the product sold**.
* *This* fact table enables you to **see the relationship between the keys as defined by a promotion, *independent of other events***, such as actual product sales
* We refer to it as a **fact-less fact table**because it **has no measurement metrics; it merely captures the relationship between the involved keys**, as illustrated below



* **To facilitate counting, you can include a dummy fact**, such as promotion count in this example, which **always contains the constant value of 1**
* This is a cosmetic enhancement that enables a BI application to avoid counting one of the FK’s.
* To determine what products were on promotion but didn’t sell requires a 2-step process
* 1) Query the promotion fact-less fact table to determine the universe of products on promotion on a given day
* 2) Then determine what products sold from the POS sales fact table
* The answer to our original question is the **set difference**between these 2 lists of products
* If you work with data in an OLAP cube, it is often easier to answer the “what didn’t happen” question because the cube typically contains explicit cells for non-behavior

### Dimension and Fact Table Keys

* Now that the schemas have been designed, we’ll focus on the dimension and fact tables’ PK’s, along with other row identifiers.

#### Dimension Table Surrogate Keys

* **The unique PK of a dimension table should be a surrogate keyrather than relying on the operational system identifier, known as the natural key**
* Surrogate keys go by many other aliases: meaningless keys, integer keys, non-natural keys, artificial keys, and synthetic keys
* **Surrogate keys = simply integers assigned sequentially as needed to populate a dimension**
* The 1st product row is assigned a product surrogate key with a value of 1, the next product row is assigned product key 2, + so forth
* The **actual surrogate key value has no business significance** + it **merely serves to JOIN the dimension tables to the fact table**
* In this book, **column names with a “Key” suffix**, identified as a PK or FK, **imply a surrogate**
* Modelers sometimes are reluctant to relinquish the natural keys because they want to navigate the fact table based on the operational code while avoiding a JOIN to the dimension table
* They also don’t want to lose embedded intelligence that’s often part of a natural multipart key
* However, **avoid relying on intelligent dimension keys because any assumptions you make eventually may be invalidated**
* Likewise, **queries and data access applications should NOT have any built-in dependency on the keys because the logic also would be vulnerable to invalidation**
* **Even if the natural keys appear to be stable + devoid of meaning, don’t be tempted to use them as the dimension table’s PK**
* **NOTE**: **Every JOIN between dimension and fact tables in the DW should be based on meaningless integer surrogate keys**
* You should **avoid using a natural key as the dimension table’s PK**
* **Initially, it may be faster to implement a dimensional model using operational natural keys, but surrogate keys pay off in the long run**
* Think of like a flu shot for the DW 🡪 like an immunization, there’s a **small amount of pain to initiate + administer surrogate keys, but the long run benefits are substantial, *especially considering the reduced risk of substantial rework***
* Here are **several advantages of surrogate keys**:
* **Buffer the DW from operational changes**
* Surrogate keys enable the DW team to maintain control of the DW/BI environment rather than being whipsawed by operational rules for generating, updating, deleting, recycling, + reusing production codes
* In many organizations, historical operational codes, such as inactive account numbers or obsolete product codes, get reassigned after a period of dormancy
* If account numbers get recycled after 12 months of inactivity, the operational systems don’t miss a beat because their business rules prohibit data from hanging around for that long
* But the DW/BI system may retain data for *years*
* Surrogate keys provide the DW with a mechanism to differentiate these 2 separate instances of the same operational account number
* If you rely solely on operational codes, you might also be vulnerable to key overlaps in the case of an acquisition or consolidation of data
* **Integrate multiple source systems**
* Surrogate keys enable the DW team to integrate data from multiple operational source systems, even if they lack consistent source keys by using a **back-room cross-reference mapping table** to link the multiple natural keys to a common surrogate
* **Improve performance**
* The **surrogate key is as small an integer as possible while ensuring it will comfortably accommodate the future anticipated cardinality (number of rows in the dimension)**
* Often the operational code is a bulky alphanumeric character string or even a group of fields
* The **smaller surrogate key translates into smaller fact tables, smaller fact table indexes, + more fact table rows per block input-output operation**
* Typically, a 4-byte integer is sufficient to handle most dimensions
* A 4-byte integer is a *single* integer, not 4 decimal digits
* It has 32 bits + therefore can handle approximately 2 billion positive values (232) or 4 billion total positive and negative values (–232 to +232)
* *This is more than enough for just about any dimension*
* Remember, if you have a large fact table with 1 billion rows of data, every byte in each fact table row translates into another GB of storage
* **Handle null or unknown conditions**
* As mentioned earlier, **special surrogate key values are used to record dimension conditions that may not have an operational code**, such as the No Promotion condition or the anonymous customer
* Can assign a surrogate key to identify these despite the lack of operational coding
* Similarly, fact tables sometimes have dates that are yet to be determined
* There is no SQL date type value for Date to Be Determined or Date Not Applicable
* **Support dimension attribute change tracking**
* One of the primary techniques for handling changes to dimension attributes relies on **surrogate keys to handle the multiple profiles for a single natural key**
* This is actually one of the most important reasons to use surrogate keys (Chapter 5)
* A pseudo-surrogate key created by simply gluing together the natural key with a time stamp is *perilous*
* You need to **avoid multiple JOINs between the dimension and fact tables** (sometimes referred to as **double-barreled joins**) due to adverse impact on performance + ease of use
* Of course, some effort is required to assign and administer surrogate keys, but it’s not nearly as intimidating as many people imagine
* You need to establish + maintain a cross-reference table in the ETL system that will be used to substitute the appropriate surrogate key on each fact and dimension table row
* See the process for administering surrogate keys in Chapter 19: ETL Subsystems and Techniques

#### Dimension Natural and Durable Supernatural Keys

* Like surrogate keys, the **natural keys**assigned + used by operational source systems **go by other names, such as business keys, production keys, and operational keys** + are identified with the NK notation in this book
* The **natural key is often modeled as an *attribute* in the dimension table**
* If the natural key comes from multiple sources, you might use a character data type that prepends a source code, such as “SAP|43251” or “CRM|6539152”
* If the same entity is represented in both operational source systems, then you’d likely have 2 natural key attributes in the dimension corresponding to both sources
* **Operational natural keys are often composed of meaningful constituent parts**, such as the product’s line of business or country of origin
* ***These components should be split apart and made available as separate attributes***
* **In a dimension table with attribute change tracking, it’s important to have an identifier that uniquely and reliably identifies the dimension entity across its attribute changes**
* Although the operational natural key may seem to fit this bill, **sometimes the natural key changes due to unexpected business rules (like an organizational merger) or to handle either duplicate entries or data integration from multiple sources**
* **If the dimension’s natural keys are *NOT* absolutely protected + preserved over time, the ETL system needs to assign permanent durable identifiers, also known as** **supernatural keys**
* **A persistent durable supernatural keyis controlled by the DW/BI system + remains immutable for the life of the system**
* Like the dimension surrogate key, it’s **a simple integer sequentially assigned**, + like natural keys, the durable supernatural key is **handled as a dimension attribute**
* It’s ***NOT* a replacement for the dimension table’s surrogate PK**
* Chapter 19 also discusses the ETL system’s responsibility for these durable identifiers

#### Degenerate Dimension Surrogate Keys

* Although **surrogate keys aren’t typically assigned to degenerate dimensions, each situation needs to be evaluated to determine if one is required**
* **Surrogate key = necessary if the transaction control numbers are *not* unique across locations or get reused**
* Ex: A retailer’s POS system may not assign unique transaction numbers across stores, or it may wrap back to 0 and reuse previous control numbers when its maximum has been reached
* Also, the transaction control number may be a bulky 24-byte alphanumeric column
* Finally, depending on the capabilities of the BI tool, you **may need to assign a surrogate key** (and create an associated dimension table) **to drill *across* on the transaction number**
* Obviously, **control number dimensions modeled in this way with corresponding dimension tables are no longer degenerate**

#### Date Dimension Smart Keys

* As noted, the **date dimension has unique characteristics and requirements**
* **Calendar dates are fixed + predetermined** (never need to worry about deleting dates or handling new, unexpected dates on the calendar)
* **Because of its predictability, you can use a more *intelligent* key for the date dimension**
* **If a sequential integer serves as PK of the date dimension, it should be chronologically assigned**
* i.e., January 1 of the 1st year would be assigned surrogate key value 1, January 2 would be assigned surrogate key 2, February 1 would be assigned surrogate key 32, + so on
* **More commonly, the PK of the date dimension is a meaningful integer formatted as “yyyymmdd”**
* A yyyymmdd key is *NOT* intended to provide business users + their BI applications with an intelligent key so they can bypass the date dimension + directly query the fact table
* **Filtering on a fact table’s yyyymmdd key = detrimental impact on usability + performance**
* **Filtering + grouping on calendar attributes should occur *in a dimension table*, not in the BI application’s code**
* However, the **yyyymmdd key is useful for partitioning *fact* tables**
* **Partitioning** **enables a table to be segmented into smaller tables under the covers**
* **Partitioning a large fact table on the basis of date is effective because it allows old data to be removed gracefully + new data to be loaded + indexed in the current partition without disturbing the rest of the fact table, which reduces the time required for loads, backups, archiving, + query response**
* **Programmatically updating + maintaining partitions is straightforward if the date key is an ordered integer** (year increments by 1 up to the number of years wanted, month increments by 1 up to 12, + so on)
* **Using a smart yyyymmdd key provides the benefits of a surrogate, + the advantages of easier partition management**
* Although the yyyymmdd integer is the most common approach for date dimension keys, **some RDB optimizers prefer a *true* date type column for partitioning**
* In these cases, the optimizer knows there are 31 values between March 1 and April 1, as opposed to the apparent 100 values between 20130301 and 20130401
* Likewise, it understands there are 31 values between December 1 and January 1, as opposed to the 8,900 integer values between 20121201 and 20130101
* **This intelligence can impact the query strategy chosen by the optimizer + further reduce query times**
* **If the optimizer incorporates date type intelligence, it should be considered for the date key**
* **If the only rationale for a date type key is simplified administration for the DBA, you can feel less compelled.**
* With more intelligent date keys, whether chronologically assigned or a more meaningful yyyymmdd integer or date type column, you need to **reserve a special date key value for the situation in which the date is unknown when the fact row is initially loaded**

#### Fact Table Surrogate Keys

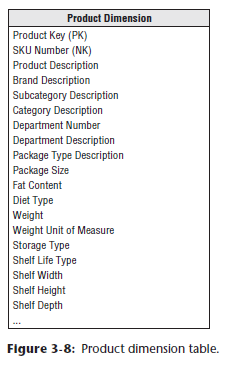
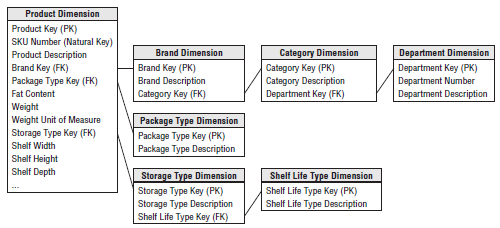
* Although adamant about **using surrogate keys for dimension tables**, we’re **less demanding about a surrogate key for fact tables**, which **typically only make sense for back room ETL processing**
* As mentioned, the **PK of a fact table typically consists of a subset of the table’s FK and/or degenerate dimension**
* However, **single column surrogate keys for fact tables have some interesting back-room benefits**
* Like its dimensional counterpart, a fact table surrogate key is a **simple integer, devoid of any business content, that is assigned in sequence as fact table rows are generated**
* Although a fact table surrogate key is **unlikely to deliver query performance advantages, it does have the following benefits:**
* **Immediate unique identification**
* A single fact table row is immediately identified by the key
* During ETL processing, a **specific row can be identified without navigating multiple dimensions.**
* **Backing out or resuming a bulk load**
* If a large number of rows are being loaded w/ sequentially assigned surrogate keys, + the process halts before completion, a DBA can determine exactly where the process stopped by finding the maximum key in the table
* The DBA could back out the complete load by specifying the range of keys just loaded or perhaps could resume the load from exactly the correct point
* **Replacing updates with inserts plus deletes**
* The fact table surrogate key becomes the *true* physical key of the fact table
* No longer is the key of the fact table determined by a set of dimensional FKs, at least as far as the RDBMS is concerned
* Thus, it becomes possible to replace a fact table UPDATE operation with an INSERT followed by a DELETE
* 1st step = place the new row into the database with all the same business FKs as the row it is to replace
* This is now possible because the key enforcement depends only on the surrogate key, + the replacement row has a new surrogate key
* 2nd step = delete the original row, thereby accomplishing the update
* **For a large set of updates, this sequence is more efficient than a set of true UPDATE operations**
* Insertions can be processed with the ability to back out or resume the insertions as described previously
* These insertions do *NOT* need to be protected with full transaction machinery
* Then the final deletion step can be performed safely because the insertions have run to completion
* **Using the fact table surrogate key as a parent in a parent/child schema**
* In cases in which one fact table contains rows that are parents of those in a *lower grain* fact table, the fact table surrogate key in the parent table is also exposed in the child table
* The argument of using the fact table surrogate key in this case rather than a natural parent key is similar to the argument for using surrogate keys in dimension tables
* **i.e., Natural keys are messy + unpredictable, whereas surrogate keys are clean integers + are assigned by the ETL system, not the source system**
* Of course, in addition to including the parent fact table’s surrogate key, the lower grained fact table should also include the parent’s dimension FKs so the child facts can be sliced + diced without traversing the parent fact table’s surrogate key
* ***NEVER join fact tables directly to other fact tables* (Chapter 4)**

### Resisting Normalization Urges

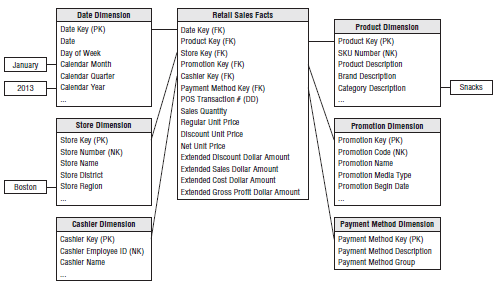
* There are several of the natural urges that tempt modelers coming from a more normalized background
* We’ve been consciously breaking some traditional modeling rules because we’re focused on **delivering value through ease of use and performance, not on transaction processing efficiencies**

#### Snowflake Schema with Normalized Dimensions

* The **flattened, denormalized dimension tables with repeating textual values** make data modelers from the operational world uncomfortable
* Let’s revisit the case study product dimension table: The 300,000 products roll up into 50 distinct departments.
* Rather than redundantly storing the 20-byte department description in the product dimension table, modelers with a normalized upbringing want to store a 2-byte department code + create a new department dimension for the department decodes
* In fact, they’d feel more comfortable if ALL descriptors in the original design were normalized into separate dimension tables
* They *argue this design saves space because the 300,000-row dimension table only contains codes, not lengthy descriptors*
* Also, some modelers contend that more normalized dimension tables are easier to maintain
* If a department description changes, they’d need to update only the 1 occurrence in the department dimension rather than the 6,000 repetitions in the original product dimension
* Maintenance often is addressed by normalization disciplines, but **all this happens back in the ETL system long before the data is loaded into a presentation area’s dimensional schema**
* **Dimension table normalization** is referred to as **snowflaking**
* **Redundant attributes are removed from the flat, denormalized dimension table + placed in separate, *normalized* dimension tables**
* The figure below illustrates the partial snowflaking of the product dimension into 3NF form

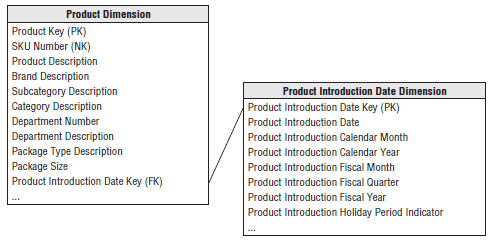
* The contrast between Figure 3-15 and Figure 3-8 is startling
* The plethora of snowflake-d tables (even in this simplistic example) is overwhelming.
* Imagine the impact on the figure below if all the schema’s hierarchies were normalized



* Snowflaking is a legal extension of the dimensional model, however, **resist the urge to snowflake given the 2 primary design drivers: ease of use and performance:**
* **The multitude of snowflake-d tables makes for a much more complex presentation**
* Business users inevitably will struggle with the complexity, and simplicity is one of the primary objectives of a dimensional model.
* **Most database optimizers also struggle with the snowflake-d schema’s complexity**.
* **Numerous tables + JOINs usually translate into slower query performance**
* The **complexities of the resulting JOIN specifications increase the chances that the optimizer will get sidetracked and choose a poor strategy**.
* **The minor disk space savings associated with snowflake-d dimension tables are insignificant**
* If you replace the 20-byte department description in the 300,000 row product dimension table with a 2-byte code, you’d save a whopping 5.4 MB (300,000 x 18 bytes), and meanwhile, you may have a 10 GB fact table!
* **Dimension tables are almost always geometrically smaller than fact tables**
* **Efforts to normalize dimension tables to save disk space are usually a waste of time.**
* **Snowflaking negatively impacts the users’ ability to browse within a dimension**
* Browsing often involves constraining 1+ dimension attributes + looking at the distinct values of another attribute in the presence of these constraints
* **Browsing allows users to understand the relationship between dimension attribute values**
* Obviously, a snowflake-d product dimension table responds well if you just want a list of the category descriptions
* However, if you want to see all brands within a category, you need to traverse the brand *and* category dimensions
* If you want to also list the package types for each brand in a category, you’d be traversing *even more* tables
* The SQL needed to perform these seemingly simple queries is complex, + you haven’t touched the other dimensions or fact table
* **Snowflaking defeats the use of bitmap indexes**
* **Bitmap indexes** **are useful when indexing low-cardinality columns**, such as the category and department attributes in the product dimension table
* They **greatly speed performance of a query or constraint on the single column in question**.
* **Snowflaking inevitably would interfere with your ability to leverage this performance tuning technique**
* **NOTE**: **Fixed-depth hierarchies should be flattened in dimension tables**
* **Normalized, snowflake-d dimension tables penalize cross-attribute browsing + prohibit the use of bitmapped indexes**
* **Disk space savings gained by normalizing the dimension tables typically are < 1% of the total disk space needed for the overall schema**
* You should knowingly **sacrifice this dimension table space in the spirit of performance and ease of use advantages.**
* Some database vendors argue their platform has the horsepower to query a fully normalized dimensional model without performance penalties
* If you can achieve satisfactory performance without physically de-normalizing the dimension tables, that’s fine
* However, you’d **still want to implement a logical dimensional model w/ denormalized dimensions to present an easily understood schema to business users + their BI applications**
* In the past, some BI tools indicated a preference for snowflake schemas, + **snowflaking to address the idiosyncratic requirements of a BI tool is acceptable**
* Likewise, if all data is delivered to business users via an OLAP cube (where the snowflake-d dimensions are used to populate the cube but are *never visible to the users*), then snowflaking is acceptable
* However, in these situations, you need to **consider the impact on users of alternative BI tools and the flexibility to migrate to alternatives in the future**

#### Outriggers

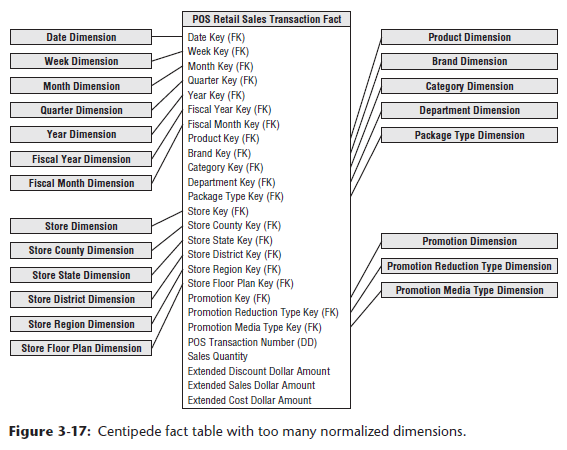
* Although we generally do not recommend snowflaking, there *are* situations in which it is permissible to build an **outrigger dimension**that **attaches to a dimension within the fact table’s immediate halo,** as illustrated below



* In this example, the “once removed” **outrigger** is a date dimension snowflake-d off a primary dimension
* The outrigger date attributes are descriptively + uniquely labeled to distinguish them from the other dates associated with the business process
* **It only makes sense to outrigger a primary dimension table’s date attribute if the business wants to filter and group this date by nonstandard calendar attributes**, such as the fiscal period, business day indicator, or holiday period
* Otherwise, you could just treat the date attribute as a standard date type column in the product dimension
* If a date outrigger is used, be careful that the outrigger dates fall within the range stored in the standard date dimension table.
* More outrigger examples are in this book (like in Chapter 8: Customer Relationship Management)
* **Although outriggers may save space and ensure the same attributes are referenced consistently, there are downsides**
* Outriggers **introduce more JOINs**, which can **negatively impact performance**
* More important, outriggers **can negatively impact the legibility for business users + hamper their ability to browse among attributes within a single dimension**
* **WARNING: Though outriggers are permissible, a dimensional model should not be *littered* with outriggers given the potentially negative impact, i.e., Outriggers should be the exception rather than the rule**

#### Centipede Fact Tables with Too Many Dimensions

* The **fact table in a dimensional schema is naturally highly normalized and compact**
* There is **no way to further normalize the extremely complex many-to-many relationships among the keys in the fact table because the dimensions are not correlated with each other**
* Every store is open every day, + sooner or later, almost every product is sold on promotion in most or all of our stores.
* Interestingly, while uncomfortable with denormalized dimension tables, **some modelers are tempted to *DE*-normalize the fact table**
* They have an uncontrollable urge to normalize dimension hierarchies but know snowflaking is highly discouraged, so the normalized tables end up joined to the fact table instead
* Rather than having a single product FK on the fact table, they include FKs for the frequently analyzed elements on the product hierarchy, such as brand, category, + department
* Likewise, the date key suddenly turns into a series of keys joining to separate week, month, quarter, + year dimension tables
* Before you know it, a compact fact table turns into an monster that joins to literally dozens of dimension tables, designs referred to as **centipede fact tables**because they appear to have nearly 100 legs, as shown below:



* **Even with its tight format, the fact table is the behemoth in a dimensional model**.
* **Designing a fact table with too many dimensions leads to significantly increased fact table disk space requirements**
* **Although denormalized dimension tables consume extra space, fact table space consumption is a concern because it is your largest table by orders of magnitude**
* **There is no way to index the enormous, multi-part key effectively in the centipede example**
* The **numerous JOINs are an issue for both usability and query performance**
* **Most business processes can be represented with less than 20 dimensions in the fact table**
* If a design has 25+ dimensions, **look for ways to combine correlated dimensions into a single dimension**
* Perfectly correlated attributes, such as the levels of a hierarchy, as well as attributes with a reasonable statistical correlation, should be part of the same dimension
* **It’s a good decision to combine dimensions when the resulting new single dimension is noticeably smaller than the Cartesian product of the separate dimensions**
* **NOTE: A very large number of dimensions typically are a sign that several dimensions are not completely independent and should be combined into a single dimension**
* **It is a dimensional modeling mistake to represent elements of a *single* hierarchy as separate dimensions in the fact table**
* Developments w/ **columnar databases** **may reduce the query + storage penalties associated with wide centipede fact table designs**
* Rather than storing each table row, a **columnar database stores each table column as a contiguous object that is heavily indexed for access**
* Even though the underlying physical storage is columnar, at the query level, the table appears to be made up of familiar rows
* But when queried, only the *named* columns are actually retrieved from the disk, rather than the entire row in a more conventional row-oriented RDB
* **Columnar databases are much more tolerant of the centipede fact tables just described**
* ***However*, the ability to browse across hierarchically related dimension attributes may be compromised**