Paper 107-2007

Demeter in the Database

Paul D Sherman, Independent Consultant, San Jose, CA

ABSTRACT

Relational database normal form is nothing more than variable dereference. Used properly by following the two most important rules of object-oriented design, separation and encapsulation, one's database model will always work well. Violating the Law of Demeter enforces a concrete implementation on an otherwise abstract model, effectively multiplying in Cartesian fashion weakly related variables and needlessly increasing table object size with redundant information. Instead, refer to table objects no more than one level away from each other. Empower each level in the model hierarchy with the privilege and responsibility it deserves, giving each table suitably named verbs (methods), behavior and characteristics (fields). In the following example, an idea of Consumption & Replenish, we will create a dynamically adapting database model, helping Farmer Jack predict when he needs to order more fertilizer to feed each of his many beanstalks. This model is easily generalized to forecasting demand or managing the supply chain.
Skill Level: Intermediate to Advanced, familiarity with SAS/Access and Proc SQL pass-through operation.

INTRODUCTION

Proc SQL offers the SAS® programmer a powerful tool for handling datasets. Even more useful is the pass-through extension allowing direct communication to an external database system directly from within a SAS program. With a few simple steps in mind, you can easily design SQL statements which maximize use of the database, while minimizing load and processing time on the local workstation.

Why will we study only the `SELECT` statement? Although SQL is rich with both content inquiring and content affecting transactions, we focus herein on the former since fullselect grammar is much more complex, more routinely utilized, and can often be applied as an argument value to the latter. Much insight will be gained from mastery of the database inquire transaction or `SELECT` statement.

In this article, we present the problem of content redundancy, which is a desire to know too much too soon by too many rather than leaving internal details to the responsible objects. We then review "flat" and normalized table designs, discuss optimal normal forms, and compare the relational model with object-oriented style.

THE LAW OF DEMETER

Demeter is the goddess of agriculture. The abduction of her beautiful daughter, Persephone, by Hades to the underworld explains the change of seasons. Winter happens as Demeter wanders withdrawn in loneliness searching far and wide and can find no trace of her daughter. When Zeus, Demeter's brother, sends Hermes to persuade Hades to allow Persephone to rejoin her mother, Demeter is happy and lets the plants blossom, bloom and grow again. As a gift, Hades gives Persephone a piece of fruit to enjoy while away, which binds her to return back to the underworld for a third of each year.

Growth of agriculture is symbolic of a bottom-up, or abstract-to-concrete, style of program design. The small steps such as birth, growth, and harvest are what object-oriented programmers call the Law of Demeter. It is a "principle of least knowledge," leading to program designs with minimal coupling between objects. In the database or data warehousing world, this relates to a minimum of information stored in each table of rows and columns. By not repeating information, data is easier to change (there's only one copy), and storage space is more efficiently utilized.

GENEALOGICALLY SPEAKING

You are allowed only to talk to your immediate siblings, children, and parents, and no other younger or older generation. This is shown in Figure 1 below. It makes sense from a linguistic point of view, because modern teenage vernacular gives grandparents a headache. To communicate with the other generations, you play the familiar game of "telephone" and let hierarchical neighbors pass the word. Let Dad do the talking to Grandma, and daughter talk to Grandson.

This way, as Grandson follows each fashion craze of jeans, sneakers, scooters, roller skates, and electronic games, Grandpa has only to remember the yearly birthday check to Mom.

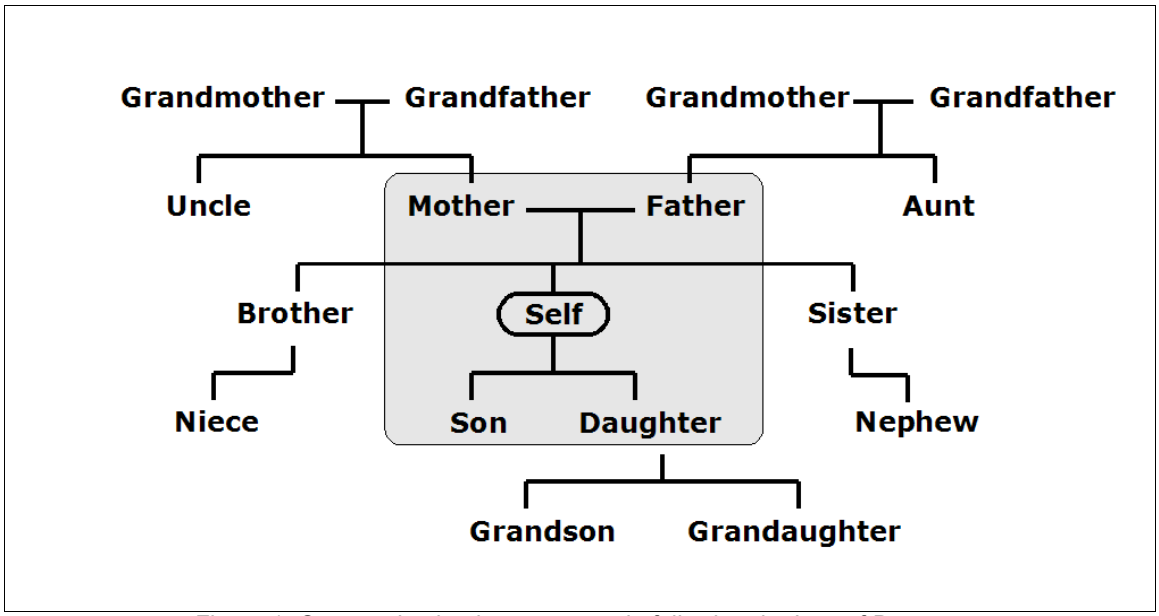


Figure 1: Communication between people following the Law of Demeter

LUNCH TIME

Said another way,

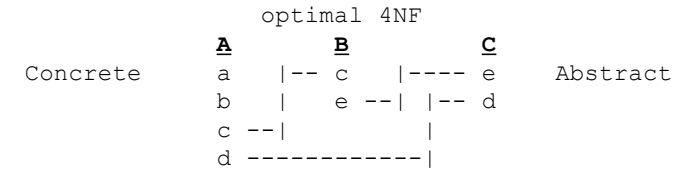
> *"... if you have a method that takes a "BigMac" as an argument and needs to access the seeds on the pickles, you don't have to know about the lettuce, onions, and sesame seed bun that are all "on route" to the pickle seeds (much less that there are 2 all beef patties lurking underneath all of that), all you need to know is that a path exists from the BigMac to its pickle seeds. You don't have to know what that path is or exactly how to traverse it, and you don't have to encode that path in your methods. That way when the burger later changes such that the pickles are now on top of the onions instead of the other way around, you don't have to care." (B. Appleton, "Introducing Demeter and its Laws").*

TABLE DESIGNS AND THE OPTIMAL NORMAL FORM

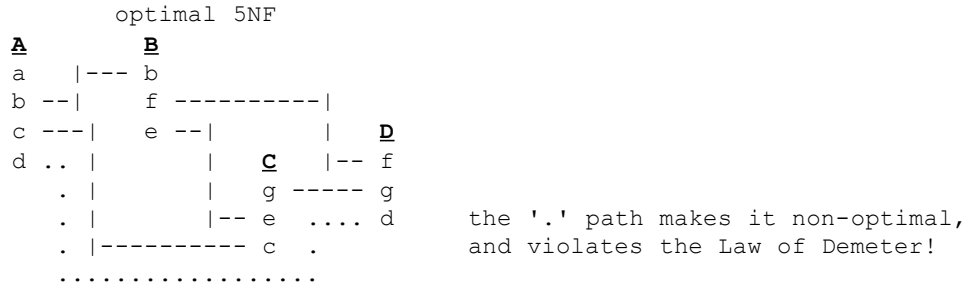
Flatten out a third normal form (3NF) into optimal 2NF in order to maintain the fewest possible relations (tables) consistent with original intent of the unique keys. Simply put, don't separate information and break relations down too far unless a dependency *increases* the level of abstraction..

```
3NF       optimal 2NF          optimal 3NF
A B            Q                  A       B
a a            a                  a    |-- b
b c            b                  b --|   c
               c
```

Tables lower in hierarchy are more abstract, similar to de-referencing and indirect addressing. This promotes low cohesion of information.



Optimal nNF will always have (n-1) tables. Non-optimal relations span more than one table.



WHAT CAN WE LEARN FROM OBJECT-ORIENTED STYLE?

Separate – remove information which doesn't belong together
Encapsulate – collect information which does

FLAT TABLE DESIGN

+ One db-row observation per key set
- New attribute column or variable requires alter table
- Redundant column storage of common attributes
- Cannot add new item having previously unknown attribute
- Attribute redefinition need update all existing records

BEAN.BAG
stalk
name
size
shape
weight
color
texture
flavor

NORMALIZED TABLE DESIGN

- Many db-rows per key set
+ Easily extended for new attributes
- Redundant row storage of common attributes
+ Can add new item with yet undefined attribute
- Attribute redefinition need update some existing records

BEAN.BAG
stalk
name
attrib
value
flavor

Size
Shape
Weight
Color
Texture

ENCAPSULATED TABLE DESIGN

+ One db-row per key set per table
+ Easily extended for new attributes
+ All implementations inherit related table changes
+ Minimal/optimal storage of unique attributes
+ Can add new item with yet undefined attribute
+ Attribute redefinition only updates rows of attribute table.

BEAN.BAG
stalk
name
flavor
look_id
feel_id

BEAN.LOOK
look_id
attrib
value

BEAN.FEEL
feel_id
attrib
value

Size
Shape
Color

Weight
Texture

In summary,

	Flat	Normalized	Encapsulated
db-rows per key set:	one	many	one per table
Storage of attributes:	redundant	redundant	minimal
Add new item:	no	yes	yes
Attribute redefinition:	update all records	update some records	update attrib rows

EXAMPLE – CONSUMPTION & REPLENISH

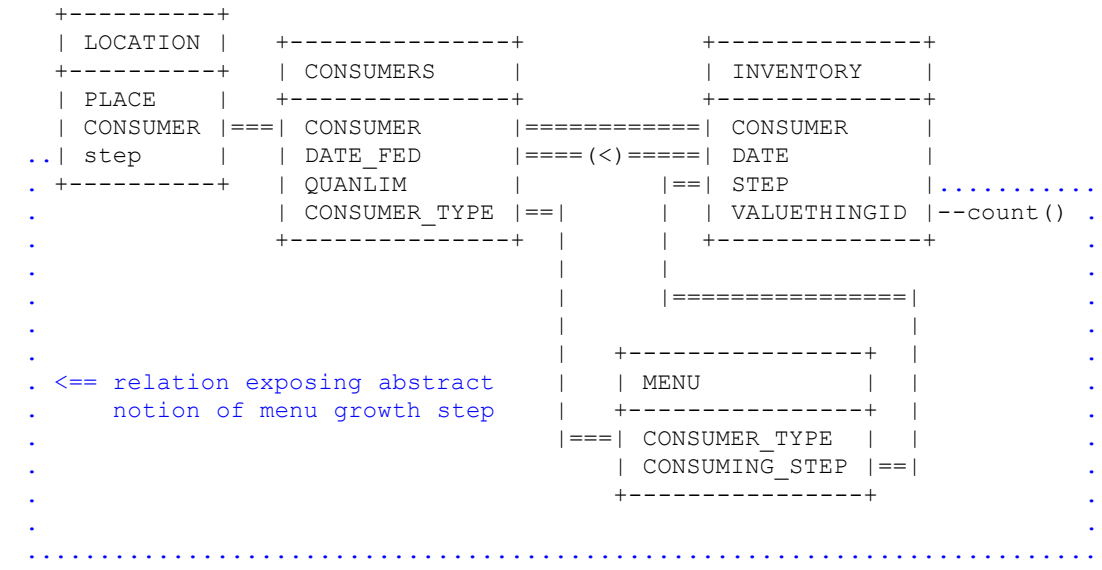
Let us think about a model of Consumption & Replenish. Suppose a place has many consumers where each consumer eats up consumables and as a result, produces something of value. The 'consumable-to-valuething' ratio might then be like a unit cost. If you like Jack and the beanstalk, then maybe consumables are the fertilizer and plant-food and water, consumers are the beanstalks and, of course, what is produced are the beans. The places are simply geographical location: San Francisco, Seattle, Montreal, Orlando, etc., where ever the consumers are located.

Our problem statement might be this: For every consumer at each place accumulate the number of valuable things processed by the consumers. Report only those consumers where the total consumable count exceeds a predetermined limit, which need not be the same for each consumer. Regardless of total consumable count, detail the status for all consumers once a week.

Think about helping farmer Jack predict when he needs to order more fertilizer to feed each of his beanstalks.

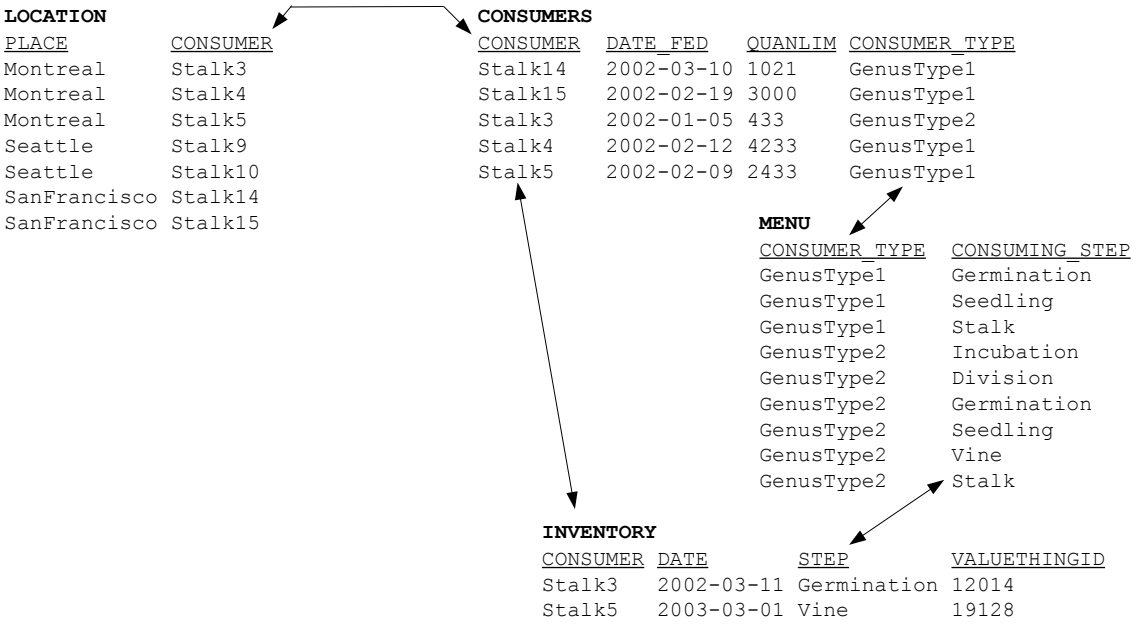
In our model, beanstalks (things) are fed their fertilizer on certain dates. Once fed, they can produce only a limited quantity of beans (value-things) before needing to be fed again. There are many kinds (genuses) of beanstalks, having different numbers of growth steps that their product beans must surpass. The inventory is a log of each stage of growth (when and where) of every product bean produced. Query efficiency is of utmost importance because this inventory or history table is usually very large, having millions of rows.

Therefore, it is a simple matter to "count the total number of value-things logged since last consumer feeding, as a function of consumers that produced them, and compare the total quantity to the hunger/satisfaction limit of each consumer." In other words, when a beanstalk (consumer) runs out of food, it can't produce any more beans. Let us assume that beanstalks at each place consume fertilizer at the same rate.

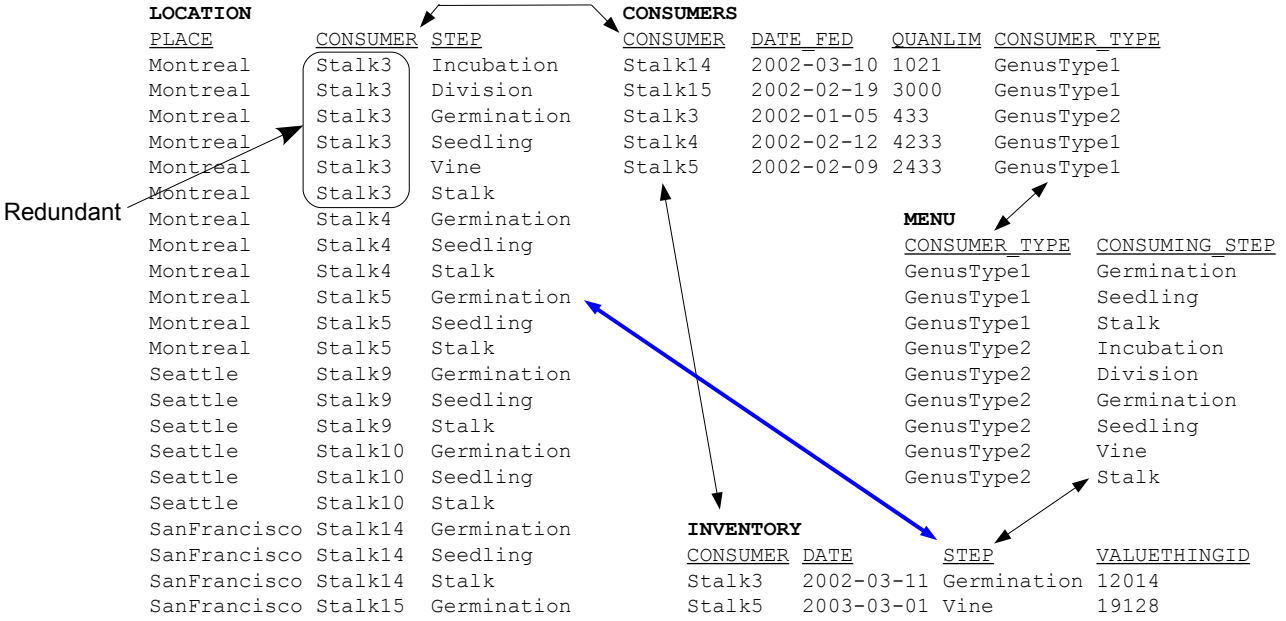


The (...) relationship and the expression highlighted in the SQL statement below violates the Law of Demeter, since it exposes the abstract notion of a consumer growth step to the concrete location of beanstalk place. It's no business of a place of many beanstalks (would you call this a field?) to know about internal details of how many steps it takes a bean to grow on its beanstalk.

First let's look at the situation *without* the (...) relationship. Tables and their data are shown in the Appendix.



When we violate the Law of Demeter, such as by taking away the Menu abstraction, we effectively perform a Cartesian multiplication of information between Location and Menu. Therefore, the Location object increases needlessly in size with redundant information.



It is clear that by violating the Law of Demeter and enforcing implementation of Location on our abstract concept of growth step, that not only is the location table below unnecessarily large, but also there is a "loss of liberty" to modify growth step progression of a beanstalk genus, such as by genetic plant research, without also affecting Location.

The SQL query statement which implements our model is shown below. Each relation-expression in the FROM clause is a line drawn on the query model diagram above. Notice that it is difficult to spot the expression which violates the Law of Demeter – it looks just like any other expression. This is why it is very important to sketch out the table model diagram, as shown above, when creating or revising any SQL query.

```
SELECT place, consumer, step, sum(n), max(nlim)
FROM (
   SELECT location.place,
          location.consumer,
          inventory.step,
          count(inventory.valuethingid) as n,
          max(consumers.quanlim) as nlim
   FROM location
        INNER JOIN consumers ON location.consumer = consumers.consumer
        INNER JOIN menu ON consumers.consumer_type = menu.consumer_type
        INNER JOIN inventory ON consumers.consumer = inventory.consumer
                            AND menu.consuming_step = inventory.step
                            AND consumers.date_fed < inventory.date
                            AND location.step = inventory.step
   GROUP BY location.place,
            location.consumer,
            inventory.step
     ) as foo (place, consumer, step, n, nlim)
GROUP BY place, consumer
HAVING sum(n) > max(nlim)
       or dayofweek(current timestamp) = 5
```

Demeter violating expression

Our query returns the following result, which indicates that three of the beanstalks have exceeded their production quota and need to be fed soon.

PLACE	CONSUMER	NUMBEANS	QUANLIM
Montreal	Stalk3	540	433
Montreal	Stalk5	2481	2433
SanFrancisco	Stalk14	1079	1021

WHAT MIGHT GO WRONG

Suppose your database doesn't have the abstract Menu table, and furthermore (like most of us) you don't have create-table authority. In this case you can use a `VALUES` list to create any table you like on-the-fly in your SQL statement.

```
FROM ...
     inner join (
     values
        ('GenusType1', 'Germination', 'Seedling', 1),
        ('GenusType1', 'Seedling', 'Stalk', 2),
        ('GenusType1', 'Stalk', null, 3),
        ('GenusType2', 'Division', 'Germination', 2),
        ('', '', '', 0)
     ) as menu (consumer_type, consuming_step, next_step, step_num) ON ...
```

Column type declarator:
Sets column count,
Defines column types,
Satisfies "one row, all columns not null"

This temporary table will exist for the entire duration of the sub-query of which it is in scope, then will be automatically deleted when the query ends. Temporary tables created in this manner are usually short because they are abstract. You wouldn't code an entire process history like this, but rather convince your dba to actually create the physical base table.

Unfortunately, many databases do not support the VALUES keyword in the FROM clause of an SQL statement. In this situation, you would use a "dummy" table, such as the DUAL object as follows:

```
FROM ...
     inner join (
       SELECT '' as c_type, '' as c_step, '' as next_step, 0 as step_num FROM dual
       UNION SELECT 'GenusType1', 'Germination', 'Seedling', 1       FROM dual
       UNION SELECT 'GenusType1', 'Seedling', 'Stalk', 2             FROM dual
       UNION SELECT 'GenusType1', 'Stalk', cast(null as char(8)), 3  FROM dual
       UNION SELECT 'GenusType2', 'Division', 'Germination', 2       FROM dual
     ) menu ON ...
```

Column type declarator line

In PROC SQL there is no DUAL object. You can simply use any small system table such as sashelp.class and specify the OBS option so that you get only one row returned. For example,

```
       UNION SELECT 'GenusType1', 'Stalk', '', 3 FROM sashelp.class (OBS=1)
```

Notice also that we must use a missing value of appropriate type in place of the `null` value.

CONCLUSION

Separating and encapsulating information provides enormous benefit when making revisions or additions to data content. Relying upon only the optimal relational normal forms allows you to build a warehouse of lookup tables or cross-reference sheets which, paralleling variable dereference from other structured languages, is the most space efficient, fastest manner of information access. Using the `VALUES` list in a pinch, when a database doesn't have a needed abstraction layer and you don't have create authority, often quickly solves many data modeling and access problems. Creating relationships only among closest siblings, children and parents keeps inter-table cohesion low, enhancing usability and revision, and is the true essence of the Law of Demeter.

REFERENCES

"Demeter". Encyclopedia Mythica. <www.pantheon.org/articles/d/demeter.html>

Appleton, Brad, "Introducing Demeter and its Laws." <www.cmcrossroads.com/bradapp/docs/demeter-intro.html>

Bock, David. "The Paperboy, The Wallet, and the Law of Demeter." <http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf>

Date, C.J. (2001). The Database Relational Model: A Retrospective Review and Analysis, Reading, MA: Addison-Wesley.

Edwards, Betty. (1979). Drawing on the Right Side of the Brain. Los Angeles: J. P. Tarcher.

Lieberherr, Karl J. "Demeter: Aspect-Oriented Software Development" Boston: Northeastern University, College of Computer and Information Sciences, Center for Software Sciences.. <http://www.ccs.neu.edu/research/demeter>

Lieberherr, K. J. and I. Holland. (1989). Assuring Good Style for Object-Oriented Programs, IEEE Software, September: pp. 38-48.

Sherman, Paul D. (2002). "Creating Efficient SQL – Four Steps to a Quick Query," in *Proceedings of the Twenty-Seventh Annual SAS User Group International Conference*, Orlando, Florida. Paper p073-27. <http://www2.sas.com/proceedings/sugi27/p073-27.pdf>

ACKNOWLEDGMENTS

Paul wishes to thank colleague, mentor, and friend Thomas Poliquin for his creative and exciting introduction to the art of object oriented programming. Xiaoguang Liang deserves a lot of credit for many helpful discussions on database design.

TRADEMARK CITATION

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Paul D Sherman
Electrical Engineer
335 Elan Village Lane, Apt. 424
San Jose, CA 95134
Home: 408-383-0471
E-mail: sherman@idiom.com

Web site: http://www.idiom.com/~sherman/paul/pubs

EXERCISE

Suppose Jack's beanstalks at different places consume their fertilizer at different rates based on the weather. You rightfully decide not to alter the Location table and add a Demeter violating relation to Menu, or create new consumer type values for each different place. Instead, you define new abstraction table Weather which has two columns, place and climate:

Weather:

PLACE	CLIMATE
Montreal	cold
San Francisco	foggy
Seattle	wet
Orlando	humid

You would connect this new Weather table between Location and which other table in the data model? Justify your choice and provide sample row content.

APPENDIX

Menu:

CONSUMER_TYPE	CONSUMING_STEP
GenusType1	Germination
GenusType1	Seedling
GenusType1	Stalk
GenusType2	Incubation
GenusType2	Division
GenusType2	Germination
GenusType2	Seedling
GenusType2	Vine
GenusType2	Stalk

Location:

PLACE	CONSUMER
Montreal	Stalk3
Montreal	Stalk4
Montreal	Stalk5
Seattle	Stalk9
Seattle	Stalk10
SanFrancisco	Stalk14
SanFrancisco	Stalk15

Consumers:

CONSUMER	DATE_FED	QUANLIM	CONSUMER_TYPE
Stalk14	2002-03-10	1021	GenusType1
Stalk15	2002-02-19	3000	GenusType1
Stalk3	2002-01-05	433	GenusType2 ***
Stalk4	2002-02-12	4233	GenusType1
Stalk5	2002-02-09	2433	GenusType1

*** Stalk3 is a special type of bean plant which progresses through more growth steps (six, to be precise) thus eating more consumables and is able to produce fewer beans; note the lower bean quanlim.
The consumable to value ratio of GenusType2 is higher (i.e., it's more expensive to grow)

Inventory:

CONSUMER	DATE	STEP	VALUETHINGID
Stalk3	2002-03-11	Germination	12014
Stalk5	2003-03-01	Vine	19128