

# OLAP Friendly OLTP Design: *Planning Ahead For The Inevitable Data Warehouse*

*June 2009*

*Matthew S. Oldham, [matthew.oldham@mmillerassociates.com](mailto:matthew.oldham@mmillerassociates.com)*

*M. Miller & Associates, [www.mmillerassociates.com](http://www.mmillerassociates.com)*

## INTRODUCTION

"By failing to prepare, you are preparing to fail." – Benjamin Franklin

Regardless of what data you are collecting and storing in your OLTP applications, someone is going to want to report on it eventually. As mainstream as email in corporations today, the data warehouse has become one of the pillars of IT infrastructure. If your company does not have a data warehouse, chances are either they really do but you just aren't aware of it, or there are plans in place for a data warehouse of which you have not yet heard. This means that you will eventually discover I/O intensive queries pounding your OLTP databases in eerily periodic fashion at odd hours of the night. If your design is worth its salt, and if the designers of these intensive queries are as good at design as you are, you will also hopefully discover that the duration and impact of these queries is minimal. The first discovery I have described is inevitable. The second discovery will be determined by you, the OLTP designer. That being the case, we must begin to architect our applications and databases with the expectation that they will be tapped for corporate information by the data warehouse. I firmly believe that we must make this a habit and core development practice. Whether you are a database designer, a pure middleware (i.e. Java/NET) programmer on the back end, or a "user experience" designer on the front end, you have a role to play in good OLTP design.

The following list of practices is presented in no particular order and is not intended to be exhaustive. My hope, in fact, is that the list grows as experts such as you read this whitepaper and respond with anecdotes of best practices you have found to be effective in your own experience. I will also say that as the list grows, the less practical it will become to incorporate all of these practices into every single design. You will encounter limitations caused either by the technical infrastructure in which you are implementing, or by existing IT policies to which you must adhere. I can, however, offer you the hope that once these practices become a pattern in any such environment, they will be noticed, they will be leveraged, and they will be a catalyst for change in the right direction.

### ❖ Use Audit Date Stamps for Change Data Capture (CDC)

Change Data Capture. The ability to determine what data has changed since last I queried it. This is a critical function of most data warehouse Extract, Transform, and Loading (ETL) architectures since it dictates whether or not one can incrementally update data warehouse tables using only source system activity that transpired in the last 24 hours as opposed to rebuilding these tables from the ground up each night with all historical activity. The difference in batch processing time and CPU cycles between these two approaches can often be measured in orders of magnitude. Recent product releases by large ETL and RDBMS vendors tout new features supporting this functionality. Some of these features are quite complex in their implementation and require enabling certain product options that have their own implications and must be

carefully considered prior to flipping the CDC switch. One basic design consideration that will solve most of the CDC challenge for your designs is the consistent use of audit date stamps on your tables. Virtually every table I create in my designs has two standard columns: `INS_DTTM` and `UPD_DTTM` - when the row was first inserted, and when the row was last updated, respectively. When possible I also populate these columns using pre-insert and pre-update triggers on the table. This removes any dependency on the application code to populate them, and also ensures that they do not get populated incorrectly. This concept can be extended to include the name or ID of the user who triggered the create action for the row, as well as the name or ID of the user who last updated the row. As of late I have also begun incorporating an “audit version” column in my table designs – something I learned from one of our clients. This attribute is also maintained by pre-insert/pre-update table triggers. Audit version (or, `AUD_VER`) is initially populated with a value of 1 when the row is first inserted, and is incremented by 1 for each subsequent update operation. Not only does it tell me when the row was inserted and when it was last updated, but I now know how many times the row has been modified since its creation. This nifty little attribute has come in very handy during test phases when performing analysis on what data has changed. In applications where workflow is involved, `AUD_VER` can also be quite useful in identifying where a particular row is in the state transition process. I have even begun carrying this attribute forward into my data warehouse designs. Simple, elegant and unobtrusive, audit columns like these I have described are a small price to pay for an effective CDC strategy.

### ❖ Take Advantage of Table Partitioning

One of the most seamless and yet positively impacting improvements to many RDBMS platforms in the recent past is the table partitioning option. While not an available feature in all RDBMS's, table partitioning has performance and administrative benefits that we as OLTP designers should consider for the long term success of our underlying databases. The relevant benefits that this option brings to the table for us are improved performance and easier administrative operations. You might be thinking to yourself, “I’m not a DBA, so why should I care about administrative tasks?” Well, as advantageous as table partitioning is, even good DBA's don't always think about it. The more you know about the capabilities available to you the better off your applications will be.

From a performance perspective, the benefits of table partitioning are very significant. Consider a table in your OLTP application that contains sales orders. Using Order Date as the key partition column, the table could be range-partitioned by week, month, or even year, depending on your needs. A query selecting orders for a particular month, for example, would only access the partition containing the data for that month. If the query selected orders over a window of two months, only two of the partitions in the orders table would be accessed. This process of query isolation is called partition pruning. The database does the work of determining which

segments (partitions) of the underlying table are needed to service the query. When Data Warehouse queries begin to access the orders table, the impact to the OLTP database is now much more likely to be limited in scope. When combined with the ability to partition indexes and leverage parallel execution in some database platforms, table partitioning brings a lot to the table.

From an administrative perspective, table partitioning introduces the ability to perform common maintenance tasks such as backup and recovery, statistics generation, index rebuilds, and bulk data loads on a per-partition basis as opposed to the entire table. This provides improved efficiency and reduces the impact of administrative activity in the database.

And, I have saved the best for last. Accessing a partitioned table from your application is identical to accessing a non-partitioned table for normal DML (i.e. CRUD) operations. That means you can partition an existing table underlying your OLTP application without having to change a single line of code. It is fair to say, however, that the degree of performance improvement you experience in your application as the result of partitioning is determined by a number of factors. If your tables are small, and/or if you have partitioned on a key column that is not used in any query generated by your application, it's likely you won't see any improvements in performance. This is all the more reason to educate yourself early on the capabilities of your underlying database. If you design with partitioning in mind you can be sure you will be recognized by others as that sage architect who was thinking long term.

### ❖ Use Surrogate Primary Keys

Without a question – in all cases and at all times – use single column, sequence-generated (i.e. surrogate) primary keys on your tables. The primary reason to do this is that legacy numbering schemes or natural keys (e.g. Customer ID, PO Number, etc.) that originate in operational systems are outside of your control and should therefore not be trusted or relied upon. As soon as you make your design dependent on something you don't control, trust me, it will change and your design will break. Save yourself and those you leave behind in your wake the headache of such a scenario and use surrogate keys from the beginning. If your RDBMS supports sequence generators or IDENTITY columns, use them. That's what they are there for. I typically let my pre-insert/update table triggers manage this work for me since they are already in place as part of my CDC strategy.

Now, having said that all of that, it is still important to take care of those natural keys by preserving their integrity and indexing them appropriately. The Data Warehouse will rely upon them heavily when creating Slowly Changing Dimensions (SCD). Once your data makes it to the Data Warehouse, natural keys are the usually the last surviving link back to the originating operational system.

## ❖ Rationalize as you Normalize

Normalization – A wise DBA once told me: “Normalize ‘til it hurts, de-normalize ‘til it works.” I have found this strategy to be very practical in my own designs. The key is finding the right balance. Reduce or eliminate as much redundancy as is feasible, and leave as much as you need to make the application intuitive and well performing. A healthy level of de-normalization will also benefit report writers and, ultimately, data warehouse ETL developers. There is no hard and fast rule to defining “healthy” for any particular design, but a good designer will recognize excessive normalization and be forward-thinking enough to trim it down. As outlined in the previous section, one particular strategy to keep in mind when it comes to normalization is the use of single column primary keys as opposed to multi-column (composite) primary keys. Here’s a general rule of thumb: if, while writing SQL against your database design, you find the number of join criteria exceeding the number of tables in the FROM clause, you have probably over-normalized. It’s very easy to let your data modeling tool propagate foreign keys to resolution tables (i.e. the entities that resolve many-to-many relationships) as the primary key of the table. There are a few problems with that approach, however. First, it violates your surrogate key design practice which dictates that you always use sequence-generated primary keys on your tables. You may be thinking to yourself, “What could one or two resolution tables with multi-column keys hurt? If I do reference them in a SQL statement, it’s only one extra join criteria anyway, right?” Well, not always. More times than I expected, I saw resolution tables I designed grow into parent entities to new tables that cropped up as the business evolved and the application matured and was extended. As this happened, my once leaf-node-like tables became branches to other segments of the database that I now could only access by referencing those extra primary key columns for every ancestral entity of the original resolution tables. Had I initially implemented a single column primary key as the identifier of these resolution tables, the resulting SQL statements would be much simpler – both for me and for every other poor SQL writer who queried my database.

## ❖ Put Referential Integrity (R/I) Where It Belongs – In the Database

Referential Integrity, or RI, is the set of rules that prevent inconsistent data in a database. This set of rules belongs in the database, not in your application. Early in my IT career I worked with a very large ERP application. This was a fantastic opportunity, and I am grateful for the experience I gained from it, as it opened the door to many other great opportunities. I remember thinking to myself as I dug into the internals of that particular package how well it was designed. What I did not realize at the time but soon learned was that all of the referential integrity in the application was embedded in the application code – not in the robust database on which it was deployed. As a result, I began to see distinct job titles of the flavor “Application DBA” appear

on the job boards over time as customers who had implemented this application were faced with addressing the major performance and maintenance issues to which this design led. I now appreciate the benefits of database constraints (or, declarative RI). We cannot rely on the application alone to preserve data integrity. It may be the single point of entry now, but what happens when a third party application or interface is introduced that influences the data in some way? All of the sudden that cleverly designed layer of application-level RI is of no effect. Regardless of what a developer does to the application code, I know I can rely on the database constraints to protect the consistency of my data. That's right, you heard me correctly. I said "my data". As OLTP designers we have to take ownership of the data and become stewards of it to preserve its integrity at all costs. It does not matter which role you play in the OLTP design. If the application design to which you contribute (and with which you are now forever associated) results in inaccurate reporting or inadequate user confidence due to poor data quality, consider the consequential "black eye" your own. It is your reputation as an application designer that will suffer as much as the end user of the application you helped design.

### ❖ Enforce Data Quality Constraints

Continuing the thread of data stewardship, we must also enforce data quality constraints in our OLTP designs. Unique to this practice is the fact that it can be applied at all layers of the design. For example, the UI can protect data quality in the form of "edits" which are invoked at the data-entry level and prevent data from ever getting *to* the database. Likewise, the database can (and should) implement NOT NULL and other various CHECK constraints to prevent invalid data from getting *into* the database.

Regardless of where it gets implemented, we must prevent bad data from ever getting into the database. Once it gets in, there will forever be an exception that must be considered when analyzing the data. While it is true that data cleansing can be applied at the ETL layer prior to loading the Data Warehouse, data cleansing rules are difficult to establish and will immediately introduce a difference between the data in the two systems. Such differences always raise questions and will therefore implicate the need for an accompanying set of explanatory documentation outlining the reasons for the differences.

Another practical reason to prevent invalid data is the sheer effort in CPU cycles and ETL batch processing hours required to parse and cleanse data. This valuable time will better serve the cause of CDC and other necessary extract, transformation and loading activities.

### ❖ Don't Forget Indexes

Indexing in OLTP designs is often an afterthought, especially as new systems are deployed to ever improving multi-CPU hardware platforms. Fast hardware will mask performance problems

related to lack of proper indexing for only so long, however. Data volumes will inevitably increase, and data consumers will contrive unique ways of getting data out of your database that very quickly point out the need for indexes. Specifically, Data Warehouse ETL processes will inherently leverage an appropriately indexed OLTP design. For example, remember those CDC columns we discussed earlier? They are perfect examples of columns that should be indexed to support ETL processing. And don't forget about all those foreign key columns that are now part of your RI regimen. Many RDBMS platforms will automatically index primary key columns, but foreign keys are usually overlooked. Not only will such indexes make ETL processing fast, but the I/O impact on your OLTP system will be greatly mitigated because a quick index scan will likely replace the full table scan that would otherwise have been required.

### ❖ Use a Consistent Database Object Naming Convention

Every database designer has their preference when it comes to object names. I believe an effective table naming convention is of particular importance as you create your forward-thinking design. As an ETL developer, many times I have found myself trying to infer meaning from an OLTP database by perusing the data dictionary. One of the biggest contributing factors to gaining an understanding of what I found was the table naming conventions that were used. If I see a table named ORDER\_DTL, a couple of things become immediately clear to me. First, I have in all likelihood just found the Order Detail table, and second, I should easily be able to find another ORDER-related table that serves as the parent entity to ORDER\_DTL. Maybe ORDER, or perhaps ORDER\_HDR? Either way, I just put together an important piece of the puzzle. A quick internet search will result in a plethora of thoughts around how to name your objects. As a matter of personal preference I like to see objects named in such a way that the prefix qualifies and the suffix clarifies. Taking the example of ORDER\_DTL, I immediately discern that the table contains order data and further that this particular table contains the detail information for orders. Other common suffixes I use are \_LOG for log tables, \_STG for staging tables, \_LKP for code and description lookup tables, and the list goes on. Another benefit to using the object prefix as the qualifier is that when viewing a list of sorted object names, those of the same subject type sort together and one can get a quick idea of the extent of the subject-related data in the database.

This concept carries over to other objects types as well. In the ORDER\_DTL table, I would expect to see a column named ORDER\_ID, which constitutes the foreign key referencing the primary key of the parent entity ORDER, and ORDER\_DTL\_ID, which is the primary key of the ORDER\_DTL table. This paints a picture of the relationships inherent in the design. Don't discount the importance of the decision for how to name your database objects. It's worth the time up front to establish a convention that is informative. It doesn't have to be fully complete or extensive to begin with, but a little forethought in the data modeling phase is a small price to pay for the time savings you and others will realize down the road.

## ❖ Metadata – Not Just For Data Warehouses

Metadata is not just for data warehouses – it begins with the source system. At least to some degree, an effort should be made by every OLTP designer to deliver a data dictionary containing business definitions of all entities and attributes along with the database. A data dictionary is the simplest form of a metadata repository, and will naturally flow downstream to the Data Warehouse. Business definitions are a critical component of solid data warehouse architectures, and should originate from a well designed OLTP source system. Collect them early on in your design and expose them in a fashion that allows end users access to them. It is true that one of the major downfalls to any data dictionary effort is the failure to keep it updated as things change, however if you incorporate metadata-type documentation into your development methodology, it mitigates exposure to this risk. One easy way to incorporate business definitions is by tagging your database objects with comments. RDBMS platforms such as Oracle and SQL Server support such comments nicely and in a way that makes them accessible to users with SELECT access to the data dictionary. Leverage these features. I am pretty religious about creating table and column comments simply because it is more or less a “metadata freebie”, compliments of the RDBMS vendor. A valuable side benefit of doing so is that I force myself to understand the data I am storing because a business definition must be gathered for every element of my database design before my build script will complete successfully.

## ❖ Maintain Lookup Tables

This may sound obvious, but be sure to store codes (or enumerated lists) in properly designed lookup tables. These lookup tables should contain corresponding description columns which contain accurate business descriptions for each respective code. There are many schools of thought and preferences over how to design such lookup tables. For example, one could decide to break it down into short name, long name, short description, long description, etc. However far you ultimately decide to take it, just keep in mind that a code without a description is only valuable to your application code – not to an end user or data analyst assigned to report on this data. Further, each code in an encoding scheme (or lookup table) constitutes a natural key. When a Data Warehouse dimension table gets built to store such codes, the descriptions are typically extracted alongside the codes in order to provide a means of CDC within the code set. In the case of a Slowly Changing Dimension table, a change in the description for a particular code would trigger the inactivation of one dimension record for the prior “version” of that code, and the creation/activation of the new version.

Also part of the design debate for lookup tables is whether to have one table per code set, or instead a single generic lookup table (also known as the “One True Lookup Table”, or OTLT) to store all code sets. I encourage you to research this debate on your own to learn about the pros

and cons of the OTLT approach, but I will offer one of the reasons I choose not to revert to this type of design. Having separate lookup tables for each code set affords me the ability to specialize each of them for the respective code set. For example, my “state/territory code” lookup table needs a time zone attribute in order to satisfy a certain business requirement. At the same time, my “status code” lookup table needs an attribute that indicates whether or not a particular status code is active (let’s call it “active indicator”). If I were using the OTLT design, each state/territory code would have to also maintain an “active indicator”. Likewise, and perhaps even worse, each status code would have to maintain a time zone attribute. Not only is this confusing and nonsensical, it is also very inefficient. Further, any specialization requirements that get introduced for a single code set automatically affect every other code set since all are stored in the same table. In my opinion this is a bad idea and I recommend avoiding the OTLT approach altogether.

## **CONCLUSION**

Do you hear the recurring theme? It’s simply this – design with forethought. The bottom line is that you will eventually have to eat your own dog food. Like it or not, your expertise and assistance will be sought after at some point in order to help those tasked with extracting and/or reporting from your application database. You may even be the one asked to develop the reports. Make it easy on yourself, your fellow team members, and those you leave behind to support your applications, by designing with forethought. The incorporation of relatively simple design concepts can mean the difference between a highly accessible, rich data set, and a closed system that limits corporate information accessibility. Make reporting from your applications an intrinsic possibility by laying an appropriate foundation. Things have a way of coming full circle and you will eventually be found with a debt of gratitude to yourself for doing things the right way the first time.

Perhaps you have certain conventions or practices that are part of your design methodology. Please feel free to send me your feedback, including any additional thoughts and ideas around optimal OLTP design that paves the way for the inevitable Data Warehouse.