IDIOM
PURE DECISIONING

The Role of SQL in Decision Centric Processes

White Paper

Idiom Ltd

2014

Contents

SYNOPSIS

This article is the most recent in a trilogy of articles that map the evolution of a proven, practical, and robust methodology that applies decisioning techniques to fundamentally remake commercial software architecture and development. The previous articles are online at Modern Analyst, and are referenced in this document:

Requirements and the Beast of Complexity[1]

Decisioning – the Next Generation of Business Rules[2]

The demonstrated results of the methodology include order-of-magnitude improvements in both development and runtime performance for complex commercial transaction processing, with examples that are relevant to investment and pension administration, payroll, utility and health billing, lending, and insurance products, amongst others.

These results are accompanied by substantial improvements in business agility, and systems that are more transparent and more durable.

The articles discuss the underlying concepts at length, providing an understandable rationale that is delivered in the context of a practical methodology.

This version of the paper includes an Epilogue that introduces the IDIOM tools that fully support the concepts described herein.

Mark Norton, CEO and Founder

IDIOM Limited

[1] http://www.modernanalyst.com/Resources/Articles/tabid/115/ID/1354/Requirements-and-the-Beast-of-Complexity.aspx

[2] http://www.modernanalyst.com/Resources/Articles/tabid/115/ID/2713/Decisioning-the-next-generation-of-Business-Rules.aspx

INTRODUCTION

Defined benefit schemes are contracts between a pension fund and its members. They are renowned for their complexity and long-life. We have recently codified the trust deed for one large fund with a membership in the hundreds of thousands. The underlying trust deed is more than 30 years old, and the complexity of the entitlement calculation has grown considerably since it was first drafted. [See the sidebar: 'Example Use-Case: Defined Benefit Scheme' for a précis of the calculations.]

A member's entitlement under the scheme can change on a daily basis; as you can imagine, the current entitlement is keenly sought on a regular basis by the members as they try and manage their affairs, both approaching and post retirement. This calculation is currently done in batch in a legacy system.

The legacy system batch process that calculates the current scheme entitlements takes >48 hours to run; furthermore, these are only top-up calculations that simply calculate the latest adjustments on the assumption that all prior calculations are sound. This is not always a good assumption.

Sidebar:

Example Use-Case: Defined Benefit Scheme

- The calculations require large amounts of source data - including 30 plus years of working hours and salary history for hundreds of thousands of scheme customers, with many rules applied at source to adjust for historical data anomalies (e.g. an employment terminated with no record of that employment ever commencing);
- A long standing customer can have as many as 40 separate periods of employment service, as a new employment service period commences with any change in service: role, employer, payroll, part time work, leave without pay, temporary incapacity, permanent disability, deferral from the scheme, leaving the scheme, re-joining the scheme, etc. with each requiring special treatment in the calculations;
- There are 40 or so intermediate calculated values that contribute to 30 or so benefit entitlement calculations, all of which incorporate many historical changes in the scheme over its 30 year life (e.g. prior to a particular date a calculation is undertaken in a certain manner with a certain set of rates applicable, which is then changed a few years later, etc.), so that some benefit calculations have 4 or 5 of these historical changes inside each calculation method;
- The result is calculations that are multi-layered, starting with several high level components and cascading down through multiple layers of sub-calculations so that some individual calculations have more than 100 discrete sub-calculations within them;
- Each calculation or sub-calculation might need to include indexation by either daily or quarterly CPI in different circumstances – e.g. just for a certain period, or from the date of a certain event forward or backward in time, or for a certain event only when other specific conditions apply;
- Some calculations require dividing period-of-service base data into multiple smaller time-boxes based on externally supplied dates (e.g. Scheme Date, Calculation "as at" Date, 20 year service threshold date, 65th birthday), with different calculation rules applying in each new time-box.

The original intent behind our involvement in this project was to confirm that assumption about the soundness of all those prior calculations. This meant recalculating the full 30+ years of contributions and entitlements from original source data, and reconciling the results with those that were calculated by the multiple legacy systems that had been used to manage the fund over the years. While we can now safely say that the current fund calculation is correct, there was a surprise result at the conclusion of our effort – the new full 30+ year calculation runs an order of magnitude faster than its current legacy system 'top-up only' equivalent.

This reinforced some prior experiences we have had with earlier projects that replaced 'SQL only' audit solutions; again, order of magnitude performance improvements when compared with the traditionally styled, SQL oriented solutions. We were intrigued as to why this should be so, and so we decided to investigate widely held assumptions about the use of SQL in complex applications that process large numbers of instances.

The resulting investigation has led us to believe that the underlying approach to the use of SQL in an application is the dominant contributor to the overall performance of the system, whether good or bad. This article will attempt to explain how the approach that we use is different, and why it can have such a profound effect on system performance.

The findings are relevant to any domain with similar, contract-defined obligations where there is an emphasis on complexity, compounded by the term of the obligations, the frequency of interactions, and the number of parties involved. Some common domain examples include investment and pension accounts, payroll, utility and health billing, lending, and insurance products[3].

It is normal for the customer obligations to be recorded in databases, and for the contract defined calculations to be embedded in programs. These calculations include many business rules, which are applied to data that are extracted via SQL according to yet more business rules. The combined effect of the SQL and program based business rules is a degree of complexity that is frequent source of errors; our experience auditing many of these systems to date suggests that such errors are a reasonably common production problem rather than a rare exception. Because of this, a small industry exists to quality assure existing systems in these domains.

Wherever data is used on a commercial scale as per the example domains, SQL is the data access tool of choice. However, the extent to which it is used, and the role attributed to SQL, can vary widely.

And especially in the case of the quality assurance processes, 'SQL only' programs are often used to verify the existing system calculations.

This article discusses how SQL can be used in concert with calculations in complex commercial transactions to consistently obtain better system performance, while at the same time increasing the business agility of systems that are both more transparent and more durable.

[3] A note on terminology: each domain has a different term for its 'customer', for instance pension fund = member, payroll = employee; loan = borrower etc. We will use the generic term 'customer' to refer to the 'other party' throughout this article.

THE PROBLEM

Pure SQL processes are common for audit and test verification in the nominated domains, where SQL queries that include a proxy for the underlying calculations are seen as offering a quick and cost effective approach to replicating, and in so doing verifying, core system functionality. For reasons outlined in this paper, whether this is true or not is very dependent on how complex the target problem is. A relatively low level of complexity can quickly overwhelm a SQL only solution.

On the other hand, most production use-cases see SQL queries embedded in traditional programs for a better division of labor, with SQL doing the data access and the program calculating the results. While these may manage complexity better than the pure SQL processes, they are still prone to the symptoms of complexity overload – processes that are difficult to understand, to test, and to debug; and which suffer from excessive database resource consumption leading to poor runtime performance.

The following symptoms summarize the motivation of clients who have requested our assistance in developing alternative solutions for these processes:

- Suspected errors, or an inability to test and verify that there are none: in one major project, 30% of the replaced SQL audit queries were themselves found to have errors.
- Lack of business agility and auditability caused by a lack of calculation transparency, and difficulty in understanding the calculation logic; the business is reliant on a manual interpretation of the code base performed by a skilled technician – however, the IT technical people themselves often have little reason to be confident that they fully understand the coded processes, particularly when they are coded in SQL.
- Poor operational performance/high resource consumption: for instance, the Example Use-Case takes more than 48 hours to execute for all its customers on the latest generation high end servers.

High complexity and poor performance appear to be linked in our assessment. As calculation size increases, so usually does the number of values consumed by the calculation and the number of intermediate steps required prior to calculating the ultimate intended result.

Because of both the network effect[4] and the subtle effects of context (to be discussed), an increasing number of component parts in a calculation has a disproportionate effect on the overall calculation complexity, which grows increasingly quickly.

As we described in our paper on Modern Analyst[5], business rules can be 'iceberg like' with an unseen but often substantial burden of data manipulation required prior to and as part of the application of the rules that actually deliver the intended result. In fact, the majority of coding effort required to process calculations like those found in the Example Use-Case is related to manipulating the data to arrive at the critical input variables that the ultimate entitlement calculation actually requires.

[4] The network effect is an exponential growth in the number of connections in a network as the number of nodes increases.

[5] http://www.modernanalyst.com/Resources/Articles/tabid/115/ID/2713/Decisioning-the-next-generation-of-Business-Rules.aspx

Each calculation of the Example Use-Case has hundreds of discrete component calculations driving demand for hundreds of source data variables across thousands of rows that are extracted from tens of tables – for each customer.

There is substantial complexity in transforming all of that raw source data into the temporary variables that are actually used by the ultimate high level entitlement calculations, and in sustaining them throughout the duration of the transaction. This can lead to complexity induced loss of control of the coded logic, and an explosion of complex queries as the developer struggles to maintain context throughout the calculation process.

SQL

Let's take a small and quite simple SQL sample and look at it in a bit more detail.

Suppose our calculation needs the interval in days between two dates that are each in different rows for the same customer. The problem is not conceptually difficult and can be done intuitively by a person.

Customer Key	Date	Days - Difference
1	1/1/2000	0
1	1/2/2000	31
1	13/2/2000	12

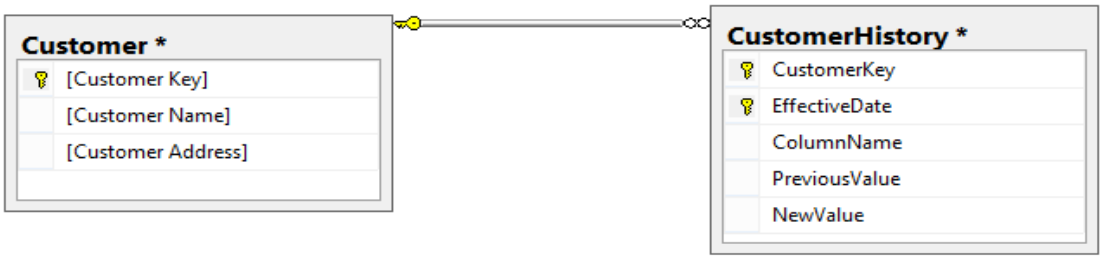
Many articles can be found on the internet defining how this problem could be approached in SQL. The two main options are: Sequential Processing and Set Processing.

In a Sequential processing approach, either a cursor approach or the creation of temporary tables can be used. Both of these options are generally considered to be poor performing to the point that we rarely see them used in real time applications.

The origin of SQL[6] is as a set processing language, and as such the correlated sub query used in set processing is the appropriate and probably best performing solution, and this is true in a simplistic case such as this. However, the structure of such queries puts them in the realm of experts when complexity increases, and one of the consequences of poor SQL design will be poor performance.

Below is the data model that is used in the following code samples.

[6] http://www.cs.berkeley.edu/~brewer/cs262/SystemR-comments.pdf



Sequential Processing

A number of sequential processing solution options exist, and we have included a sample cursor statement below to demonstrate the complexity and potentially poor performance of this solution, which requires separate SQL statements to be run for each row. As mentioned earlier, there are other potential solutions in the sequential processing domain but all require a number of steps to complete and are subject to similar performance constraints.

```
declare @dayDiff as int
declare @currentDate date
declare @lastDate as dateTime
declare GetDayDiffs CURSOR
for select  CustomerHistory.EffectiveDate from
       customer inner join CustomerHistory on Customer.[Customer Key] =
CustomerHistory.CustomerKey
order by EffectiveDate asc
Open GetDayDiffs
Fetch Next from GetDayDiffs into @currentDate
While @@FETCH_STATUS = 0
       Begin
              select @dayDiff = datediff(day,@lastDate,@currentDate)
              Fetch Next from GetDayDiffs into @currentDate
              select GetDayDiffs.
       End
close GetDayDiffs
deallocate GetDayDiffs
```

Set Processing

While this is a better option than the sequential process above, there are still issues. Internally the statement needs to run multiple queries per row returned; the statement is complex and the joins are

critical to maintaining performance. While this demonstration is somewhat trivial, an increase in complexity can rapidly make this SQL hard to build and maintain.

```
select *, (select EffectiveDate from CustomerHistory as c
where c.CustomerKey = b.CustomerKey
and c.EffectiveDate = (Select max(EffectiveDate) from CustomerHistory d
                                        where c.CustomerKey = d.CustomerKey and
d.EffectiveDate < b.EffectiveDate)) as LastDate
from Customer a inner join CustomerHistory b on a.[Customer Key] = b.CustomerKey
```

Scale and Complexity

Keep in mind that the above is required to calculate a single intermediate value. And our Example Use-Case has hundreds of such values that are calculated in a hierarchical fashion, with each value being consumed by potentially many dependent downstream calculations.

We can recalculate each value when required by a downstream calculation, or we can calculate it once and save it for reuse – but if so, where? If we are to stay with a SQL centric approach, then we presume that we would save the value in a variable in a temporary table. For a complex set of intermediate variables like those required in the Example Use-Case, many such tables are required.

If we were to use a set based approach, then we would likely be calculating each value for all customers in the portfolio, progressively calculating and storing the values in temporary variables in the exact order required to allow us to ultimately derive the top level results. The implication is that the same data is read and re-read many times as we process the full extent of the database multiple times.

And because we are processing for all customers each time we calculate a new intermediate variable, this processing will not occur in memory. The full weight of database overhead to read and write every variable, including index maintenance on any temporary tables, is required to manage variables whose values are often only actually required to exist for a fleeting instant in time – that is, the split second between being created and being consumed.

And this approach increases complexity in another dimension – time. Rule 2 (the 'Guaranteed Access Rule') of Codd's 12 Rules[7] specifies that any given value should be able to be retrieved using table name, column name, and row key. In a complex process using many intermediate steps, this is not sufficient because the value can change over time within the calculation itself, meaning that when we read the variable becomes important.

[7] http://en.wikipedia.org/wiki/Codd's_12_rules

Context

What is Context

The above brings us to the question of context. “Context” has a number of precise technical meanings in specific computer systems and languages which are not relevant here. Its everyday meaning as per TheFreeDictionary[8] is as follows:

“The part of a text or statement that surrounds a particular word or passage and determines its meaning.”

While the FreeDictionary version is relevant to this paper as a general case, we will offer our own idiomatic definition for use in this paper. Our definition is aligned with Bill Schilit and others who have defined ‘context’ in a database context viz. “where you are, who you are with, and what resources are nearby” [9].

In our definition of context, the ‘you’ in the last sentence is equated with the key of each datum, as defined by Codd’s ‘Guaranteed Access Rule’; that is, table name, column name, row key. To this we need to add a time component. So we can now define our version of context as:

- [where you are], the location of the datum as defined by its key;
- [who you are with], each other datum that shares the key;
- [what resources are nearby], what data is reachable from that key, where reachable means that a single value can be read with integrity (the navigation path must not traverse collections, for instance).

For the time component, we must read the value in the correct sequence as the current context requires.

Context as just described is critical to the problem under discussion. The context of every datum used in every calculation must be known with certainty at the time of its use, and all such contexts must be correct relative to all other contexts used in the calculation.

If we were an art collector we might think of context as the ‘provenance’ of the datum; that is, the ‘record of its origin’ at the time the value is read[10]. This is not just the value and its key; the provenance describes why and how the value was selected for use in this calculation at this time. For instance, selecting one value for one calculation in our Example Use Case might need to take into account not just the owning ‘customer’, but such other factors as the state of the customer (retired or not?), the type of fund they belong to, their investment preferences, their fee status (paid up?), their current entitlement (perhaps as a threshold), multiple date boundaries, and so on.

Within a calculation, the data often need to share the same provenance – for instance, by being selected for the same set of high order keys, according to the same value comparisons, classifications and other

[8] http://www.thefreedictionary.com/context

[9] http://www.cc.gatech.edu/fce/ctk/pubs/PeTe5-1.pdf

[10] http://www.thefreedictionary.com/provenance

non-key criteria, and perhaps for the same date range. This 'provenance' is NOT visible in the key of the datum, nor in the datum itself.

We have another example that demonstrates this in practice. A government front-line department had miscalculated average daily pay for a decade, thereby underpaying all termination payments. The average daily pay had to be recalculated for the entire period from original source data. So in the new process, daily accumulators were created for every day that an employee was employed. Then every payment that spanned any given day needed to be added to the day's average on a pro-rated basis: for instance, weekly and overtime earnings, shift allowances, various monthly, quarterly, and annual adverse condition and other bonuses, annual and long service leave etc. The 'provenance' of each and every payment made had to be assessed against the provenance of each individual daily accumulator – that is, the context of every payment to the employee had to be matched with the context of the daily accumulator, while that specific accumulator was being processed.

Our Example Use-Case calculation processes thousands of rows per customer (just one dependent table in the relevant database has an average 1000 rows per customer) for potentially hundreds of thousands of calculation values per customer. These data exist in a universe that is defined by the full extent of the database, which in this case multiplies the per customer contexts by all of the customer primary keys (at least).

The natural 'set processing' orientation of SQL means that all customers are 'reachable' with a single statement, which significantly extends the scope of the context problem.

If we are to process 'longitudinally' across the full extent of the database by processing each intermediate calculation for all customers, we introduce a scenario where the calculations need to be stepped, query by query. This stepwise processing is artificial, introduced as a consequence of the set orientation of SQL. It is not inherent to the calculation itself because the calculation does not inherently require the retention of intermediate values (e.g. any more than we need to retain the intermediate '7' if we are adding 3+4+5). And with stepped calculations we have now introduced a time component to the context, because our proposed definition of context above could deliver different values for the same key after each step.

Implications of Context

Different SQL developers, even experienced ones, frequently interpret context in different ways resulting in different outcomes. And there is not just one context.

Our approach uses one critical reference point that we call the current context – this is the context of the output value that will be derived by the currently active calculation component. The current context can temporarily move as we reach the boundaries of reachability (see "what resources are nearby" in the discussion above). When we reach into a collection, then we need to temporarily move the current context into the collection and do a sub-calculation; then on leaving the collection again, we need to reset the context back to what it was.

Context creates a more complex situation any time we need to access multiple values. Context is not just the key of a value; context is having the right key at the right time as the current context requires. We need to continuously validate that the context of every value read in our queries is appropriately

matched with the current context. For instance, if the current context is date sensitive, then the related contexts usually must also be selected according to the same date constraints.

We have described our Example Use-Case as having a hierarchy of calculations; each calculation has a hierarchy of operations on values. Every value must have the correct context (including the time dimension) relative to the current context at the time it is included in the calculation. There are many contexts, all morphing in concert as the current context requires. Regardless of how we do it, this is an expensive process.

Embedded SQL

Given this complexity, most developers capable of doing so will resort to a program for the calculations rather than stay exclusively within SQL, in which case multiple SQL queries are likely to be embedded in the program and executed on demand as separate queries when and as required. As noted, each query will need to establish or re-establish context for every value that is to be used in, or returned by, the query. This means that query logic (for instance, selection within date boundaries) is likely to be repeated as contexts are re-acquired and reused. Simply re-establishing context (the location of exactly one value at a point in time) may require multiple joins or complex row-by-row processing to re-collate the values that define the context. Repeated determination of context is expensive in terms of database processing, and complex in terms of ensuring correctness.

Using a native language program as described above mitigates, but does not avoid, the performance or complexity issues related to context.

This approach also carries an additional downside – the SQL and its host program are tightly coupled. These are not discrete programs - each is dependent on the other, and must be developed, tested, and deployed as a single integrated system image, which increases complexity, and eventually, cost, time, and risk. T Capers-Jones[11] has analyzed tens of thousands of projects, and provides overwhelming evidence that program size is the basis for a corresponding exponential growth in development cost, time, and risk, so that two discrete and separate programs are less costly and risky than one larger one that incorporates the functions of both. Increasing complexity eventually translates into inertia for the business and is a contributing reason as to why IT can become a business impediment.

Is the SQL Correct?

Our experience is that complex SQL queries are prone to errors that are not easily found in the normal course of SQL development. It is possible that the skill level of individual SQL developers is more varied than generally expected, and so some errors occur simply through inexperience. This suggestion is supported by the frequent reference in the literature to the need for a high level of SQL skills – for instance, this is the opening statement in an Oracle SQL practitioner's guide[12]:

[11] http://en.wikipedia.org/wiki/Capers_Jones

[12] http://docs.oracle.com/cd/E38689_01/pt853pbr0/eng/pt/tape/task_UsingSetProcessing-07720a.html

> "You should be a SQL expert if you are developing row-by-row programs with Application Engine and especially if you are developing set-based programs. . . . If you have a complex SQL statement that works functionally, it may not perform well if it is not tuned properly".

Even without the extended context dimensions described earlier (i.e. considering all primary entities at the same time; and retaining intermediate values across context changes through time), it can be difficult for the SQL programmer to maintain a clear view of the context of each value at the precise instant that the value is used in a calculation, and even more difficult to prove.

For this and other reasons discussed above, SQL offers a conceptually difficult testing problem. Possibly for this reason, and especially surprising considering that four decades have passed since the birth of SQL, there is relatively little third-party automated testing support for SQL.

Essence of the Problem

At this point, we suggest that any solution strategy that requires multiple complex SQL queries is likely to come at a higher processing cost, and to have a tendency to be more error prone. These effects imply higher development and operating costs.

This is not the fault of SQL. We believe that it is simply that the fundamental nature of the problem is transactional, not set processing; SQL is fundamentally set processing, not transactional. What we actually need to do is process all of the variables for each customer; not all of the customers for each variable. When we process each customer as a discrete unit of work, we are executing it as a transaction. We are also reducing the complexity of the processing context by the dimension represented by the number of customers (at least).

However, a quick survey of the literature will confirm that SQL and its underlying database processing engines are optimized for set processing rather than sequential processing. The power and appeal of SQL lies in its ability to plan and execute complex queries, where complex is used here to mean concurrent access to more than one table and/or access to one table using other than a unique index. And it is only natural for SQL programmers to be tempted to use the power of SQL when solving problems.

There is a fundamental misalignment between entity focused transactions and SQL set processing.

This may not be the fault of SQL; but complex SQL queries come at a cost.

THE SOLUTION

The Role of SQL

Simplify the SQL

It is intuitively obvious that the most efficient way to process this problem from a database perspective is to read every required input variable from the database exactly once; and to NOT write any temporary variables into the database at all. When we can do this, we have by definition achieved the lowest possible database overhead. This observation suggests that our aim should be to reduce all SQL queries to single table reads through existing unique indexes (i.e. no joins; any join implies multiple reads of the parent table).

In doing so we would acquire all data for each customer, and then process that customer as a stand-alone transaction entirely in memory. The customer key should already exist as a unique index on all its dependent tables, and so this approach has the smallest possible read cost in a SQL world. The SQL that is needed to read the required data from each table for our sample problem is very simple and looks like the following:

*Select * from Customer_History where Customer_History = @Customer_Number*

Our approach is to simply use the above single-table approach to read into memory all of the customer data, table by dependent table. This is also much faster; the same server that takes >48 hours to execute our Example Use-Case can deliver the required customer data for all customers using the above approach in around 3 minutes. And because we can process each customer independently, we can use as many processing streams as necessary to match this delivery rate with similarly fast processing of the calculations, no matter how complex.

XML

How do we capture and process the customer data that we retrieve in the above proposed solution?

Our preferred approach is to load the incoming customer data into an XML object (a Document Object Model or DOM), table by table. XML allows us to build a complex multi-dimensional structure that can hold collections of dependent database records in a single, schema defined object in memory. This multi-dimensional structure cannot be replicated using SQL.

The XML schema is important as it provides us with a ‘context map’ for each and every datum, a map which will be used to guide and manage the development of the calculations.

This XML schema and its equivalent runtime object will mirror the database relations that exist for each customer, so that the relational integrity of the database originated data is retained. The forced parity of the database and XML images ensures that only ‘simple SQL’ is required. In fact, we can generate this SQL from a simple mapping configuration, and then execute a generic, high performance conversion of the retrieved SQL results into the customer XML object (and back again if required). This essentially removes the SQL complexity, and its associated risk of errors, from the approach.

Furthermore, the XML is easily and dynamically extensible, so that any variables that do need to be temporarily stored within the transaction process (perhaps so they can be reused, or while waiting for other values to be calculated) can be simply added to the existing XML object, including (if needed) new collections and key combinations that are not supported by the persisted database structure. For instance, generating and apportioning values into various time periods (e.g. average daily rates, hourly costs, etc.) is a frequent requirement of payroll and/or billing systems. This requires that the time periods be created and populated dynamically within the calculation, often with a dependency on some other multi-valued customer data.

A further advantage of the transaction oriented, XML based solution is that the atomic transaction is easily reused: for instance, in an online process as a single stand-alone transaction; and/or it can be bundled into streams for parallel processing on any scale.

We recognize XML has a poor reputation for memory and processing overhead, but with better than order of magnitude performance improvements already demonstrated using this approach, this overhead is easily forgiven. And the ease of use, industry support, and sheer simplicity of the concept make it a favored choice.

Abstraction of Data

The complete abstraction of the data into an XML transaction record that fully describes the customer not only mitigates the downside attributable to tight coupling as described in the 'Embedded SQL' section above, it introduces some powerful and important new advantages.

1. The calculations are now independent of the database and can be developed in an entirely separate and distinct life-cycle by business aligned SME's.
2. The data and rules can change independently. By extending or swapping the mapping, large variations in the source systems can be accommodated. For organisations relying on vendor supplied systems, this can dramatically simplify vendor system support and upgrade issues. It is even plausible to migrate a system (or version of a system) out from under the abstraction and replace it with a new mapping to a new system; there need never be another legacy system.
3. By further expanding the abstracted 'view' of the data across additional external systems, we can supplement the core transaction data with other data sources: for instance, in an insurance example, data might be sourced from a doctor's 'Patient Management System' and attached to a health insurance application; or customer credit details might be sourced from a credit agency and attached to a loan application. This can expand the scope of the calculations and the utility of the overall system without making the existing 'core system' itself more complex; in fact the core system need not know that the data extensions are even occurring.
4. Separation and abstraction of the data into XML means that the client can mix and match data sources by location and by type – database one minute, file system the next (literally). For instance, we can store the XML itself, either in a database, or on the file system, to provide a complete and virtually cost free audit trail of the completed calculation and all of its component sub calculations.

5. We can also re-read and reprocess the above stored XML, which can lead to an even simpler and more cost-effective SQL processing cycle – one read of the XML to obtain all of the data in context! At the very least, we can store this XML for both development and regression testing purposes.
6. And with complete XML records available and easily accessible, it is a simple matter to extend their use to supporting simulations and what-if testing of variations in the calculations. This can then be integrated into the business process of developing the underlying business policy itself, so that IT becomes an active and involved enabler of an agile business rather than a constraint on it.

Exceptions

So why hasn't everybody done this? Why are there complex programs developed entirely in SQL?

We think that advantages inherent in our approach as discussed so far are widely (but not universally) recognized. In a traditional development environment, the calculation engine would usually be built in a standard computer language. For project sponsors targeting smaller, localized problems, this may mean more complex development projects with additional implied cost, time, and risk; so perhaps, it is sometimes easier to accept that a SQL programmer can deal with the whole issue without the organizational overhead implied by a larger multi-disciplinary development project. Or perhaps sometimes the SQL fraternity simply adopts a project and provides a solution, Zen like.

And realistically, for simple problems, a SQL centric, or even SQL only solution is a reasonable option to take. Our discussion here is about when complexity should dictate a different solution approach – these are shades of grey. In our view, many practitioners initially underestimate the degree to which intermediate variables are calculated in a complex process, as per our Modern Analyst[13] article. Because of this, a seemingly simple problem can easily escalate into a more complex issue during development. Like the slow cooking frog[14], we should remain conscious of a subtle increase in complexity that can 'cook' a simple SQL processing solution into a beast of complexity[15].

A Tool Assisted Approach

A New Approach

There is another element to our solution that helps to address the issues raised above.

We introduced the concepts behind a declarative decisioning tool in our article in Modern Analyst: 'Requirements and the Beast of Complexity[16]'. The Beast article laid the foundation for decisions to be the fundamental unit of requirements specification, now defined as follows:

[13] http://www.modernanalyst.com/Resources/Articles/tabid/115/ID/2713/Decisioning-the-next-generation-of-Business-Rules.aspx

[14] http://en.wikipedia.org/wiki/Boiling_frog

[15] http://www.modernanalyst.com/Resources/Articles/tabid/115/ID/1354/Requirements-and-the-Beast-of-Complexity.aspx

[16] http://www.modernanalyst.com/Resources/Articles/tabid/115/ID/1354/Requirements-and-the-Beast-of-Complexity.aspx

Decision: A single definitive outcome that is the result of applying business knowledge to relevant data for the purpose of positively supporting or directing the activity of the business.

According to the article, these decisions are defined in the context of a decision model, which is defined as:

Decision Model: An ordered assembly of decisions that creates new and proprietary information to further the mission of the business.

And decisioning itself was defined as:

Decisioning: The systematic discovery, definition, deployment, and execution of computerized decision making.

The above decisioning concepts were expanded in a second article to include data transformation as a critical part of any business rules solution (see 'Decisioning – the next generation of business rules[17]'). Data transformation is required to create data that is aligned with the nouns and noun clauses of a proprietary idiom (a. 'A specialized vocabulary used by a group of people'[18]) that we asserted is always present when describing business rules. The idiom is always proprietary to the author of the business rules – by definition, because it is the need to define the business rules that gives rise to the idiom in the first place.

And to close the circle, we can observe that the idiom's nouns and noun clauses are actually the labels for the various values in context. Using terms that identify not just the value, but the value in context as described by its 'provenance', is logical if we are going to be able to describe the business rules using the idiom. So the nuances of context are exactly matched by the nuances of the idiomatic language that is used to describe the fundamental rules of the business. The rules of context are the rules used to create the business' proprietary idiom and vice versa.

As noted, an organized set of decisions is a decision model. If the bias of the decisions in the model is towards calculating new data that 'supports' the activities of the business, it can be considered to be a 'calculation engine' as per our Example Use-Case – a calculation engine is simply the generated form of a decision model, and as such it will include an array of data transformations that dynamically construct the idiom, and end up producing the ultimate decision outcomes that add value to the business. On the other hand, if the bias of the decisions is towards 'directing' the activities of the business, it could equally be considered to be a 'workflow engine'; usually a decision model has elements of both calculation and workflow in the same model.

This paper is now extending the decisioning concept (as described by the two earlier papers) by highlighting context as an integral part of a decisioning 'requirements specification'. With a decision authoring tool that inherently manages context, we can quickly and completely define an end-to-end calculation and/or workflow engine that leverages simple SQL to provide complete and error free calculations of any complexity; and these calculations are likely to execute orders of magnitude faster than solutions developed using a more traditional approach that attempts to use the power of SQL in its more complex forms. This performance gain is primarily a reflection on the way that SQL is used.

[17] http://www.modernanalyst.com/Resources/Articles/tabid/115/ID/2713/Decisioning-the-next-generation-of-Business-Rules.aspx
[18] http://www.thefreedictionary.com/idiom

Managing Context

Our approach uses a 'schema aware' decision management tool to provide the SMEs – the subject matter experts who actually define the calculations – with a drag and drop GUI to graphically build the calculations directly over the XML Schema defined 'context map', without needing to know any XML navigation syntax or other programming skills.

The schema provides our context map; this map is active within the tool, so that the complex task of understanding and controlling context is actively managed by the tool. The SME will only be able to reference data that are valid for the current context. The current context is the locus of the calculation that is currently in focus, and is always a single, unique point on the schema defined context map. The tool will ensure that there is always exactly one current context for any operation, moving the current context as required and allowed, and validating the relative contexts of all other variables. At every step, both the current context of the calculation variable and the context of every variable being consumed in the calculation should be visible.

The calculation and its related logic are graphically captured as a 'decision model' using the tool's decision model and formula palettes. A decision model is strictly hierarchical, executing left-to-right/top-to-bottom so that the time element of context is graphically displayed and easily controlled.

The tool must allow the SME to define new variables at any time. These plug into the schema defined context map as required, and are usable in the same way as any other value.

As described in the earlier articles, the tool must also allow the SME to test any part/whole/group of calculations directly in the GUI development palettes during the development process, with context being graphically displayed as the test execution takes place. The abstracted XML records described above can be used immediately and directly for this purpose.

When finished, entire libraries of test cases can be regression tested directly from the decision model palette, before generating the program and deploying it with a single click for system testing using real data. Again, this is a use for the abstracted XML records described above.

Because the decision model and its associated schemas provide a complete transaction specification, the required high performance and well documented 'calculation engine' program can be completely and automatically generated in source code form 'without fingerprints'.

The following outline summarizes the 'new approach':

1. Develop an XML Schema to describe the 'real-world' entity that is the subject of the transaction, to be used as a 'context map' for subsequent process development.
2. Use a series of 'simple' SQL queries to collate a complete, in memory XML object as defined by the schema. These queries can be generated from a simple relational-to-XML mapping. The reverse mapping can also be generated for write back if required.
3. Use a drag and drop GUI to declaratively build out the calculation process, using the schema defined 'context map' as an active and extensible requirements template.
4. Test the evolving process 'in situ' to ensure correctness, completeness, and consistency at all times.

5. Regression test in the same tool to confirm absence of unintended consequences; and/or run simulations to verify that changes in the underlying business approach achieve the desired objectives.
6. Generate and deploy without manual intervention.

Further Effects of the Approach

Building a complex process using the above approach has many other positive downstream effects.

1. The 'decision model' is tightly structured, and able to be rendered into many formats, including logical English, computer source code, and XML. It is therefore transparent, auditable, and reliable as a specification of the executable code.
2. This is a tool for the SME. The high degree of automated assistance and reduced development complexity (particularly around context) means that the SME is only contributing what they already know – their own proprietary business knowledge. Productivity is quickly and reliably achieved; arcane IT skills are not needed and offer no advantage.
3. Generation of code means that the code per se does not need to be tested; only the logic supplied by the SME needs to be tested. This means a substantial reduction in testing effort.
4. The automated tool assistance and reduced human input reduces the potential for errors. This gain is compounded by the extensive on board testing support already described, so that it is rare for a decision model to be anything other than complete, correct, and consistent when released.
5. The decision model itself, and all the artefacts that support it, fully support perpetual and continuous versioning. When the models are implemented as 'content' in computer systems, they provide a practical and robust mechanism for business SMEs to independently develop, manage, and deploy the organisation's codified knowledge (including complex calculations, workflow, and other business policies) on a daily basis – continuously and perpetually.
6. The decision models are technology agnostic and technology independent, forming a complete historical record and the ultimate 'source of truth' for the organisation's proprietary knowledge; extracted, tested, confirmed, and documented by that organisation's SMEs in the normal course of business.
7. A tool assisted approach as described allows SME's to capture, test, and deploy this knowledge extremely quickly. In a 100 developer year project, 80% of the system code was generated from decision models that were built in 20% of the total development hours – all of which were contributed by business analysts drawn from business branches.
8. This development efficiency is sustainable over the long term, offering huge business agility with reduced cost, time, and risk
9. A decision model is a durable life-long artefact that defines the business in perpetuity. As a complete record of organisation knowledge, and with multiple and extensible source code generation options, there should never be another 'legacy system'.

CONCLUSION

We acknowledge that the transactional solution approach described in this paper is oriented towards a particular class of problem – the 'complex, entity centric transaction'. However, in recognizing this, we also suggest that the same class of problem is not a suitable candidate for a solution that requires complex SQL – where complex is intended to mean anything other than simple, index supported single table reads.

This is because complex SQL is associated with excessive database overhead that can in some cases degrade performance by an order-of-magnitude. This scale of performance variation has been shown empirically and repeatedly in our projects.

And there is another important reason for many organisations. The lack of transparency in complex SQL with its potential for hidden errors can be an existential risk in some industries. Again, this statement is empirically supported by our real-world projects, where replacing complex SQL based calculations has frequently unearthed long-lived errors in production code.

The conclusions above are amplified by the overwhelming predominance of 'complex, entity centric transactions' in commercial processing. Some generic categories include the following, with existing real-world processes supplied as supporting examples:

- External event driven processes that cause a state change in the target entity for any industry:
    - Examples: Underwrite and rate insurance; Accept a claim; Calculate an entitlement; Approve a Loan; Calculate medical invoices; Pay a contract; Reset next intervention date for a clinical pathway; Calculate municipal rates and charges; Calculate travel ticket; Calculate frequent flyer rewards; Calculate telco/utility charges; Approve entry for border control.
- Time based processes that cause a state change in the target entity:
    - Examples: Renew insurance; Follow up a claim; Charge a fee; Age based adjustments; Place a loan into delinquency; All period end processing for a superannuation fund.
- Validate, audit, and/or remediate existing entities and their legacy processes on a large scale:
    - Examples: Use the approach to confirm correctness, completeness and consistency of relevant entities over their full history for industries with specific legal obligations towards increased accuracy (for instance under APRA's CPG235[19]), such as pension funds, payroll managers, and trust funds.

Finis

[19] http://www.idiomsoftware.com/uploads/gallery/temp/1383188444DataIntegrity-Financial%20Services5.pdf

Author's Bio:

Mark has more than 35 years history in software development, primarily with enterprise scale systems. During the 1980s Mark was actively involved in the development and use of data- and model-driven development approaches that later achieved widespread use. Application of these approaches to the development of an Insurance system for one of the world's largest insurers was formally reviewed in 1993 by the University of Auckland who concluded that "this level of productivity changes the economics of application development."

In 2001 Mark led a small group of private investors to establish Idiom Ltd. He has since guided the development program for the IDIOM products toward even more remarkable changes in "the economics of application development." He has had the opportunity to apply IDIOM's "decision oriented" tools and approaches to projects in Europe, Asia, North America, and Australasia for the benefit of customers in such diverse domains as superannuation, finance, insurance, health, government, telecoms, and logistics.

IDIOM Bio:

Established in 2001, IDIOM Limited is a private company based in Auckland, New Zealand.

IDIOM develops and licenses decision-making software that automates business policy on a large scale, making systems more transparent, agile, and durable, while reducing development cost, risk, and time.

IDIOM's innovative business oriented software is used by business users to graphically define, document, and verify corporate decision-making and related business rules; it then auto-generates these into small footprint, non-intrusive software components for use in systems of any type or scale. IDIOM is a pioneer in the development and use of decision automation concepts, and has applied these concepts to develop and automate business policy for customers around the world in local/state/central government, insurance/superannuation/finance, health admin/clinical health, telecoms, logistics, and utilities.

IDIOM automated business policy and decision making extends far beyond mere business rules, so that larger and more complex decision making can be fully delegated to business experts. IDIOM enabled development and management of policy based decision making by policy owners creates a propitious 'business policy life-cycle' that significantly improves business agility and transparency.

IDIOM develops and licenses: IDIOM Decision Manager™, IDIOM Forms™, IDIOM Document Builder™ IDIOM Mapper™, IDIOM Tracker™, and IDIOM Decision Manager Workbench™.

Author's Contact Details

Mark Norton | Director | Idiom Limited |
Office +64 9 630 8950 | Mob +64 21 434669 | After Hrs +64 9 817 7165 | Aust. Free Call 1 800 049 004
1-8 93 Dominion Rd. Mt Eden Auckland 1024 | PO Box 60101, Titirangi Auckland 0642 | New Zealand

Email mark.norton@idiomsoftware.com | Skype Mark.Norton |