Best Practices for Semantic Data Modeling for Performance and Scalability

SQL Server Technical Article

Writer: Sharon Bjeletich
Technical Reviewer: Thomas Kejser

Published: August, 2008
Applies To: SQL Server 2008

Summary: More and more business applications are architected as business frameworks, where the core data model of the framework must support customers who work with different database objects and attributes, as well as allow for extensive customization. This paper covers some of the issues that can arise when it is difficult to decide whether to use an object-oriented or relational approach to designing the database. It includes approaches to improve performance and scalability. This paper is for database developers who are familiar with semantic modeling challenges.

Copyright

Table of Contents

Introduction

More and more business applications are architected as business frameworks, where the core data model of the framework must support customers who work with different database objects and attributes, as well as allow for extensive customization. For example, take a manufacturing company that develops an application to capture all sensor data coming from a plant's equipment. Every plant floor is potentially different, with different equipment types, sensors, readings, and needs. A plant where automobiles are made has very different equipment and sensors than a plant where chocolate bars are made.

Relational databases that are developed for these applications tend to be very object oriented since there is no real way to identify all of the data definitions at design time. Objects with attributes are commonly used. These are often called semantic models. when implemented, these very generic or "universal" data models can be complex on many levels. They are very difficult to write queries against because the "object" table is aliased over and over in a query, making the query very difficult to understand. Furthermore, cost-based optimizers have a difficult time with a database that has many self-joins. In addition, the most common data tends to be close together on disk, resulting in scalability issues.

This paper covers some of the main issues that can arise in these scenarios and some approaches to improve performance and scalability. It is targeted for database developers who are familiar with semantic modeling challenges.

The "Universal" Data Model

Most objects and transactions can be modeled by using a "universal" data model—a model of nouns, adjectives, verbs, and adverbs, if you will. The following model could be created to store just about any kind of data for any kind of application. (This example model is extremely simplistic and is for illustration purposes only.)

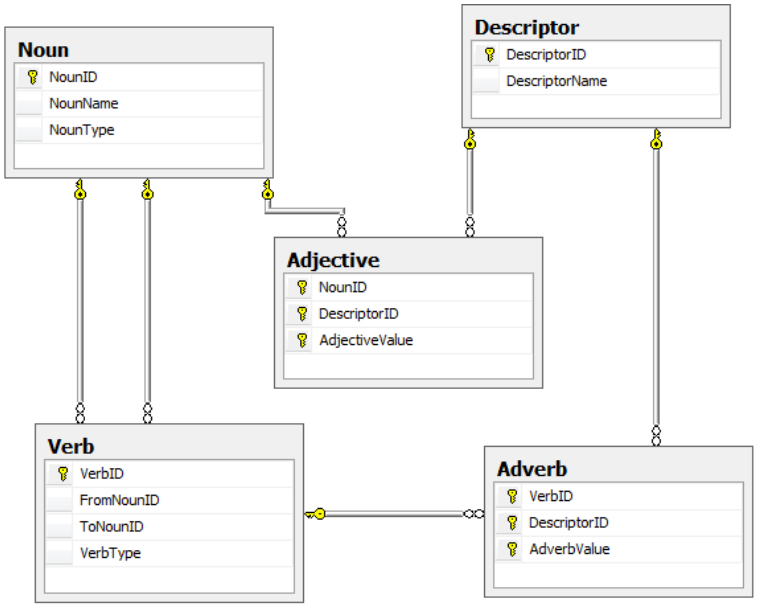

Figure 1

This model is translated as follows:

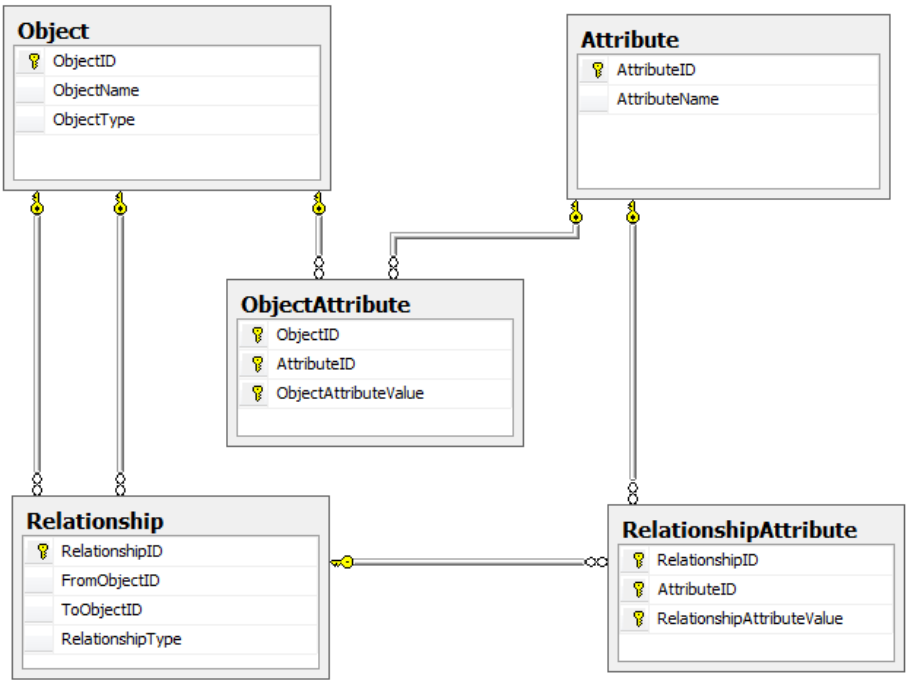

Figure 2

If this model is applied to an online book -selling application, books and stores are nouns (objects); book names and types are adjectives (attributes); the sale is the verb (relationship); and the date of sale and quantity are adverbs. This is a very simple example, and it would be common to add grouping, containers, and types, but as a foundation this model can support most applications even without specific table and column names as might be required in a traditional relational data model. Since it is data driven at implementation time, the database is essentially "runtime".

However, it is very difficult to write queries against this model. A query that returns a list of book titles sold during a particular day at a particular store—a very simple query—would look like the following:

```
select
       convert(varchar, SaleDateValue.ObjectAttributeValue, 101) as SaleDate

Store.ObjectName as StoreName
Book.ObjectName as BookName
Author.ObjectName as AuthorName
Customer.ObjectName as CustomerName
SaleBookQtyvalue.RelationshipAttributeValue as BooksSold
from Object as Sale
   join ObjectAttribute as SaleDateValue
      on Sale.ObjectID = SaleDateValue.ObjectID
   join Attribute as SaleDate
      on SaleDateValue.AttributeID =  SaleDate.AttributeID
```

```
        and SaleDate.AttributeName = 'SaleDate'
    join Relationship as SaleStore
        on Sale.ObjectID = SaleStore.FromObjectID
        and SaleStore.RelationshipType = 'SaleStore'
    join Object as Store
        on SaleStore.ToObjectID = Store.ObjectID
        and Store.ObjectType = 'Store'
    Join Relationship as SaleBook
        on Sale.ObjectID = SaleBook.FromObjectID
        and SaleBook.RelationshipType = 'SaleBook'
    join Object as Book
        on SaleBook.ToObjectID = Book.ObjectID
        and Book.ObjectType     = 'Book'
    Join Relationship as BookAuthor
        on Book.ObjectID = BookAuthor.FromObjectID
        and BookAuthor.RelationshipType = 'BookAuthor'
    join Object as Author
        on BookAuthor.ToObjectID = Author.ObjectID
            and Author.ObjectType = 'Person'
    Join Relationship as SaleCustomer
        on Sale.ObjectID = SaleCustomer.FromObjectID
        and SaleCustomer.RelationshipType = 'SaleCust'
    join Object as Customer
        on SaleCustomer.ToObjectID = Customer.ObjectID
        and Customer.ObjectType = 'Person'
    join RelationshipAttribute as SaleBookQtyvalue
        on SaleBook.RelationshipID = SaleBookQtyvalue.RelationshipID
    join Attribute as SaleBookQty
        on SaleBookQtyvalue.AttributeID = SaleBookQty.AttributeID
        and SaleBookQty.AttributeName = 'SaleQty'
where Sale.ObjectType   = 'Sale'
```

This query is complex and constrained. Many companies have failed at applications based on this type of model because of the abstraction of the objects. Although many customer scenarios can be used with this model because it is so extensible, it is almost impossible for customers to write queries and reports against it.

If you know at design time that the application is for a bookseller, the model might look like the following. The query would be easy to write and understand, and to optimize.

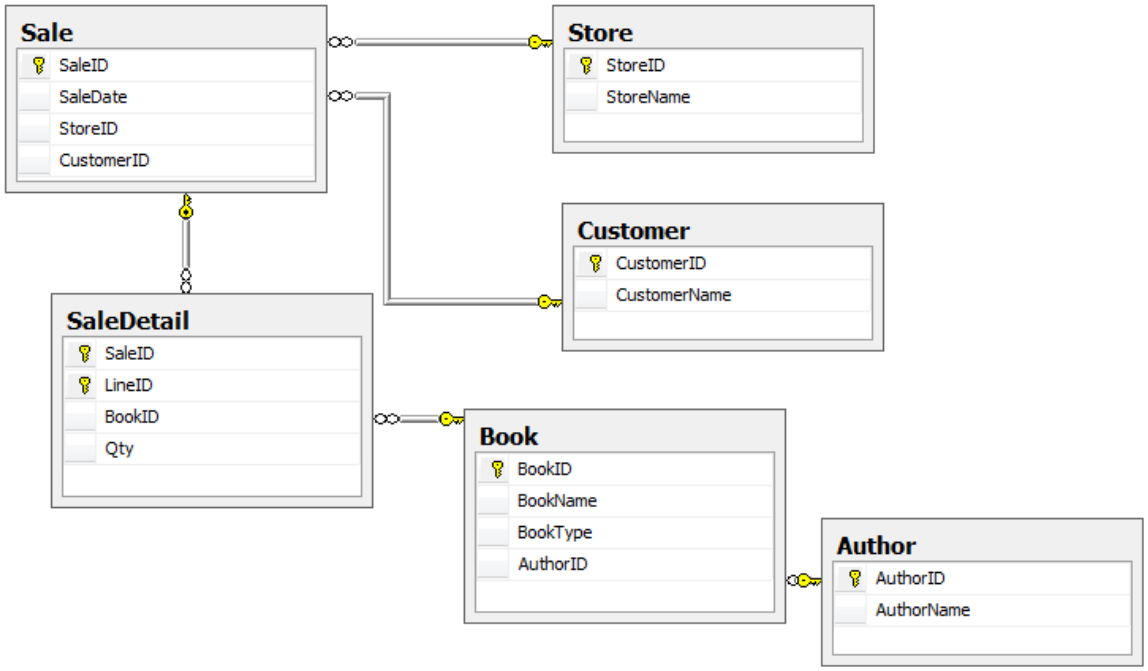

Figure 3

The same query against the above model would like more like this:

```
select
   convert(varchar,  Sale.SaleDate, 101) as SaleDate
   ,Store.StoreName
   ,Book.BookName
   ,Author.AuthorName
   ,Customer.CustomerName
   ,SaleDetail.Qty
from Sale
   join Store
      on Sale.StoreID = Store.StoreID
   join SaleDetail
      on Sale.SaleID =  SaleDetail.SaleID
   join Book
      on SaleDetail.BookID = Book.BookID
   join Customer
      on Sale.CustomerID = Customer.CustomerID
   join Author
      on Book.AuthorID = Author.AuthorID
```

Although this is much easier to understand, it is not extensible unless a customer adds columns and modifies the structure after implementation, which has obvious shortcomings.

Supertypes and Subtypes

A compromise between the two extremes exemplified by these models is necessary. That comprise will differ from customer to customer, depending on the differences in the end systems, the amount of structure that can be predetermined, and the perceived abilities of the customer.

Supertypes and subtypes are one way to allow for a more understandable data model that can still logically support the object model. A *supertype* is a construct that allows for keeping all common data in one table while splitting off the data that is significantly different into subtype tables. This enables all of the parts to be seen as one logical entity, and obtuse "object" tables become more understandable.

In the model in Figure 1, the Object table represents "nouns," such as Authors, Customers, Stores, and Books. In this example, we assume that all final customer models will need a customizable application that handles people (Authors and Customers), companies (Stores), and things (Books). Subtyping the Object table to these new tables removes a great deal of the difficulty in understanding these entities, but still allows for the flexibility and extensibility that are necessary to sell this application to any type of retailer.

To design a subtype table, we must first determine which entities are in common. At the highest level is the Object table. Supertype tables also require a discriminator column—to split the supertype table rows by object type—which usually corresponds to the subtype table name. The ObjectType column is a natural discriminator so we leave that in the supertype table.

The model for the supertype and subtype tables might now look like this:

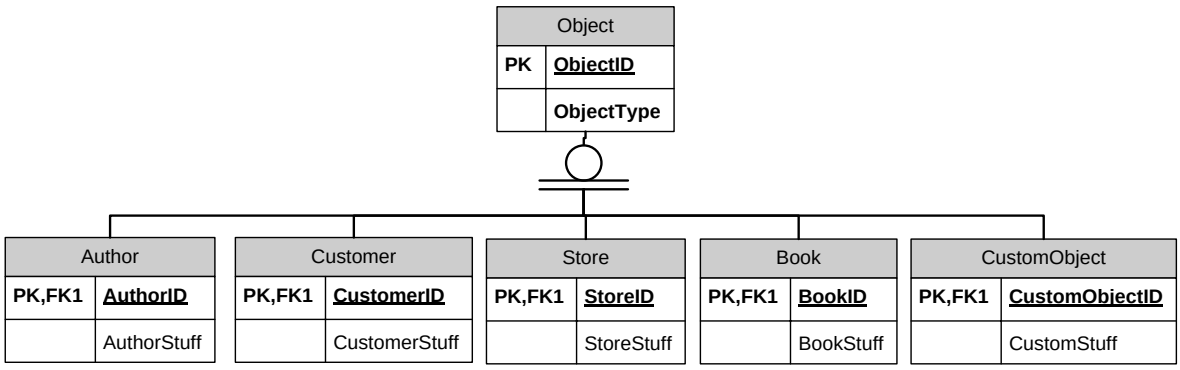

Figure 4

Each subtype table has a primary key that is actually a foreign key reference to the Object table, renamed to something that is easy to understand. The CustomObject table enables users to add other subtypes as needed. In this scenario, no Author can have the same ID as any other object. This is very important, as some data could take either one – that cannot be modeled as one column without a supertype table. For example, in a Sale table, the same person could be either a Customer or an Author.

It becomes clear that Author and Customer are not correctly modeled—these are roles, not things. The object in this scenario probably should be a Person. When a PersonID is

in the Sale table, the person is a customer, when a person is in the Book table, he or she is an author. In addition, a sale could be made to a company in addition to a person. The optimal way to model this is to have two subtype tables—one named Individual and one named Company. This is a common practice to ensure that this data can be treated interchangeably when needed and separated when that is required.

The new model looks like the following:

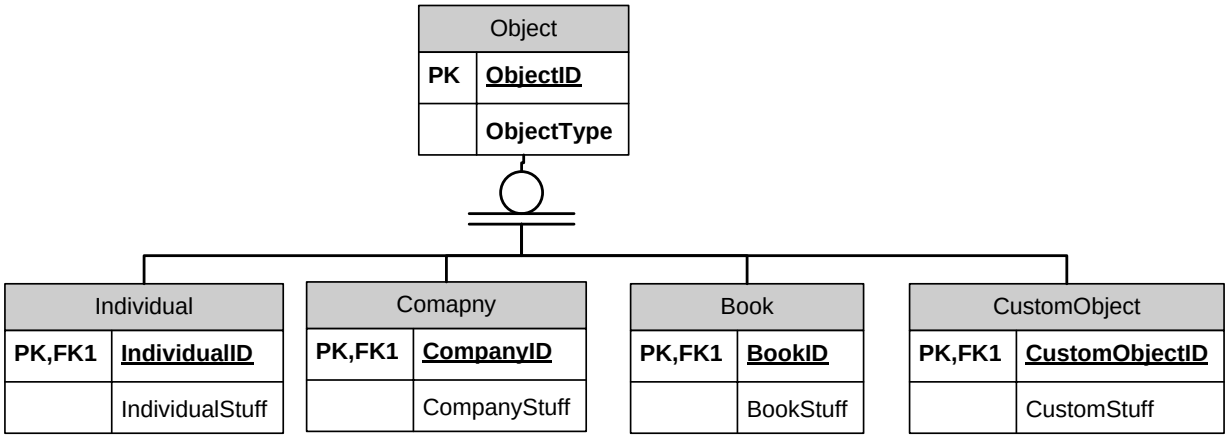

Figure 5

The Book subtype table may prove to be problematic, as the end user may sell magazines, magnifying glasses, and other products in addition to books. Changing the table to a more general Product name makes the model more usable. Supertype and subtype tables do not impose performance or scalability constraints on databases, and they allow flexibility. In some cases the Object table need not even be physically implemented. The Object name can be moved into each subtype table, and the primary keys for each subtype need to be ensured programmatically to be different across all subtypes, aligning to the logical implementation of the Object table.

Extensible Attributes

In the model, many attributes can be predetermined. For example, we can presume that most users will want to store individual names and e-mail address. These can be added as columns. The problem now becomes how users will add their own columns. There are two options to solve this and both have usability and scalability compromises.

One option is to add columns to the schema. This carries numerous risks, such as users who might delete necessary columns or overwrite columns, in addition to potential upgrade issues. Adding many columns for every possible option that might be needed ahead of time is also problematic—it takes space, and scalability can be severely limited when there are queries that require many OR clauses. Almost every enterprise application needs some kind of ad hoc query builder screen and programmatically creating a query to run this is often written as a sequence of OR clauses. Cost-based optimizers will make a decision at some number of OR clauses to revert from an index seek to an index scan. Once that threshold is reached, the query becomes long running and can lock large amounts of resources. This must be avoided to help ensure that the application is enterprise scalable.

One advantage of adding columns to the schema is that the data type is known both by the user and the database engine, allowing the database engine to error on invalid functions. It is also much easier for end users to create custom reports when using this option.

The second option is to add a property bag table, which is usually modeled as a name/value pair. This enables new columns to be added as rows, without changing the schema. Programmatically built queries can be built as UNION clauses. This is a highly scalable option and index seeks can always be performed. Finding one attribute in a table of a billions rows requires about the same resources as finding one attribute in a table of a thousand rows.

However, name/value pairs do not have enough information to make them really useful. A decision must be made about how to store the value when it could be of any type. One option is to store it as a SQLVARIANT, another is to place values in data type tables—one name/value pair table for strings, one for floats, and so on. In addition, there must be more metadata available to be used. Can you aggregate the value? SQLVARIANT has not been shown to cause performance or scalability issues, but it is not as easy and clean to use as a typed table.

The biggest issue by far is that all of this data must be pivoted so that it can be reported against. SQL Server Reporting Services (SSRS) is a very popular product because you can use it to develop an enterprise application without building the reports individually. End users can point SSRS to the database schema or to a SSRS Report Model and create their own reports. This is clearly a desired feature. Using name/value pairs requires some work by the development team to pivot rows into columns, either in data marts or programmatically. SQL Server Analysis Services does know how to pivot these rows into measures but the problem of losing the data type still remains.

The best practice is to pick a mix of the two approaches that works in your application and your implementation environment. Adding custom columns that can be managed by the customer as well as a name/value pair approach for the more obscure attributes seems to works best.

In the model in Figure 6, the Individual table contains known columns, custom columns, and an attribute table that has been subtyped from the main attribute table.

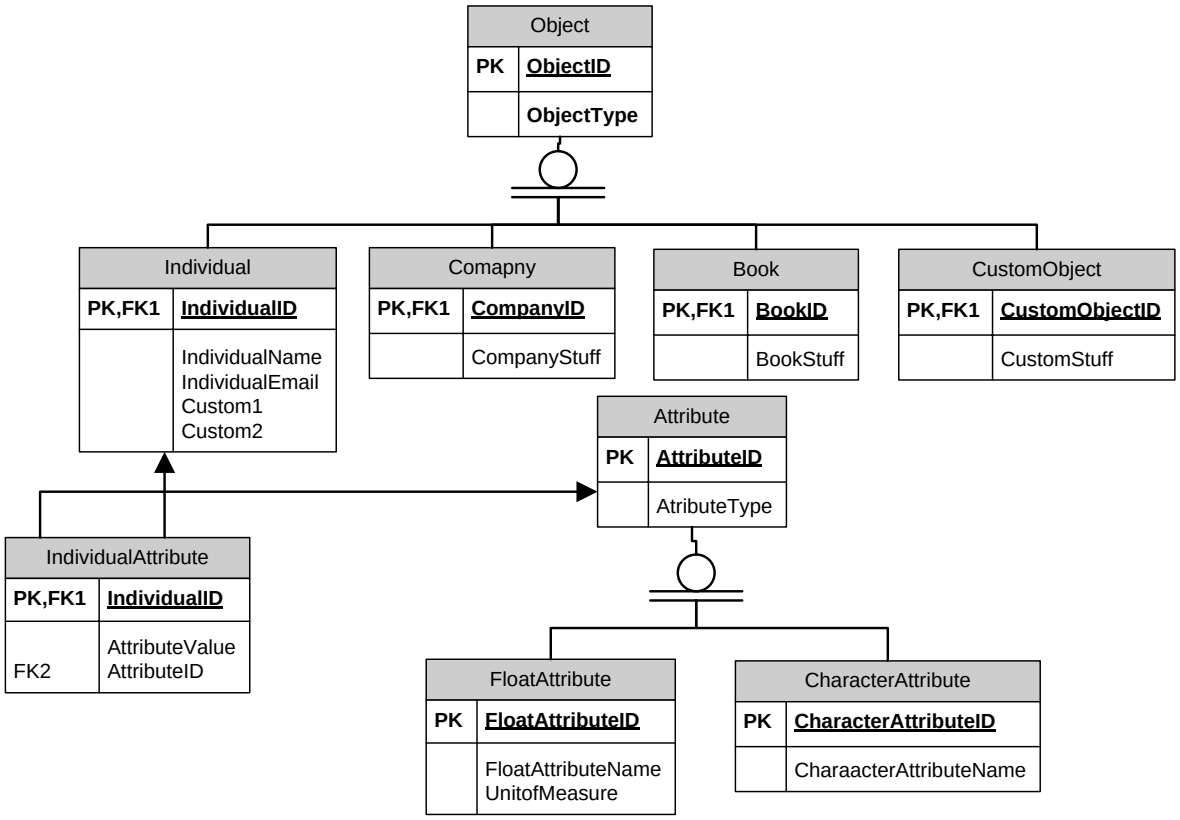

Figure 6

Limiting the number of possible OR clauses increases scalability, and limiting the number of name/value pairs increases usability. The optimal mix must be determined for Each application. The only way to determine this is to run load tests to determine where the model will fall over. You then must fine-tune the mix and test again.

Microsoft® SQL Server® 2008 provides another option—sparse column support with column sets and filtered indexes. *Sparse columns* were designed specifically for the optimized storage of columns that often contain null values. *Column sets* enable these columns to be retrieved as a single xml document. Filtered indexes provide the ability to create an index on a subset of data; in this scenario that would be only the data with actual values. Filtered indexes improve performance and could conceivably be created on every customer or extended column, helping to alleviate table scans on query builder queries.

Normalize, Normalize, Normalize

A common question in database modeling is when and where to denormalize. The current answer for semantic models is to normalize as much as possible unless there is a proven and tested reason not to. There are some common areas where we know we probably need to denormalize. One example is folder hierarchies. A table containing folder hierarchies might look like the following:

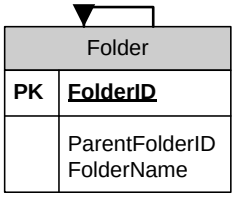

Figure 7

This model is the most efficient for space and the data will always be consistent. However, to retrieve the full path of a folder requires writing a common table expression to iterate through the hierarchy, which can significantly impact scalability. In this scenario, if folder locations do not often change, it is a useful denormalization to add a column with the materialized path, which can then be retrieved with one seek operation.

SQL Server 2008 provides new functionality for this in the **hierarchyid** data type, which can be used to represent this tree in an extremely compact manner, and can be sorted.

The SQL Server engine was completely re-architected in version 7.0, away from the original shared Microsoft/Sybase code. This meant SQL Server could design around a more purely relational model, setting the foundation for applications that could scale and perform out of the box if the database model also follows the relational model. This is a huge advantage. All of the metadata for a model—the primary keys, foreign keys, nullability, check constraints, and so on—are used by the optimizer to find an optimal execution plan.

The best practice is to design at least a third normal form semantic model and materialize that in the database with all of the corresponding constraints. This is especially critical for semantic models because there tends to be a lot of self-referencing in these and the impact of having no relational information will be felt in scalability much more than in other types of models.

Nullability

Nullable columns and relationships can limit scalability as there are many options for the optimizer to consider. If a foreign key relationship is nullable, programmatically built queries must use outer joins to ensure that all parts of the data are presented. Forcing all foreign key relationships to be non-nullable and creating a foreign key constraint on them gives the optimizer information that allows it to optimize the query and outer joins are usually not needed. The optimizer can even eliminate parts of a query.

For example, take the following model:

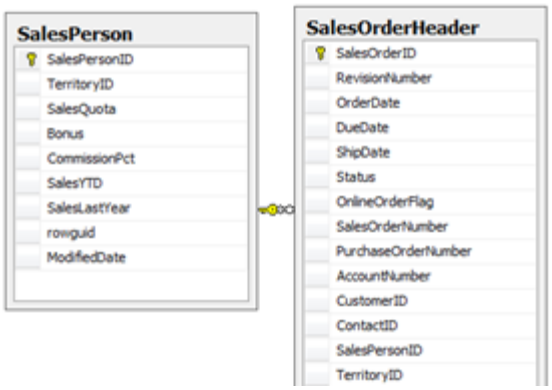

Figure 8

In this model, the relationship between the SalesOrderHeader table and the SalesPerson table is not nullable and there is a foreign key relationship defined. There is also an index on OrderDate to support the query predicate. It is common for a programmatically built query from a query builder application to write queries in a standard form, including columns that are on the application search form. a query against two tables that actually returns only columns from one of the tables is not uncommon.

This query and its execution plan would look like this:

```
select SalesOrderHeader.OrderDate
from SalesOrderHeader
join SalesPerson
on SalesOrderHeader.SalesPersonID = SalesPerson.SalesPersonID
where SalesOrderHeader.OrderDate between '01/01/2004' and '01/02/2004'
```

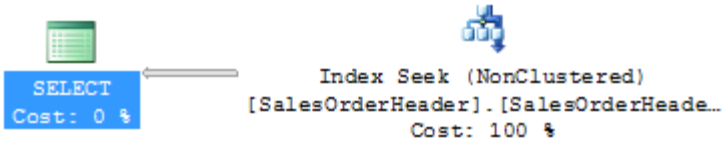

Figure 9

Removing only the foreign key constraint on the SalesOrderHeader table and rerunning the query produces the following query execution plan:

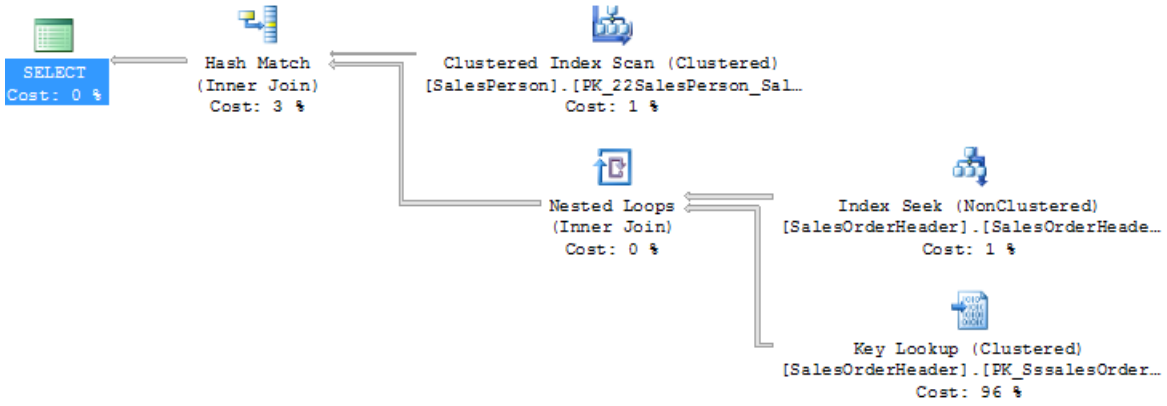

Figure 10

Clearly the foreign key constraint metadata is being used by the optimizer to eliminate the table that is not needed.

Note that there is another important issue with nulls that is often overlooked; their correct usage is not always understood by either developers or customers. Developers must code differently for nulls because the correct syntax is different from the syntax for non-null values. Not doing This leads to runtime errors. Queries can also return different results depending on the CONCAT_NULL_YIELDS_NULL setting. The following query should, and does, return 'AB':

```
declare @nvar1 nchar(1)
declare @nvar2 nchar(1)
set @nvar1 = 'A'
set @nvar2 = 'B'
select @nvar1 + @nvar2
If you set one of the variables to null:
set @nvar2 = NULL
```

The result returned depends on the CONCAT_NULL_YIELDS_NULL setting.

Three-Valued Logic

Many of us are not completely familiar with three-valued logic. Take the following test to illustrate this point. Complete each box for two expressions that are ANDed. (That is, if `exp1` evaluates to `TRUE` AND `exp2` evaluates to `TRUE`, the Boolean AND of both is TRUE)

AND	TRUE	FALSE	UNK
TRUE			
FALSE			
UNK			

Now complete the following for Or expressions.

OR	TRUE	FALSE	UNK
TRUE			
FALSE			
UNK			

Answers are in the following tables. The point is that avoiding nulls avoids this situation and potential problems for customers.

Three-Valued Logic Answers

AND	TRUE	FALSE	UNK
TRUE	true	false	unk
FALSE	false	false	false
UNK	unk	false	unk

OR	TRUE	FALSE	UNK
TRUE	true	true	true
FALSE	true	false	unk
UNK	true	unk	unk

Compensating Actions for Denormalization

It is very important for any denormalization that there be a compensating action. Normalization provides conformance to the rules of the relational model, for which relational engines are designed. Thus, normalization gives the optimizer the best chance to do its job effectively. It also protects data consistency. Both of these are critical elements to any system.

If a decision to denormalize is made, there must be a compensating action to ensure data consistency. In the example of the folder hierarchy and a materialized path, if a folder is moved by changing the parent, the materialized path is not automatically updated. the compensating action is that there must be a trigger or programmatic solution to change the materialized path as well. A denormalization always has other repercussions that may make it not as scalable or perform as well as intended. In the case of the folder hierarchy, the work required to build the path on every query is worth the cost of the compensating action.

Parent/Child Tables and Sequence IDs

Semantic models tend to be very deep in parent/child relationships; this is simply a common characteristic that is not often found in OLTP or OLAP data models. Problems with incorrectly modeling parent/child relationships typically do not affect scalability for the parent and child, but for all the grandchildren. Each generation compounds the problem because each generation provides a natural narrowing of the data and parent/child tables are not very different for the optimizer than a table with a domain

table. (A domain table is generally a type table, with a foreign key relationship with a regular table. This table provides the domain of allowed values for the column in the regular table.)

The following tables represent common entities on a manufacturing plant floor (simplified for this example). There is a sequenceID for each table and there are currently no relationships defined, so the optimizer does not have any information about relationships (these relationships are a concept and not defined in the system). A Site is the plant building, an area is inside the plant building, a line is a production line inside an area, a node is a piece of equipment on that line, and sensors live on nodes.

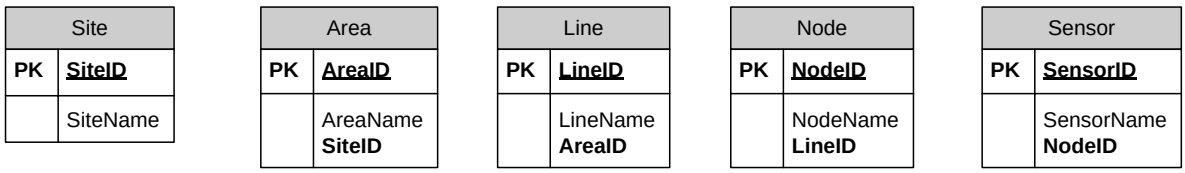

Figure 11

This is a common data model that we see when databases are migrated from Oracle. even though there are column names that are the same in each table, that does not indicate to a system that they have a relationship. the optimizer can make a good guess whether there is a relationship between the SiteID in Site and the SiteID in Area, but there is no guarantee that it will be correct.

A model with foreign key constraints for each table looks like this:

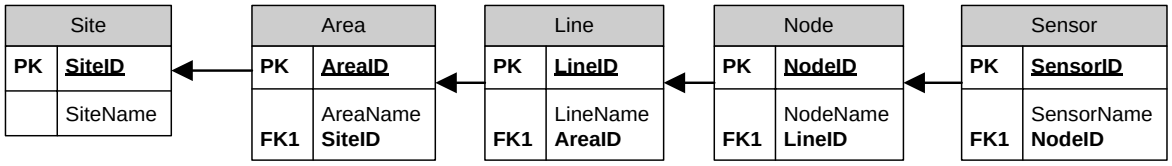

Figure 12

The problem with this model is that the optimizer (as well as a developer) will see these as domain tables that constrain the value of the NodeID in the Sensor table. But that is not actually correct. A sensor cannot exist by itself—it is a subpart of a Node—so it is a child table. This is true of the entire group of tables—each is a child of the previous. children are identified in the relational model by a compound key (the parent ID and an Instance or LineID).

Domain Relationship Parent/Child Relationship

Figure 13

Data modeling tools usually pull the primary key from the parent to the child table when you draw that relationship. This is what the model should actually look like:

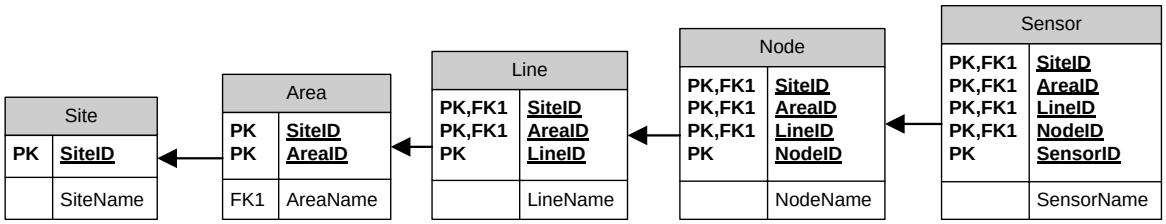

Figure 14

It is good practice to keep the parent keys in the same order in each child table for ease of reading. It is very clear in this diagram that each is a parent of the next. Another rule of relational databases is that you must join on the entire key; this model makes that clear as well. Joining the Sensor table to the Node table requires a join on SiteID and AreaID and LineID and NodeID on both sides of the join. Note that incorrect modeling, such as the model in Figure 12 can lead to the entry of incorrect data.

This task looks very onerous—there are so many keys and so much code. However, it is normal form and makes a huge difference in scalability in semantic models. The optimizer can pick absolutely correct join execution plans, and you can jump in the hierarchy at any point. For instance, finding all the sensors in one area can be accomplished by querying only the Sensor table. In the previous model in Figure 12, the parent keys are not carried down and you would have to join every single table to get that information. Watch for queries that return duplicate data and that use a DISTINCT clause to fix this. Duplicate data is a clear sign of a model problem and the problem is usually just this one—domain relationships instead of parent/child relationships. The DISTINCT clause should not be necessary to make a query correct.

To test the physical data model, reverse engineer the database into Microsoft Visio® and see if all of the relationships are intact. If Visio can pull out the entire model and relationships, so can the optimizer.

Surrogate Keys

Clearly the fact that the number of keys contained in the primary key gets larger as you move down a hierarchy is problematic. Many would be tempted to use a surrogate key instead of the composite key. Surrogate keys were initially used to save space, and as keys were carried to all other indexes it mattered; as that was the major concern a number of years ago. Surrogate keys have no business meaning—they are meant to stand in for something else. A CompanyID in a Company table is a surrogate key because it takes the place of the CompanyName, which is long, hard to use, and may change. (A primary key must not change.) In the plant model, the SiteID is a surrogate key for the SiteName. However, the SiteName column is still in the table. Generally when a surrogate key is used, you should still store the business data in the table—otherwise that data is lost. Surrogate keys save space only where primary keys are carried to other indexes or tables.

In our example, we could use a surrogate for the Sensor or Node tables, but the SiteID, AreaID, and LineID must be in the table because they are part of the definition of the table.

Sequence IDs

Sequence IDs tend to be found in databases migrated from Oracle. As there are usually Oracle DBAs available to optimize queries, the problems they cause for relational query

engines is hidden in the cost of their services. One of the main architecture goals for SQL Server was to minimize the need for optimization experts. To achieve this low total cost of ownership (TCO), take the time to model your application correctly, and remove ALL Sequence IDs. Let the optimizer do its work for you.

Data Model Designs

Data model designs are the first and most important piece of any database application. The first logical model is about the business requirements. After the logical model is complete and approved, the physical model is materialized into a database with constraints and indexes, nullability, data types, and so on. This enables the optimizer to work as a relational optimizer. The data model also indicates how queries can be correctly written. Joins should be used only along the relationship lines shown in the diagram. Because there tends to be a lack of modeling discipline in some database groups, developers join on columns that seem to have the same name or whatever criteria they can come up with. In the plant model, although there is an AreaID in the Sensor table, there is not a relationship line, so there should not be a join between those two tables. In fact, if a query is written that way, duplicate rows will be returned and a DISTINCT clause would be required. Joining all the tables results in good performance—joining numerous tables is not a problem; the problem is incorrect joining of numerous tables.

Indexing

After the semantic data model is relationally correct, the next step is to add indexes for all foreign key constraints as shown in the model, and to add a single column index for each column in tables that you anticipate will end up in predicates. When there are many small indexes, the optimizer can put them together in the most appropriate manner.

It is a best practice to have the primary key be the clustered key in semantic models, unless there are good reasons not to (such as performance tests that reveal a problem with this). Because we must stay very relationally correct and tight for scalability results, the keys are eventually carried forward anyway, so a clustered primary key saves space and gives the optimizer the best options

This should take care of about 80% of your query optimization needs. The other 20% you will find during your performance and scalability tests. It is important to reiterate that there is an optimal balance between an object view and a relational view of the data and you can find this for your application only by testing.

Query Builders

Most applications designed by vendors for end customer use supply some way to perform ad hoc searches without knowing ahead of time what those searches might contain. Generally, a query builder page that allows users to fill in whatever data is needed and perform a search is created. These are often non-performant and inhibit scalability as they tend to turn into table scans. Having many table scans limits scalability because there is too much contention on the physical data layout.

Because the query must be programmatically built, there might be many OR clauses that combine the user-selected data. These generally degrade into table or index scans

after a certain number of OR clauses because the optimizer determines that it is more efficient to scan each row and check for inclusion against the many criteria.

To lessen the impact of this, try to avoid data conversions, as they generally degrade to a table or index scan because the index is no longer of value. If you keep metadata about the columns that are included in the query builder page, you can programmatically ensure that numbers are entered into the query as numbers and not as converted strings, or another data type. Design your queries to use seeks as much as possible.

Paging

Customers can submit queries that return large numbers of results, whereas often the customer does not actually need all of the returned data. To keep a system scalable, you must design into the application a way to limit this impact. Including a TOP clause does not ensure that the entire table is not scanned before returning the limited amount of rows. The underlying work must still be performed, after which the end results are limited. Large amounts of resources are still being consumed and potentially locked.

A useful design pattern is to page the results, giving the user the top few results, with an option to request more data. Do not provide user controls that allow scrolling through all of the result sets as doing so requires that all data be returned. Use the design pattern often used in web searches—a result set of a certain size is returned, and the option to retrieve more is offered.

Lazy Loading

Like paging, lazy loading is a client design decision that can have a large impact on the scalability of the database. Trees should be initially displayed as collapsed with only the data required for the collapsed version. When a node is selected, the database call should be made to return just that data.

Developers often worry about the round trips required for lazy loading trees, but that data should come back well under the time needed to display it, and it increases scalability of the application by limiting resource usage.

Semantic/Metadata/Runtime Data Model Checklist

Evaluate semantic models for the following to help isolate potential scalability issues. Mitigating each area at design time will help to limit run-time problems.

- Build a logical model and make sure it is approved by the business
- Subtype from a Universal model
- Normalize
- If you denormalize, add a compensating action
- Remove all SequenceIDs
- Model parent/child relationships correctly
- Create a physical model and implement it in the database
- Avoid nullable relationships as much as possible
- Reverse engineer the database into Visio to check for correctness
- Design the user interface to support paging

- Lazy load trees as much as possible
- Evaluate all queries that require a DISTINCT clause
- Load test for scalability

Summary

Semantic data models can be very complex and until semantic databases are commonly available, the challenge remains to find the optimal balance between the pure object model and the pure relational model for each application. The key to success is to understand the issues, make the necessary mitigations for those issues, and then test, test, and test. Scalability testing is a critical success factor if you are going to find that optimal design.

For more information:

SQL Server Web site

SQL Server TechCenter

SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high because it has good examples, excellent screenshots, clear writing, or another reason?
- Are you rating it low because of poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of white papers we release. Send feedback.