MANAGING THE EARTHQUAKE: SURVIVING MAJOR DATABASE ARCHITECTURE CHANGES

Michael Rosenblum, Dulcian Inc.

Introduction

Changes are inevitable in any information system. Even with the best possible requirements analysis, it is extremely hard to anticipate all of the potential changes that may be required in a system over time. The best solution is to try to predict major failure points in order to reduce the chances of such failures to an "acceptable" level.

Used in this sense, "acceptable" is greatly dependent upon the project budget and management tolerance. However, in any efficiently run organization, proposing system modification costs that are comparable to the original system development expenditures could be considered a “fireable offense.” As a result, system architects are doomed to add layers upon layers of patches to allow systems to survive longer without any major modifications to the underlying mechanisms.

Good system architecture means building systems in such a way that the inevitable changes are less likely to make the system collapse under the weight of accumulated quick-fixes. The key principle seems logical. There should be NO quick-fixes at all. In reality, all rules are meant to be broken and this one is not an exception. There will be always some deadlines, project restrictions and other reasons to push developers to the “dark side.” The only solution is to create conditions for making changes correctly. This approach requires some serious thinking by the core system architecture team. This article discusses a number of approaches including real “war-stories.”

Two War-Stories

As an example of bad system design, the following problem was faced by our development team while enhancing and rebuilding the Budget and Finance System for the Finance Ministry of Ethiopia. In the existing data warehouse, a decision had been made to use “smart codes” for organization IDs. For every Region/Zone/City, etc., each block had a fixed length. Since all values were fixed-length, the simplest way to check the type of organization was to check the length of the ID (Ex. 14 meant city, 20 meant zone, etc).

Unfortunately, at some point an extra level in the middle of the hierarchy was introduced for reporting purposes, which destroyed almost 100% of the existing organization-based reports. The result required an enormous effort to repair and devise a workaround to glue the system back together.

This problem exemplifies a case when the volatility of a subject area contradicts very strict software implementation best practices. This is tantamount to building a house from the playing cards in a seismically active area. It is critical to take into consideration the probable length of the system usage, since a misjudged “event horizon” could turn perfectly well-thought out areas into danger zones.

As an example of good system design, the following modules were used while building a system for the United States Air Force Reserve. In all cases, we were able to handle significant system changes without expending a great deal of time and money. The following were achieved in very little time with minimal disruption to the existing production system:

1. Shift from single to multi-organization structure - This involved adding users from Air Force Active Duty and Air National Guard to the Reserve system and introducing the notion of shared vs. organization-specific data.
    - Technical side: 99% of the business rules (including structural) were stored in repositories with appropriate code generators. As a result, only a fraction of the system was hand-coded. Everything else could be changed by modifying repository data.
    - Development time: 3 months
2. Adding an extra level of organization hierarchy across the board
    - Technical side: From the start, the organization tree was implemented as an abstract hierarchy plus a set of materialized views (for performance reasons). This double-structure was a bit challenging to configure and maintain, but it provided enormous flexibility.
    - Development time: 1 month

3. Introducing offline capabilities for a module with approximately 300,000 lines of generated PL/SQL
   - Technical side: All online object-to-object transformations were described in the repository, while the PL/SQL code was fully generated. Since all transformation rules were preserved, the only required change was to create a new generator for the Java-based offline tool that would read data and write to offline XML storage and vice versa.
   - Development time: 3 months

Proposed Strategies for Success

Fifteen years of experience at Dulcian have led to the development of the "waterfall of strategies":

1. You must start with robust data models. They are less likely to be disrupted by changes and, in general, make the whole development process more straightforward.
2. Good system architecture allows major changes to be made inexpensively. One of the key definitions of good system architecture is that you are not doing similar things multiple times. As a result, different engines and repositories make the development process more efficient because you can delegate tasks to the engine rather than to 20 coders with a higher probability of bug-free code.
3. From the front-end side, a "thick database" approach is less susceptible to inevitable UI architecture shifts. Keep in mind that you are building systems for people to use. However, every once in a while there are conceptual changes to how people would like to see their data. Those changes sometimes have no clear rationale (other than contemporary trends), but those trends should not endanger the system as a whole.

Although this list is neither exhaustive, nor carved in stone, it has proven effective in many situations and provided the highest ROI for an organization's IT budget.

Data Models

In the current database development environment, the trend of pushing everything into the middle-tier and using the database as nothing more than persistent data storage, the skill of effective data modeling becomes lost mainly due to the shifting of complexity to different areas. However, there are some common modeling patterns and anti-patterns that persist:

1. Patterns:
   a. Type instead of multiple associations
   b. Organizational Unit (~abstract creature) instead of specifically designated types
   c. Generic tree structures
2. Anti-patterns:
   a. Hard-coded structures
   b. Smart attributes
   c. Over-generalizing

The following sections illustrate how these patterns impacted real-life production systems.

Types vs. Hard-Coded Structures

At Dulcian, an official system development policy was set up whereby the request to add a second association of the same kind between two classes MUST include a guarantee that there will NEVER be a third association. If this guarantee is not possible, then an intersection class is created.

This policy arose when building a system for the US Coast Guard. The original specifications defined that ships were repaired by special organizations. In later stages of the project, it was discovered that organizations could also manage and maintain ships. Since the deadline for completion was imminent, the in-house development team unanimously voted to add two more associations between SHIP and ORG, so there would be no impact on the existing code. Unfortunately, there was no

guarantee that there would be no more surprises later on. It took a direct act of management to implement the generic structure shown in Figure 1.

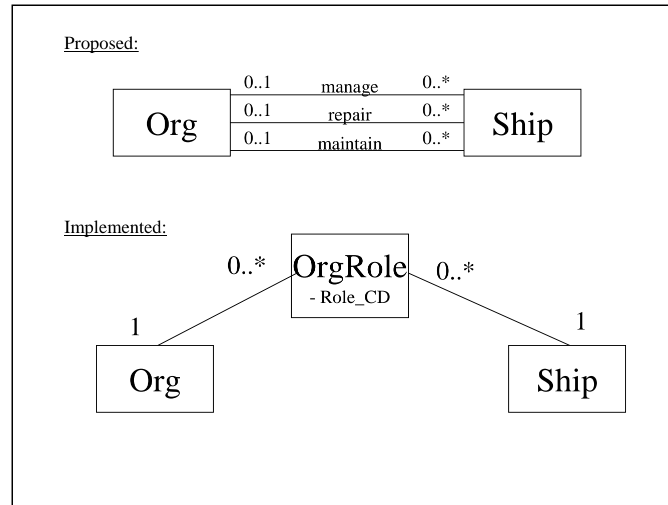

Figure 1: Generic Organization Model Structure

Two years later, a system audit revealed seven different types of roles in the OrgRole table, not just the original three. Introducing each of these additional types took significantly less time and required less money than anticipated.

Lesson to be learned: If it is not ONE, it is MANY.

Organizations and Organization Trees

Another common best-practice used at Dulcian is building an organization structure as an abstract tree of elements, where each element in the tree is an organization unit of a specific type as shown in Figure 2.

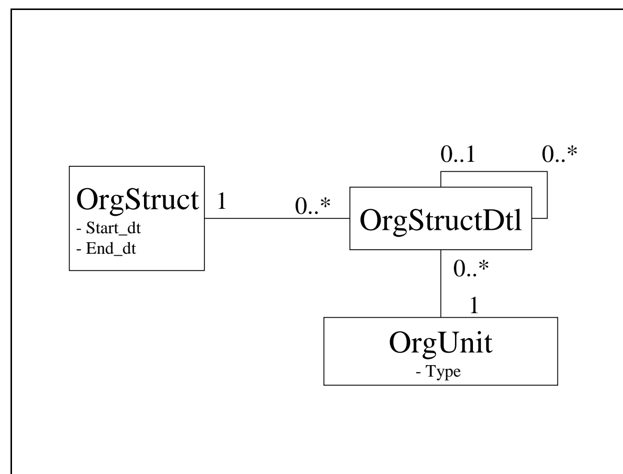

Figure 2: Abstract Tree of Elements

This approach contradicts standard design patterns, because instead of Department/Region, etc. as a part of Organization we use an abstract "Organizational Unit" as a brick with which to build the hierarchy. This UNIT table will contain all attributes common for Unit Types (Name, Code, etc.) and could have child tables with specific attributes. The last part is used on an "if

needed" basis. For example, in the United States Air Force Reserve system there is only one special table (DESK), while every other unit type is covered by the UNIT table.

Another unexpected element of the design is the fact that there are start/end dates only on the tree (ORGSTRUCT table) itself, instead of on each association between nodes. This single point timestamp makes the process of figuring out organizational chains much simpler, although it forces much more data in all tables, because the whole tree has to be versioned any time that a change is made.

This generic tree has proven its effectiveness a number of times:

1. Adding an extra organizational level required minimal development instead of a major system rewrite.
2. Creating a common structure for other armed services branches became possible by adding 10 extra organization types and providing rules about how to correctly link them together.
3. Time-stamping the whole tree (rather than separate elements) made reporting straightforward, because, at any point in time, there could have been one, and only one, active tree upon which reports can be based.

After several years (when the number of nodes reached the hundreds of thousands, a number "must-haves" were introduced in order to make development easier and maximize system performance. People often forget that hierarchical queries are significantly more expensive than regular ones. There are always two very costly questions to be answered: What is the state of the tree now? and How much time will an organizational unit be active?

The first question is critical for day-to-day activities because this information could be called thousands of times by all sessions and should provide the data instantly. Our solution was to introduce a set of denormalized materialized views with the "current" snapshot. This strategy avoids expensive hierarchical searches. In addition, materialized views are automatically refreshed every night and contain many indexes for fast querying.

The answer to the second question also required denormalization. But instead of materialized views, it was necessary to use real appendable tables that "flip" the data. Instead of answering the question "What organization belongs to what tree?," they show the length of time that the organization belonged to the specified chain of command. These tables are appended when a new organization tree becomes active.

Lesson to be learned: Generic recursive structures can simply represent very complex structures. Too much data in recursive structures without any special handling can significantly impact the system performance.

Over-Generalizing (Anti-Pattern!)

It is a common trap for many new developers when discovering the power of generic data models to "over-abstract" the data model and end up with something like the "Stuff" model shown in Figure 3.

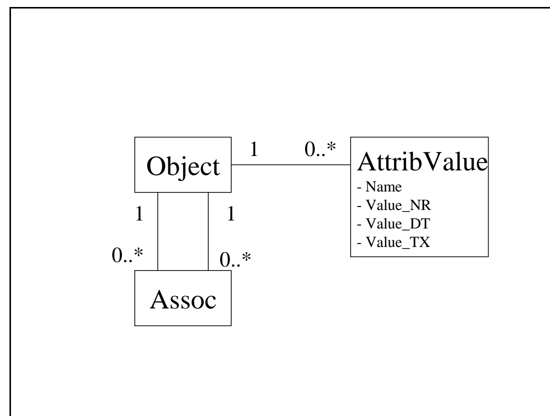

Figure 3: "Stuff" model

Although this model articulates everything and works just fine for SCOTT/TIGER-level problems, (i.e. very localized data models with few tables), trying to scale these models for large production systems will cause them to collapse under their own weight. This occurs because the cost of retrieving of a single data element from this type of structure is comparable to hard-coded solutions. In other words, beware!

Lesson to be learned: "Generic model" is not always a synonym for "good model".

System Architecture

Very often the best way of avoiding future problems when building a system is to radically change the way in which to approach the problem. This may not only solve the original problem much more efficiently, but also lead to other improvements. We strongly believe that the following set of concepts can make any system better:

1. Repository-based development allows all system rules to be well-organized and manageable. If a major part of your system is generated, it is much easier to introduce new features and make changes.
2. Using an Event-Action framework makes user interface rules independent of the current front-end technology and also decreases the total number of roundtrips between all three system layers.

Repository-Based Development Example: Data Mapper

At Dulcian, we have a number of different repositories: Object Transformation (data mapper), Data Validation, Process Flow, and Data Modeling. All of these repositories have different interesting pieces, but the most widely used one is the Data Mapper.

Originally the Data Mapper started as a purpose-built solution for the US Air Force Reserve. The core of this system involved maintaining hundreds of different forms. Prototypes of all forms were hard-coded, which caused major development issues any time additional rules or data structures needed to be added to the system. Abstracting the process helped to solve this problem using the following elements:

1. Generic repository to define object transformation (bi-directional):
    a. Data in the database → data in the form
    b. Data in the form → data in the database
2. Code Generator to make the transformation performance efficient. Eventually, we ended up generating the whole transformation into a PL/SQL package because real-time Dynamic SQL was more expensive. But having real generated code also made the debugging process more direct, since there was also a piece of code to play with.

The model shown in Figure 4 is based on the following points:

- All required structural objects and their attributes are defined in the database.
- A map is a process that reads from the source object and writes to the destination object.
- All “reads” can be arranged hierarchically (First - process department, second – process all employees for this department, third – pick up next department)
- “Writes” are done to a different structural object defined in the database. Those “writes” could be either initial or repeated, i.e. newly created record can also be updated with the data from the following “read”.
- Any data element from the source can be transformed via a specified expression (stored in MAP_COLUMN).
- MAP_COLUMN can contain a pointer to a unique identifier of an element generated by a different “write.” For example, if you want to create a department and its employees, you need to pass DEPT_ID to every EMPLOYEE record, but DEPT_ID is not known until the runtime. As a result, we need a mechanism to create a pointer to this future value.

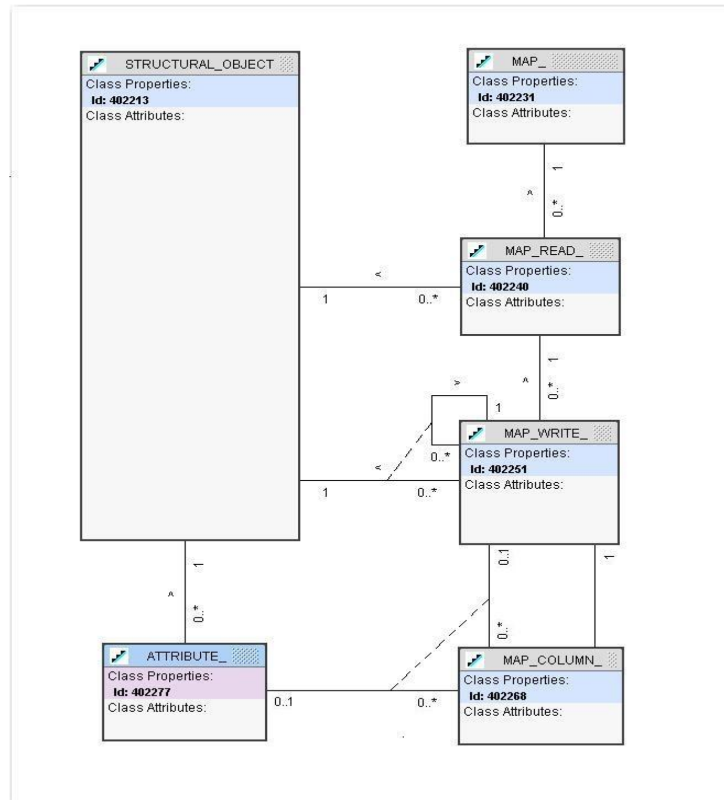

Figure 4: Mapper Data Model

Although the process of changing the data model required several months, once the shift was complete, the development process was greatly accelerated. Two hundred additional forms were added in two months (instead of the originally expected 12 months). Bug fixes could be made in a matter of minutes and major data model changes only required a few days to update the dependencies in the repository.

In addition to the development speed benefits, the same repository could support much more including:

- Data transformation inside the database
- Data migration
- Processing data to/from XML
- Offline data export

Currently, there are more than 1,000,000 lines of PL/SQL code generated from the same repository which demonstrated that creating a single generator is much more efficient than writing hundreds of hard-coded routines.

Lesson to be learned: A well-built repository can often be used for much more than it was initially intended.

Event-Action Framework

One project implemented in a developing nation required a major shift in system architecture. The biggest challenge was the network connection speed (often less than 5K). In order to deploy the solution over the web, there were two problems to solve:

1. Minimize the number of requests between the client and application server
2. Keep the page size as small as possible.

Both of these goals were achievable using the event-action framework shown in Figure 5.

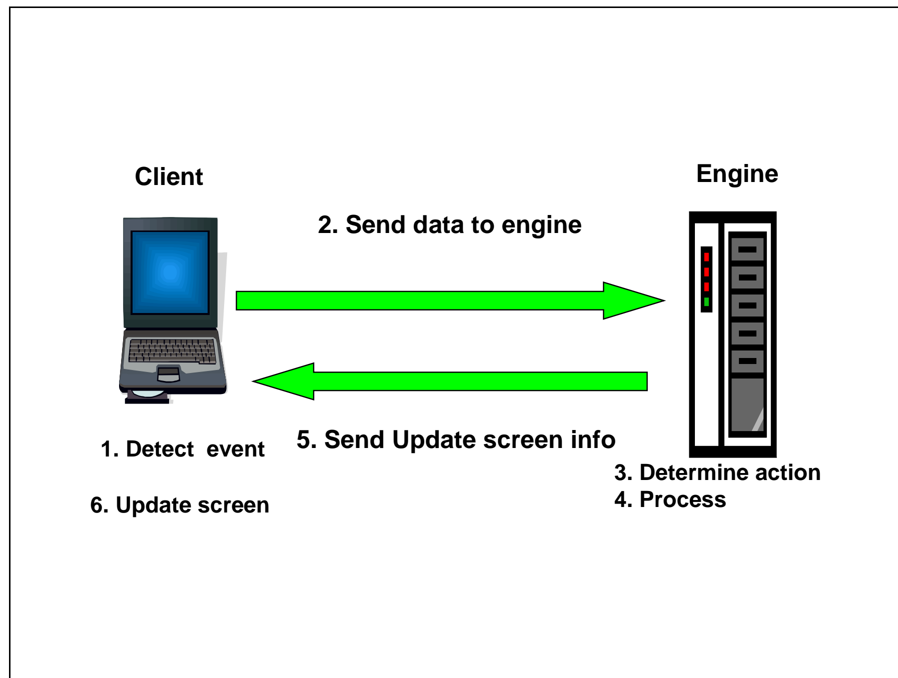

Figure 5: Event-Action Framework

Instead of transferring complete (or even partial) pages between the engine and the client, the client only indicates any changes to the engine (event) and the engine performs the appropriate actions and applies them to the known state of the client. Since the server is aware of the old (before action) and new (after action) client states, the only information required deals with any visible changes. For this system, the average transfer was less than 1K per screen.

Of course, behind all of this there was a repository where all events and actions were defined in conjunction with complete screen elements. This repository linked the whole system together and fed the engine with required data. Also, the repository allowed some niceties such as using only one UPDATE statement to generate labels for all required UI elements.

In terms of a client-side event detector, the great advantage of this approach when migrating from one platform to another (for example from JavaScript to Flash) is simply a matter of writing a different event/action interpreter, while all UI rules remain intact.

Lesson to be learned: It is possible to articulate UI behavior in a technology-independent way. To our surprise, this style of thinking made our front-end more efficient, because UI developers were focusing on the action flow of end-users rather than on screen beautification.

"Thick Database" Approach

This concept is still considered by many to be "heresy" in the IT world. However, based on our experiences, following one very simple rule significantly increases the chances of both short-term and long-term project success: The user interface screens never touch a table!

The idea is to convert relational data into something that will make user interface development easier. The best way to accomplish this is to separate the data representation in the front-end from the model. The solution is to specify all columns from all tables (with appropriate joins) that would be needed and create a view as shown here:

```
create or replace view v_customer
as
select c.cust_id,
       c.name_tx,
       a.addr_id,
       a.street_tx,
       a.state_cd,
```

```
       a.postal_cd
from customer c
left outer join address a
   on c.cust_id = a.cust_id
```

To make this view updatable, a special kind of trigger (INSTEAD OF) can be created. The INSTEAD OF code construction can be used to Insert/Update/Delete as shown in the following three code samples:

```
create or replace trigger v_customer_ii
instead of insert on v_customer
declare
  v_cust_id customer.cust_id%rowtype;
begin
  if :new.name_tx is not null then
   insert into customer (cust_id,name_tx)
    values(object_seq.nextval,       :new.name_tx)
   returning cust_id into v_cust_id;
  if :new.street_tx is not null then
   insert into address (addr_id,street_tx,
        state_cd, postal_cd, cust_id)
   values (object_seq.nextval,:new.street_tx,
    :new.state_cd,:new.postal_cd, v_cust_id);
  end if;
end;

create or replace trigger v_customer_id
instead of delete on v_customer
begin
    delete from address
    where cust_id=:old.cust_id;
    delete from customer
    where cust_id=:old.cust_id;
end;


create or replace trigger v_customer_iu
instead of update on v_customer
begin
 update customer set name_tx  = :new.name_tx
 where cust_id = :old.cust_id;

 if :old.addr_id is not null
 and :new.street_tx is null then
  delete from address where addr_id=:old.addr_id;
 elsif :old.addr_id is null
 and :new.street_tx is not null then
  insert into address(…) values (…);
 else
  update address set … where addr_id=:old.addr_id;
 end if;
end;
```

It is not always possible to represent all required functionality in a single SQL statement, which means that denormalized views cannot be built directly. But Oracle has provided a different mechanism that allows you to hide not only the data separation, but also all of the transformation logic. This mechanism utilized object types and object collections (aka “nested tables”), because these collections can be used in SQL queries with the special operator TABLE. The core idea is pretty simple. TABLE operation transforms a user-defined collection into a SQL dataset. This means that all procedural transformations happen under the hood in PL/SQL and the results can be presented in straight SQL.

Of course, such a solution requires advanced knowledge of the Oracle toolset, but the benefits are significant enough to bother learning. To illustrate the effectiveness of this SET-based development, a few years ago I needed to build a back-end for a very complex search screen. It included dozens of different parameters crossing a number of huge tables. Of course, I could always write one huge SELECT statement that would cover all inputs, but performance was sub-optimal, mainly because we were using more elements in this super-SELECT than were needed. The only way to rewrite the query on the fly was to use Dynamic SQL, which meant a PL/SQL implementation.

Since the result set eventually should be used in SQL, the starting point was to create an object type to match the output (and corresponding collection):

Managing the Earthquake

```
CREATE type search_ot as object
       (Name_TX Varchar2(50),
        Phone_TX varchar2(20),
        …)
CREATE type search_nt as table of search_ot;
```

The next step was to build a function that would take all o fthe search criteria, and the built-on-the-fly SQL statement, execute it, and return a collection of detected entries.

```
FUNCTION f_search_tt (i_ssn_tx varchar2, i_city_tx varchar2, …)
Return   search_nt
IS
    V_tt search_nt;
    v_sql_tx varchar2(32000);
BEGIN
    v_sql_tx:='select search_ot(...) '||chr(10)||
               'from ... '||chr(10)||
               'where ...';

    if i_ssn_tx is not null then
         v_sql_tx:=v_sql_tx||' and cust.ssn_tx like ''%'||i_ssn_tx||'%'' ';
    end if;
    ...
    execute immediate v_sql_tx bulk collect into v_tt;
    return v_tt;
END;
```

This function can return an object collection and can be shown to any front-end developer in the following form:

```
select name_tx, address_tx, phone_tx, …
from table(
           cast(f_search_nt
                  (:1, -- ssn
                   :2, -- city
                    …
                  )
           as search_nt)
           )
```

This query completely hides the data transformation complexity by design. But it also makes our UI safe from any low-level structural changes. After the data model transformation, as long as we can generate the same result (even from different tables), there will be absolutely no front-end impact. This logical firewall introduced by both function-based queries and denormalized views is the reason why it is so critical to hide the real tables. There are also security reasons, but they are secondary to the notion of separation between data-as-stored and data-as-presented.

The concept of "thick database" requires developers to understand such a separation and work on appropriate levels only with needed elements. This leads to some other interesting best practices. This explains why all bulk operations are done in PL/SQL because PL/SQL has the shortest access pass to the real data (not its presentation). In addition, data-related UI logic can be pushed into the database. However, thinking about data validation, it is also clear that the majority of data validation rules involve the stored data and not the front-end represented data).

Lesson to be learned: Maintaining a clear separation between data stored in the database and data presented to end-users makes the system much more independent of front-end changes.

Summary

There are many different approaches to help make systems more reliable and sustainable over time. To summarize the most important ones:

- Good system architecture does matter.
- Good system architects play a key role in building successful systems.
- Devoting adequate time and resources to good system architecture in the short run will always pay off in the long run.

About the Author

Michael Rosenblum is a Software Architect/Development DBA at Dulcian, Inc. where he is responsible for system tuning and application architecture. Michael supports Dulcian developers by writing complex PL/SQL routines and researching new features. He is the co-author of PL/SQL for Dummies (Wiley Press, 2006), contributing author of Expert PL/SQL Practices (APress, 2011), and author of a number of database-related articles (IOUG Select Journal, ODTUG Tech Journal) and conference papers. Michael is an Oracle ACE, a frequent presenter at various Oracle user group conferences (Oracle OpenWorld, ODTUG, IOUG Collaborate, RMOUG, NYOUG, etc), and winner of the ODTUG Kaleidoscope 2009 Best Speaker Award.