

Design Considerations for Transparent Databases

Sandy Steier

1010data, Inc.

January, 2010

© Copyright 2010 by 1010data, Inc.

Introduction

The typical database implemented on 1010data's platform is an example of what I call a "transparent database." What is a transparent database? I recently Googled "transparent databases" (with the quotes) and got a relatively small number of hits, most of which didn't use the words in a particularly insightful way. This isn't surprising since the subject of this essay is not, as far as I know, one that has gotten a lot of recognition.

For our purposes a database is **transparent** if its data and table structure is visible to end users. Conventional, large databases are typically not transparent. End users usually access information through applications that intermediate between the user and the database and hide some of its data and architecture. This is expected and even logical given the nature of most database management systems (DBMS). Interacting with a conventional DBMS requires not only familiarity with database concepts in general and the specific protocols of the DBMS in particular, but also a deep understanding of how database design details impact query performance. Queries written by lay users can be inefficient and have deleterious effects on the system.

Often, too, a database's architecture is designed to be data-centric rather than application-centric so that it will remain useful and relevant even as business needs change. By definition, this further distances end users, who tend to be application-oriented, from the details of the schema.

But despite the theoretical and technical arguments, in many cases end users do in fact directly access databases. It may be because the appropriate applications have not yet been built or that the end user needs to do some sort of temporary ad-hoc analysis, but it happens pretty often. In fact it would undoubtedly happen even more often if the technical barriers were lowered and if the right kind of data were stored in the database.

I should point out that this discussion is focused on analytical rather than operational databases. Operational databases are used to collect data rather than analyze it so transparency is generally not an issue. Generally data is taken out of operational databases via **ETL** (**Extract, Transform and Load**) processes and stored in a data warehouse, the latter being specifically designed to support reporting and analysis.

The archaeological record from the iron, i.e. mainframe, age (which includes me) reveals an interesting fact. On the one hand there were many databases (on in-house, company mainframes) that were highly technical and often application specific, and those da-

tabases were clearly off-limits to end users. But there were also databases (also on mainframes, but not in-house and not even owned by the user company) that were meant to be used by end users. These databases (and mainframes) were maintained by **timesharing** services that offered a relatively inexpensive way of housing databases and a flexible, *transparent* way of accessing them. (As an aside, several of the timesharing services used a columnar database model, a paradigm that is only recently regaining attention in the computer world at large.)

The timesharing model eventually declined and largely disappeared and was replaced with the modern in-house data warehouse concept. Although external hosting of databases has made a comeback - witness the Cloud - the concept of transparent databases has not. But there is movement in that direction and recent technological advances, such as those developed by 1010data, have made the paradigm quite compelling.

When end users can play with a database directly, all sorts of interesting things happen. Analysts come up with new ways of looking at things. Previously neglected data becomes valuable and key data becomes even more valuable. Even application programmers benefit in that they can focus on real, persistent applications without getting diverted by day-to-day fire drills. Transparency is also a great way of prototyping applications so the development process can be smoother and faster, and result in a better product.

Clearly, if a database's architecture and data is to be visible to users, some traditional design concepts need to be re-examined. That is the subject of this paper.

Raw Grain

One of the most important considerations when designing a database is what preprocessing is required. In a typical ETL process, "raw" data is extracted from source operational databases, transformed by a set of **business rules**, then loaded into the data warehouse. The business rules form a sort of filter through which the raw data is passed, both testing the data and, in some cases, cleaning it. Business rules may, for example, define what values and formats are valid in a particular column and how to fix bad values or formats. Other rules may state whether a column can have missing values and, if there are missing values, how to "fill them in". The obvious goal is to turn messy source data into clean, useful information so that applications don't have to worry about data quality.

Another important consideration is the level of detail to be captured, sometimes called **grain**. Will the database store transaction-level data or summary data? If the latter, what is the summary level? For example, in a retailer's data warehouse, information might be stored at the store level, the department level, the sale ("basket") level or the item (e.g. two-liter bottle of Pepsi) level. The answer generally depends on anticipated reporting requirements and technical feasibility. Storing greater detail can impact query performance and sometimes make analysis more complicated, but storing less detail can

render other analysis impossible. Data at its lowest level of detail is often called "atomic."

Conventional databases, including data warehouses, are designed with certain queries or at least classes of queries in mind. Thus it is possible to determine, in the **requirements** and **logical-design** phases ¹, the requisite level of grain and business rules that will govern the database. Aside from logical considerations, the performance constraints of most DBMS also argue for careful consideration of grain and ETL processing in that both finer grain and less preprocessing lead to longer query response times. Finer grain implies more data, which in turn implies longer query run times. Similarly, less processing during ETL means more processing later on, i.e. during query run time. So for a variety of reasons it has become standard practice to incorporate both business rules and non-atomic grain as an integral part of database definition.

Which brings us to transparent databases. Because the whole point of transparency is to allow a user to do completely ad-hoc analysis, it is clearly advantageous to store atomic data. Any summarization will lose information that the user may find useful and *the very definition of ad-hoc analysis means that even the users cannot know what their requirements will ultimately be*. After all, sometimes you need to see the forest and not the trees, sometimes you need to see the trees, and sometimes you need to see both. But there is also an advantage in taking it a step farther and storing *raw* data since the very process of filtering and cleaning results in a loss of information. Granted that the information loss is not as great as in summarization, but it can be important. An example will help.

Suppose that we have a table in a weather database that shows the temperature for various locations, dates and times of day. Most sources of data have issues so let's assume that this case is no different and that there are missing values due to occasionally malfunctioning equipment. Since most applications that query the table would appreciate it if such anomalies were fixed, it would seem like a good idea to have an ETL business rule of the following sort,

If a temperature value is missing, fill it in with the average of the previous and following values for the same location.

That would be a pretty good estimate of the temperature and most analyses will give good results. For example, a chart of the temperature will look good with a smooth line that is undoubtedly a close approximation of the truth.

But what if we want to investigate which locations have malfunctioning equipment? Once the ETL process has "fixed" the data, that information is no longer available.

Simply stated, the fact that data is missing or invalid is *itself* information. Since we cannot anticipate what a user may want to do, and therefore what the user will find useful, it would seem prudent to simply store the raw data and let the user decide what to do with it.

At this point you may have a couple of objections.

First, you may ask, is the weather example really a fair example? It is after all numeric, scientific data and such data is not necessarily the same as, say, text, business data. If the table were a customer table that listed, among other things, the customer's address, would it not be reasonable to have a business rule like,

If the name of the city is missing but the ZIP code is not, use the ZIP code to determine the city and fill it in.

This is likely to produce an exactly correct value rather than an approximation and there is no malfunctioning equipment involved, so why not get rid of that pesky missing city name? What possible reasons could there be for knowing that the city name was not originally filled in?

I can think of at least two. First of all, there is a sort of malfunction going on here. Clearly some data entry person (or cashier or teller or clerk) neglected to enter the name of the city. Perhaps we would like to run a query to find which such people have a habit of leaving things out? It might be valuable to have a chat with them and encourage them to be more careful. More importantly, if the person was sloppy enough to not enter the city, can we be sure that the ZIP code is correct? It might be interesting to correlate the occurrences of missing values in some columns with incorrect values in others.

These examples relate to missing data and are a bit arcane, but they make the point that transformations result in a loss of information. It is easy to come up with more pertinent examples with respect to value and format transformations.

A better objection to storing raw data has to do with efficiency and ease of use. If 95% of queries will benefit from implementing business rules, doesn't it make sense to do that and not worry about the rest? Why should 95% of applications have to deal with bad data? Effectively, we are pushing the responsibility for implementing the rules into applications. If many applications have to implement the same rules, aren't they reinventing the wheel each time and, worse, isn't it likely that not all of them will get it right?

The answer to the last question is, of course, yes. However the question presupposed that, if rules are not implemented in the ETL process, each application has to implement rules independently. There is in fact a third way. Instead of applying the rules on the way *into* the data warehouse, it is possible to apply them on the way *out*. Applications that require clean data should not access the raw-data tables directly, but rather use views or, if necessary, alternative tables containing clean data. Such tables would be created from the raw data tables in the data warehouse and would be updated after the ETL processes move the data from the operational databases to the warehouse.

The advantage to using views as opposed to real tables is that there is no chance that the tables get out of sync. Also, views don't require update time. So the choice of views or tables depends on the number of business rules, the size of the database, and, most im-

portantly, on the efficiency of DBMS. I'll discuss this subject a little more in the next section.

To sum up, transparent databases argue for less transforming in the ETL process (how about "EtL" or even "EL"?) in favor of keeping raw data in the warehouse and implementing the transformations *on top of* the warehouse.

Preaggregation and Indexing

For a transparent database to be successful, the DBMS must allow arbitrary, ad-hoc queries and it must be fast. If the DBMS is slow or if it constrains the range of queries or even if it allows a broad range of queries but they need to be optimized or structured in particular ways, there will be little value in direct access by users. These technical requirements are the reason, of course, that traditional databases have usually not been transparent.

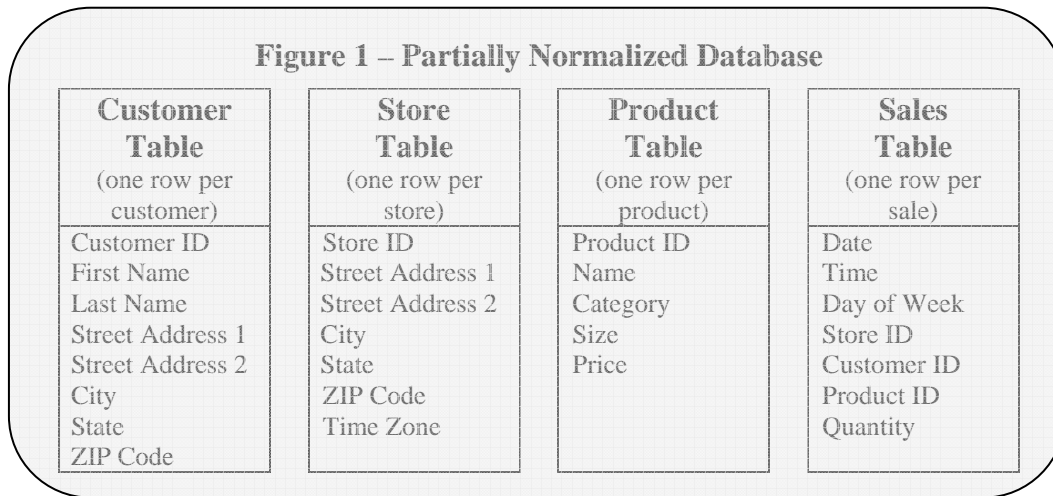
Assuming then that the DBMS is fast and unrestrictive with respect to queries, there are several implications for data warehouse design. For one thing, not only is OLAP-style preaggregation unnecessary but, as alluded to in the previous section, it also doesn't fit the paradigm. By their very nature, OLAP cubes limit the kinds of analysis that can be done since they contain very specific pre-computed results. In fact, with a fast DBMS that allows analysis of raw data, most types of data marts are unnecessary.

A reliance on indexing is also best avoided since it dictates what can be done to which columns. If certain queries work only with certain columns, the benefits of transparency are reduced. Occasionally it makes sense to get an extra performance boost by indexing heavily-used columns, but queries should ideally work reasonably quickly on all columns.

Having said all that, precomputation is sometimes helpful. Some databases are so large and some queries are run so frequently that it makes sense to conserve CPU cycles and virtual memory usage by precomputing certain results. (But, per the previous section, it is better to do it after the raw data is loaded into the database instead of during the ETL process.) Depending on the intended use of such precomputed results, it may or may not make sense to make the results themselves transparent. If they are meant as stepping stones for users, so that users can build their own analyses on top of them, they obviously need to be transparent. If however they are designed to speed-up a particular application, it may be better to hide them from users; that way, if the application's requirements change, the precomputed results can be modified without worrying that an ad-hoc user will be affected.

Table Structure

Broadly speaking, table design typically revolves around the question of **normalization**. Without getting overly technical, in a normalized data architecture every class of information has its own table. For example, a typical retailer's database may contain tables describing each customer, each product, each store and each sale.



This structure is both logical from a user's perspective and neat in the sense that there isn't a lot of repeated information. Contrast this to a fully **denormalized** version of this database where there is a single Sales Table that contains all information about every sale including information about the customer, store and product.

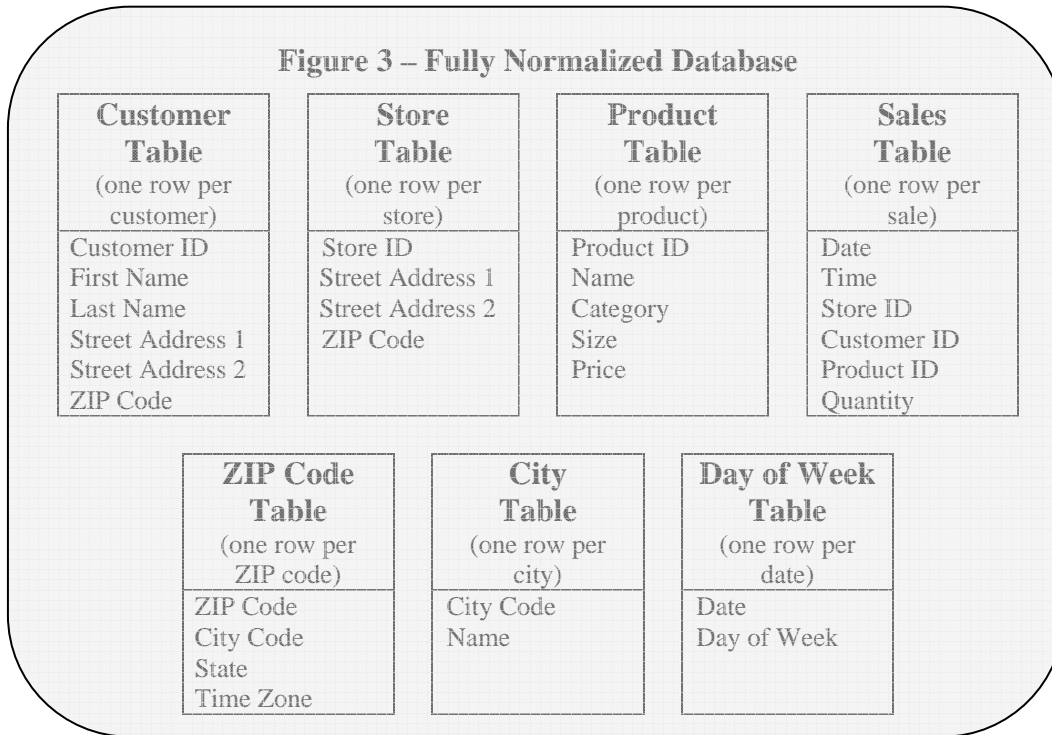
Figure 2 – Fully Denormalized Database

Sales Table (one row per sale)
Date
Time
Day of Week
Store Street Address 1
Store Street Address 2
Store City
Store State
Store ZIP Code
Store Time Zone
Customer First Name
Customer Last Name
Customer Street Address 1
Customer Street Address 2
Customer City
Customer State
Customer ZIP Code
Product Name
Product Category
Product Size
Product Price
Product Quantity

While this table may seem useful for certain queries, for example determining how many sales happened in California, it has many obvious disadvantages. For one, a lot of information is repeated many times. It is both messy and inefficient to repeat a store's address for every sale, especially when there may be millions of sales for that store. It also makes many queries more difficult, such as determining how many customers live in California. And, with respect to queries that seem to require a denormalized table, such as our earlier example of determining the number of sales in California, every database product allows one to **join** tables, so it is a simple thing to join the Sales Table to the other tables thereby creating a *logical* version of the fully denormalized Sales Table.

It is also possible to take normalization further by organizing the data into even more tables. Following this process to its logical conclusion results in a **fully normalized** database.

Figure 3 – Fully Normalized Database



Summarizing some of the advantages of normalization ²,

1. A normalized database can take up significantly less disk space. This is a natural consequence of not repeating data.
2. A normalized database allows no (or fewer) potential contradictions and other inconsistencies. This is also a consequence of not repeating data. If the data were repeated, as in Figure 2, it is possible that the same city could be spelled differently in different rows or that it could be assigned to different states. A customer's name or address could also be inconsistent across rows. (For the purposes of this example we can assume that customers do not change their names and do not move.)
3. Perhaps most importantly, in a normalized database many kinds of queries are easier, particularly those that don't require the Sales Table. Not only are they simpler to express but they often run faster since the Sales Table is much larger than the others.

As we are about to see, however, these advantages are not always as important as one might think and they have diminishing returns, plus there are sometimes outright disadvantages to normalization.

Advantages 1 and 2 above are generally true when contrasting a partially normalized database with one that is fully denormalized, but the marginal advantage declines as more and more data is normalized. The difference in disk space between the architecture in Figure 3 and that in Figure 1 is marginal at best, in fact certain aspects of Figure 3 actu-

ally require more disk space. The addition of the City Table, for example, probably costs a small amount of space in that we have replaced the city name in the ZIP table with the more-or-less equivalent City Code and then added the City Table. So from a purely pragmatic point of view, Figure 3 does not really provide much benefit in terms of disk space or data consistency.

Data compression, where available, also closes the gap between denormalized and normalized data with respect to disk space. Compression algorithms automatically get rid of most duplicates so the fact that a denormalized table logically has a lot of duplicate data doesn't mean that the data is actually duplicated on disk.

In addition Advantage 2 turns out to be something of a red herring in that data integrity can usually be enforced programmatically. Even if the source, operational data contains inconsistencies, the ETL process that loads the data warehouse can resolve them. (Remember, we are talking about non-operational databases.) In fact if the source data contains inconsistencies, the ETL process *must* resolve them if it is to recast the data in a normalized schema.

Advantage 3 is certainly important when it comes to formulating certain queries, but it doesn't always lead to better query performance. Depending on the database management system architecture, *many queries actually run significantly faster when applied to a denormalized table*. The reason for this has to do with inefficiencies in the way joins are sometimes implemented and, more insidiously, with the dramatic difference between sequential and random disk I/O. The differences in disk I/O is something that deserves more attention than it typically gets ³.

As a result of all this, data warehouses tend to be architected in a way that preserves the essential aesthetics of normalization while allowing for performance considerations. Star schema, for example, are semi-normalized forms that reduce the number of joins required for analysis. But, modulo performance considerations, the design process remains oriented towards greater normalization, which makes sense given the practical advantages of normalization, the reigning data-centric philosophy, *and the fact that databases are not transparent*.

The user experience, a consideration largely absent in conventional, large database design but important in a transparent database, argues for more limited levels of normalization. Even assuming that the DBMS can perform well with high degrees of normalization, and although a highly normalized schema may be attractive for theoretical and aesthetic considerations, an analytical end user is more likely to be interested in ease of use. The user would probably prefer to avoid having to examine and then join a myriad of tables.

So what is the bottom line? In general, in a transparent database, tables should mirror larger, real-world concepts rather than be motivated by abstract standards like 3NF ⁴. Users naturally take to the notions of customer, product, sale and so on, so organizing the data around such concepts provides a friendly environment for analysis. It also facilitates

most types of queries, uses disk space efficiently, and satisfies theory up to a point. As a counterexample, take the City Table in Figure 3. I would guess that such a table would confuse a non-technical end user and would appear overwrought. As I pointed out earlier it also has no practical benefit anyway.

Of course the user experience cannot be the only guiding principle: I have seen cases where query performance considerations strongly argued for a *less* normalized (or even fully denormalized) model and disk-space considerations can clearly point in the opposite direction. All these things need to be taken into account. But organizing the data according to real-world concepts and the end-user experience should be the starting point of the design.

Availability During Updates

There is a tricky issue that affects most analytical databases. Imagine that you are an analyst using an application that runs various queries and combines the results into a report. Now suppose that the data in the database is changing while the application is running. Each query is applied to what is effectively a different dataset and the resultant report will lack integrity. Since every database must be updated at some point, there are going to be times when a database isn't really fit to be used.

Fortunately, unlike operational databases that are updated in real time, data warehouses are usually updated in some sort of batch cycle. Once a day, say, data is extracted from an operational database and loaded into the warehouse. But even the batch update can take time and, during that time, the database is in a semi-updated and changing state.

With conventional databases, the set of applications is finite and they can often be run at certain known times. The obvious solution then is to not run them during database update windows. In fact, databases are often made unavailable at predefined times of the day (or weekend for weekly updates) so that integrity issues don't arise.

With transparent databases the problem is exacerbated in that there isn't really a well-defined set of applications and users may want to play with the data all day and night. (Most likely this will arise when users are in widely different time zones but I've known some individuals who, by themselves, could work through a twenty-four-hour period – or even longer!)

Digging a little deeper, there are really two levels of concern. The first and most serious is consistency within a given table; if one is running queries on a table and the table itself is being updated, data in any given column cannot be trusted. For example, if new rows have been added but obsolete rows have not yet been deleted, numbers may add up to more than 100% of what they should be.

Inconsistency can also occur between two tables when one has been updated and the other has not. If the two tables are joined, the combined information will be out of sync and could give incorrect results. While of concern, this is less likely to be as problematic

as an internally inconsistent table. Since any given column comes from exactly one table and each table is internally consistent, the column's data must be internally consistent. It is more likely that some of the columns will simply be out of date.

Focusing on the first, more serious problem, the obvious solution is to deny access to a table while it is being updated. To eliminate downtime, a *copy* of the table may be updated while the original table remains unchanged. Eager users can continue to use the original even as the copy is being updated, so they don't experience a service interruption. Once the copy is finished updating, it can replace the original so that users will see the new version instead. This switch can happen instantaneously so at no time will a user see an inconsistent table in the midst of being updated. We call this scheme the **A/B method** because it involves two copies (A and B).

Figure 4 – A/B Method

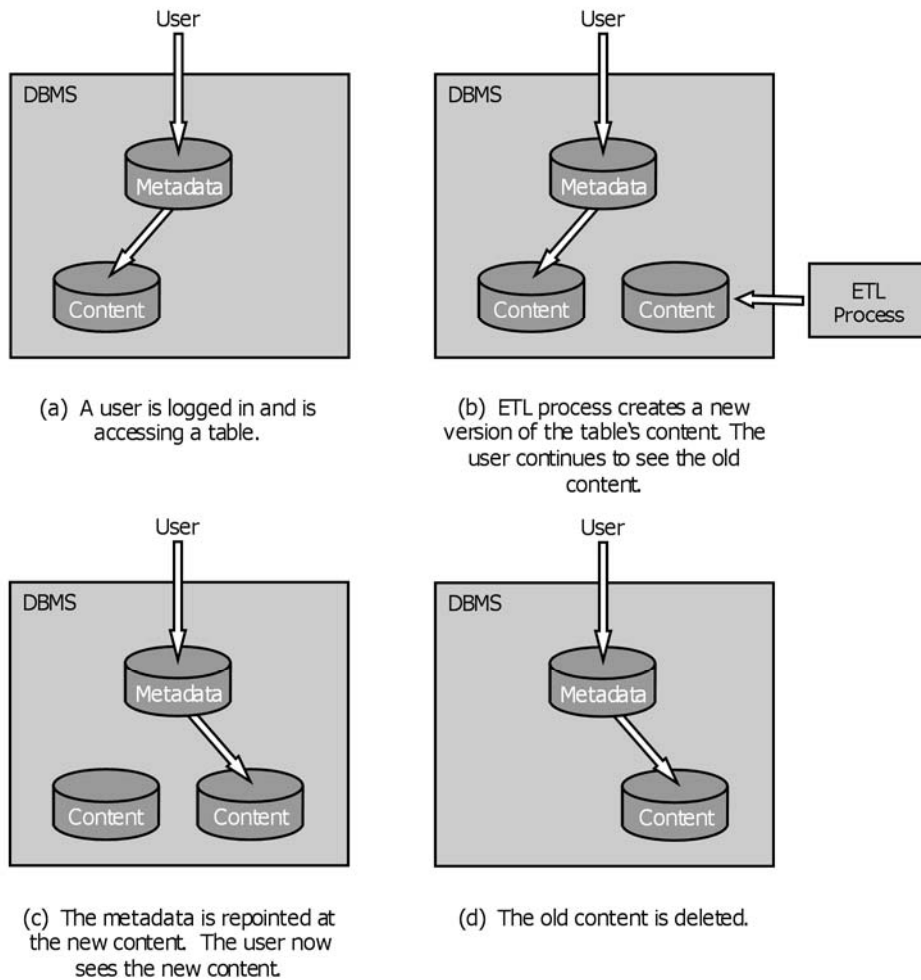
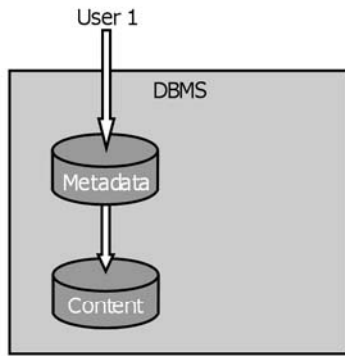
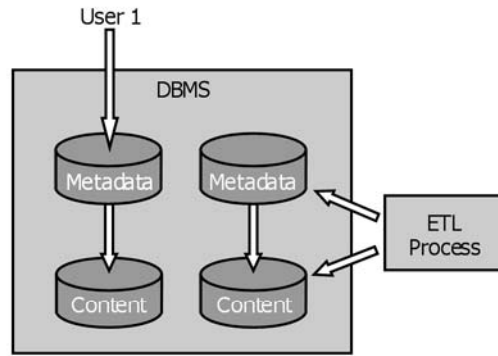


Figure 4 illustrates the A/B method and also points to an inherent problem. While the user never sees an *internally* inconsistent version of the table, at step (c) the table suddenly changes. Any analysis done prior to (c) will be inconsistent with that done afterwards. Ideally a user should continue to see the *old* version until he or she takes specific action to switch to the new. That way any analysis that the user is doing will be valid (although perhaps not completely up-to-date.) What we need then is a sort of extended A/B method, where not only is a table updated via a copy but where a user with an active session continues to see the old copy throughout the session. In other words, the information regarding a database must be kept as part of the user's session information or "state". Let's call this the A/B method with state, or **A/B-S** for short. Figure 5 illustrates this method.

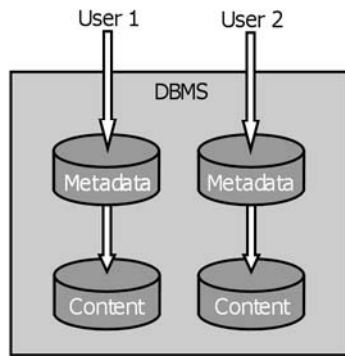
Figure 5 – A/B-S



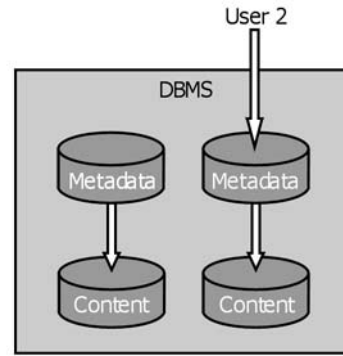
(a) User 1 is logged in and is accessing a table.



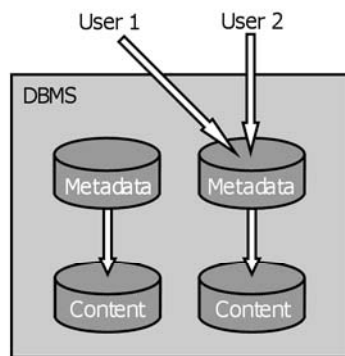
(b) ETL process creates a new version of the table. User 1 continues to see the old version.



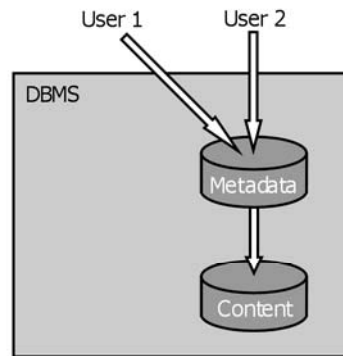
(c) User 2 logs in and sees the new version of the table. User 1 continues to see the old version.



(d) User 1 logs out.



(e) User 1 logs in again and sees the new version.



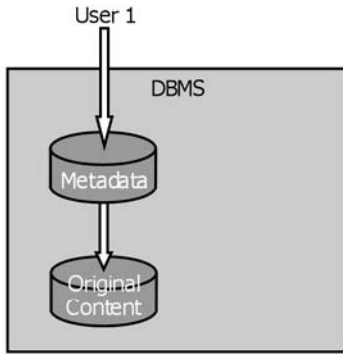
(f) Once no users are using the old version, it may be deleted.

In many traditional database systems A/B-S would be tricky to implement, but 1010data handles it easily, in fact automatically. Whenever a table is updated, by either replacing the entire table or by appending rows to an existing table, a user who is already logged into the system does not see any of the changes for the duration of the session. Instead, the user continues to see the original copy of the table as if no updates happened. Of course, once the user logs out and then logs back in, the new version will present itself, so the user has control over when to switch to the new data.

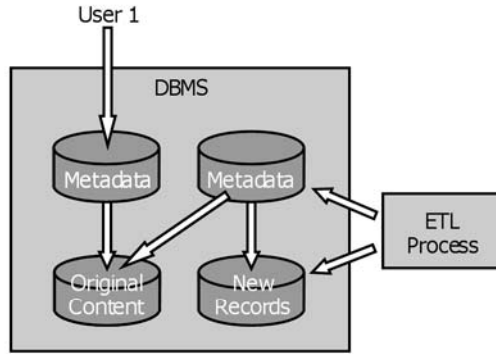
In the previous paragraph I mentioned that A/B-S is appropriate for both a complete replacement of a table and an append. The latter requires some explanation. To be efficient, an append should "touch" very little data (only the appended rows) and not involve copying the entire table. This would appear to be incompatible with an A/B method but it may not be, depending on the DBMS. In 1010data, for example, it is possible to "copy" a table by reusing the same data files as the original, so both the original table and the copy point to the same data on disk. This allows A/B-S to be applied to appends because, although both tables initially point to the same data, only the copy will end up pointing to both the old data and the appended data. The old data is not physically copied (see Figure 6.)

Now let's talk about the problem of consistency between tables. When one of two tables has been updated and the other has not, the combined information will be out of sync. As I said earlier, this isn't necessarily a critical issue, which is fortunate because there isn't a perfect solution. In theory, one could apply A/B-S to a set of tables, that is to update copies of all the tables and make them visible once they are *all* completed. The problem is that it may take a long time to update all the tables and users may want to access the new data as soon as possible. The value of getting the data faster may outweigh the potential issues that arise when different tables are out of sync, so simply updating and releasing one table at a time is often the best strategy. Importantly, A/B-S helps here as well. As long as a user maintains the same session, all tables that he or she sees will be the original versions that were available at log-in time and will be unaffected by updates. So assuming that the user doesn't log in in the middle of an update cycle, all tables remain in sync throughout the session.

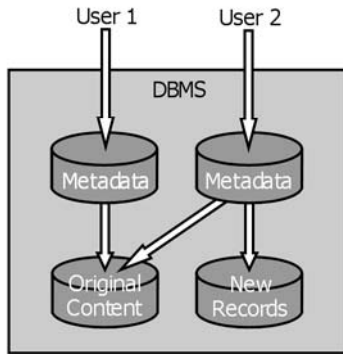
Figure 6 – Append Using A/B-S



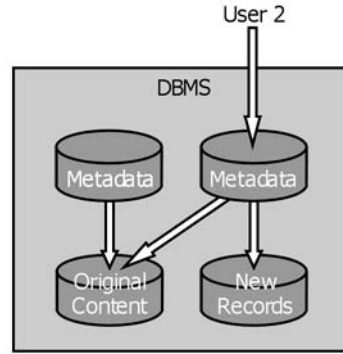
(a) User 1 is logged in and is accessing a table.



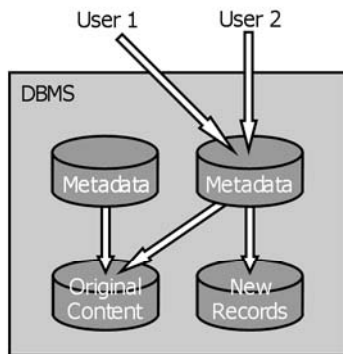
(b) ETL process adds new records and alternate metadata. User 1 continues to see the old version.



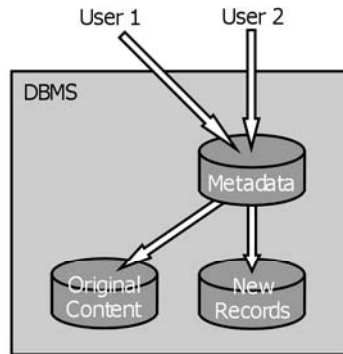
(c) User 2 logs in and sees all records. User 1 continues to see only old records.



(d) User 1 logs out.



(e) User 1 logs in again and sees all records.



(f) Once no users are using the old version, it may be deleted.

Updating Master and Transaction Tables

The content of virtually all tables changes over time. Some, like lists of states or cities don't change very often and a table of continents (as opposed to a table of contents) isn't likely to change at all, but given our discussion under "Table Structure" above, these kinds of tables probably don't belong in a transparent database anyway.

Although all tables change, the way they change divides them into two broad categories. **Master tables**⁵ contain information that is not specifically keyed to time, for example the Customer, Store and Product tables of Figure 1. Even though customer information may change, in the example we only care about the latest value. (In practice some "master" tables do have timestamped records and I will address this later.) Updates to such tables usually include additions (e.g. new customers), deletions, and replacements (corrections or changes.)

On the other hand, **transaction tables** contain information that is specifically keyed to time, a classic example being the Sales Table of Figure 1. Typical updates to such tables are additions (e.g. new sales), with deletions or corrections possible but less common. Even if a transaction were to be voided, most databases will keep the original record and add a new record for the void; the result is that no row is ever deleted. The same is true for corrections.

Given our previous discussion, in a transparent database tables should be updated by essentially creating a copy and replacing the original table (the A/B method). How does that relate specifically to master and transaction tables?

With regards to a master table, the answer is simple: The table should be entirely reloaded, via an A/B copy, from new source files. Such tables are relatively small so complete replacement is quite efficient, especially given a fast load mechanism like 1010data's. Also, because records can be added, deleted or changed, the cleanest and safest approach is to get the latest version of the truth from the source and reload it. Often in fact the ETL process doesn't even know what data has changed; unless the source database keeps a log of changes and feeds that information to the ETL process, there is no practical way for the process to sift the new source data for changes. In that case, the process has no choice but to reload the table from scratch anyway. A/B-S in its simplest form is ideal for master tables.

Transaction tables are a bit more interesting. Such tables tend to be large and replacing them in their entirety may not be efficient or even possible given time constraints. Fortunately the very nature of transaction tables suggests a couple of simple and elegant solutions. Given that we are discussing analytical databases that are updated in a batch process (as opposed to a real-time feed), and given that historical records in transaction tables are not often deleted or changed, all we need to consider are batch additions. So one obvious way of updating such a table is to append the new rows to the existing table and, as I pointed out, A/B-S applies equally well to appends (at least in 1010data.)

An alternative to appending data to a single table is to create a new table for each update. The result will be a set of tables, each corresponding to one update. For example, if the data is updated on a daily basis, there will be a table for each day. This is particularly suited to environments where many queries are applied to only the latest information, i.e. the most recent period. Such queries can run more quickly and use fewer resources when applied to a small periodic (e.g. daily) table than when applied to a large table containing data spanning many periods. Note that this scheme does not preclude running queries against many periods. To create a single (logical) table for historical analysis, one merely needs to merge the periodic tables vertically, as in SQL's UNION or 1010data's native merge facility.

This is all very neat and clean but as you may have already realized, it is a bit too neat. There are in fact tables that are neither master nor transaction but a hybrid of the two. I am thinking in particular of tables that contain master-table type information but that also have a limited time dimension ⁶. Such a table may contain more than one row for a given item (e.g. customer), each representing a change to the information about that item. Figure 7 illustrates a simple example in the form of a more realistic customer table than that of Figure 1.

Figure 7 – Customer Table with Updates

Customer Number	First Name	Last Name	Street Address	City	State	ZIP Code	Update Date
1	John	Doe	558 58 th St	New York	NY	10045	6/14/06
1	John	Doe	4321 Palm St	Los Angeles	CA	90005	9/03/09
2	Susan	Smith	76 Oak Blvd	Cleveland	OH	44123	1/01/05
3	Mary	Jones	123 W 8 th St	Houston	TX	77010	2/26/03
3	Mary	Doe	4321 Palm St	Los Angeles	CA	90005	9/01/09
4	Homer	Simpson	742 Evergreen	Springfield	XX	00000	2/15/09

Note that there is more than one row for some customers. Customer #1 (John Doe) moved in September of 2009 and customer # 3 got married and changed her last name at approximately the same time. (Coincidence? I think not.)

In the simpler scheme of Figure 1, we would have simply replaced the old information with the new - and in many real-world applications that would be fine. But in other cases it may be important to go with the scheme in Figure 7 and keep both the old and new values. If we wanted, for example, to study customer loyalty to particular stores, the fact that a customer moved would clearly need to be taken into account. Transactions before a move will probably be in different stores than transactions after the move.

The question then is, how should such a table be updated? Replacing the entire table clearly does not work, so it cannot be updated like a simple master table. Creating a new table for each update is impractical since we will end up with a lot of very small tables, which is both ugly and potentially inefficient. This eliminates the second of the two methods described above for transaction tables. What we are left with is append with

A/B-S, which is actually fine. There could be some degraded efficiency using this method, for example in 1010data doing many small appends can result in fragmented physical storage and less than optimal performance, but this can be easily remedied by occasionally replacing the table with a copy of itself. The act of resaving the data will clear up any issues.⁷

Time Series

Time-series data is simply data keyed to time so all transaction tables contain time series. In particular, a **periodic time-series** is a collection of data representing observations at *regular* time intervals⁸. Some examples are monthly loan payments, daily sales figures, or hourly temperature measurements. Not every time period need have an observation, i.e. periods can be skipped, but on the whole there should be a clear grid pattern to time. In this sense, raw cash register data and stock trades are not periodic time series although they could be transformed into such by aggregating them into regular time buckets. For example, the average trade price per ten minute interval is a periodic time series.

Time-series, especially periodic time series, are often analyzed differently than other data. Queries like period over period comparisons and moving averages are most meaningful when the data is periodic. The common denominator in these kinds of queries is a comparison or other calculation *across* rows. It rarely makes sense to apply a cross-row calculation to a master table and, although such calculations may sometimes be appropriate for any type of time series, they are more typically applied to periodic time series. In this paper I will refer to all cross-row queries by the more analysis-oriented term, **time-series analysis**.

Conventional database technologies have long struggled with time-series analysis. SQL is set-oriented and sets are not ordered, so there is no notion of order in a standard relational database. The most fundamental property of time is that it *is* ordered so SQL and time are not naturally compatible. Various techniques have been employed to bridge the gap, such as joining a table to itself with the time key shifted in time, but they tend to be complicated and inefficient. Transparent databases should provide a more natural and efficient way of doing time-series analysis.

1010data is the only data warehouse technology that I know of that handles time-series analysis in an efficient, user-friendly way. In this section I will therefore focus on 1010data's solution that natively allows users to explicitly refer to multiple rows in a given calculation. Calculations like "subtract last period's value from this period's value and divide by last period's value, then multiply by one hundred" (to get a percentage difference) are directly expressible in the 1010data query language. For example, assuming we want to apply it to column "price", the above query could be written as,

```
100*(price-t_tshift(price;-1))/t_tshift(price;-1)
```

or, equivalently,

```
100*(price/t_tshift(price;-1)-1)
```

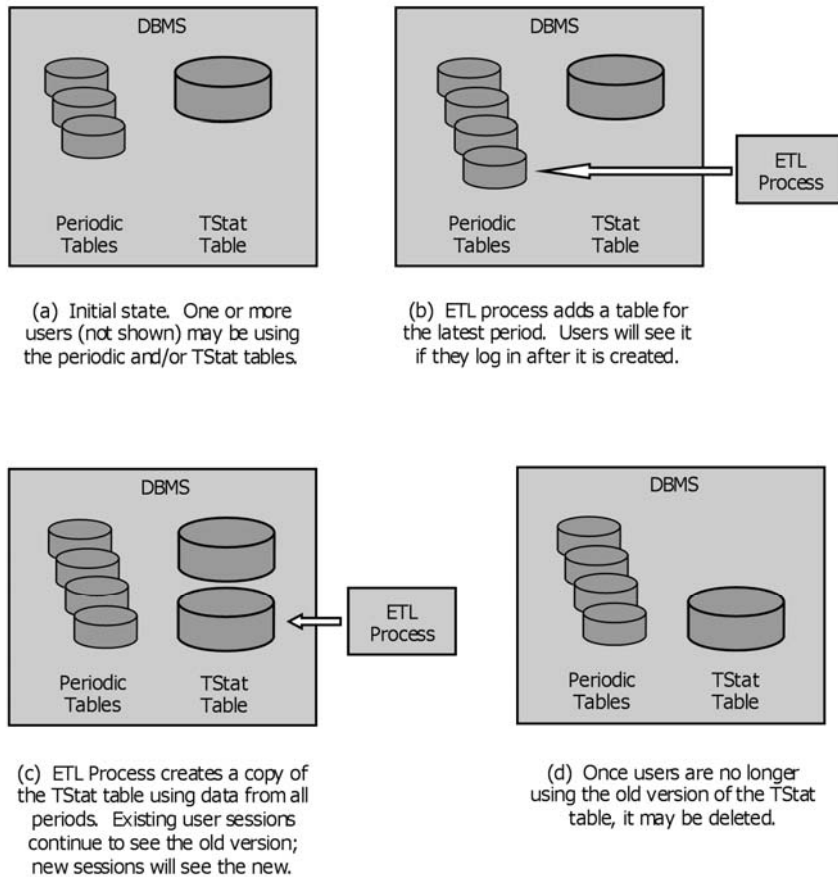
either of which is easy to understand and executes quickly.

To allow this type of simplicity of expression yet maintain high performance, the system imposes certain restrictions on the structure of a table. Without getting into the details of what that structure is, suffice it to say that it is *not* any of the structures we've discussed so far. That is to say that a table formed from the union (merge) of periodic (e.g. daily) tables *cannot* be used for time-series analysis, nor can a table formed by a series of appends of periodic data. Since, as we discussed in the previous section, these two methods are the best ways of updating transaction tables, you may well wonder where this leaves us. How can you update a periodic time-series table while allowing users to do time-series analysis?

The answer is you can't. Well, at least not directly. Starting with either a table formed via appends or a collection of periodic tables, the data must be copied into another table, called a "**TStat table**", that has the proper format. This extra step, although not difficult to invoke, does take some time and disk space, so it's something to keep in mind when planning for update duration and disk utilization.

A good strategy then is to maintain both a set of periodic tables and a TStat table that is derived from the periodic tables. Users then have the choice of using the small periodic tables when querying individual periods or the larger TStat table when doing multi-period historical analysis. Of course all updates should use A/B-S (Figure 8.)

Figure 8 – Updating Time Series Using A/B-S



Conclusion

Transparent databases narrow the gap between source data and end users, empowering them with more information. A prerequisite to employing such a strategy is choosing database and analytical technology that can handle the extra workload that comes from supporting truly flexible analysis of raw data. But, beyond the basic technological challenge, the paradigm impacts all aspects of database design, from ETL to table layout to update methodology. The good news is that there are existing, proven ways of implementing both the technology and the methodology. The future is *now*.

End Notes

1. These are standard steps in the database design process, wherein the architect gathers information from the user community, encapsulates it in various write-ups and diagrams, and echoes the information back to the users as a double check.

2. Normalization is critical in operational databases insofar as it facilitates updates. Our focus here is on non-operational databases.
3. See Adam Jacobs, *The Pathologies of Big Data*, ACM Queue Volume 7, Issue 6, July 2009.
4. 3NF, for Third Normal Form, is one of several forms of data normalization and probably the most widely adopted. Most 3NF tables are free of repeated data and the possibility of data inconsistency.
5. The terms "master table" and "transaction table" are in common use although there does not appear to exist a formal definition. The definitions used here conform to the essential properties of these types of tables. Those familiar with the terms "dimension table" and "fact table" know that they have similar meanings to "master table" and "transaction table", respectively. I chose not to use them here for a few, related reasons. First of all, "dimension" and "fact" are highly abstract terms used specifically in conjunction with star schema. Since I downplay the importance of highly structured databases and even star schema, it seems better to avoid such terms. Second, the term "dimension table" comes from the fact that they are look-up or reference tables that describe dimensions. But what I call "master tables" includes not just look-up tables but also tables that have significant, independent information content, i.e. someone can do real analysis using nothing but such a table. What distinguishes them from transaction tables specifically is that they lack a time dimension. Finally, "fact tables" can contain any measured data, even if none of the dimensions are time. For example a snapshot table showing sales by store and product is a perfectly valid fact table. Although they are uncommon, such tables can exist in standard databases. To me, non master tables always have a time dimension so "transaction table" is more descriptive and specific than "fact table."
6. A common term for this phenomenon is "slowly changing dimensions".
7. The real challenge with this type of table is not in updating it but in querying it. Joining such a table to a transaction table is tricky since the join is not based on an exact match-up on the time dimension. Rather, a given transaction must be matched to the *latest customer record whose update date is on or before (i.e. current as of) the transaction*. This is very difficult to do in standard relational databases but is directly supported by 1010data's **as-of link**.
8. The terms "time series", "time-series analysis" and "periodic time series" are used to describe different (although related) things by different people. The definitions used here are the most appropriate for this discussion.