Steve Rubenstein's Recurring Billing

Row Level Archiving

Version 1.1
Last Updated October 15, 2013

This purpose of this document is to document the various methods used in Steve Rubenstein's Recurring Billing used to archive data when it is changed.

Archival Requirements:

1. Provide audit history of records that have changed
2. Maintain referential integrity for historical records
3. Enable database join on a single field from foreign key to primary key

1. Provide audit history of records that have changed

Wherever possible and practical, the system stores all versions of customer data. For instance, if the customer's name or address changes, it is necessary to track all instances of the data for future reference.

2. Maintain referential integrity for historical records

If a field value is updated, it is preferred to know which version of the data was in effect for a related instance. For example, if an invoice references a particular address and the address is later updated, when viewing that invoice, the system should be intelligent enough to automatically determine the invoice used the old address and not the new one without having to manually determine which address was in effect on the date of the invoice.

3. Enable join on a single field from foreign key to primary key

Where possible, a single primary key field is used in each table. All single primary keys are "identity" or "autonumber" fields automatically generated by the database – integer fields incremented with each new record. Using a single field primary key enables a direct relationship from a separate table via the primary key / foreign key method. This is more efficient when querying and joining tables than joining on multiple fields, such as an ID field and a version field.

Archival Types:

A. Archive individual updated fields, not the entire row
B. Archive entire row by making original row inactive and inserting new row
    i. Update necessary foreign key values with updated primary key
    ii. Copy existing data to replicate settings of original row

A. *Archive individual updated fields, not the entire row*

When user and company fields such as first name, email, company name, etc. are updated, only those fields that were updated are archived. Because it is likely that all of the user or company fields were not updated all at once, it is un-necessary to archive all

user or company fields if only a few fields were changed. Moreover, just because a user's last name changed – perhaps the person got married – it does not mean their first name changed. While it is possible that a person changed both their first and last names at the same time, it is not nearly as common.

B. *Archive entire row by making original row inactive and inserting new row*

Unlike user and company fields, an address change typically means more than one address field has changed, e.g, it is unlikely that only the street address changed. Often times, the address, city, state and zip code may change. Archiving only those fields within the address that changed make it difficult to view the full old address vs. the full new address. Piecing together the old address when each address field is stored in a separate row is not intuitive or even correct since a single address field does not adequately describe an address.

For these types of instances, such as updating an address, the entire row is archived and a new row is inserted. This makes it easy to view the full old address or the full new address. However, inserting a new row also creates a new record with a new primary key. The records are linked so that it is possible to determine that row #3 is a copy of row #2, which was itself a copy of row #1. This link is stored using 2 fields: ID_parent and ID_trend.

addressID – Primary key field for Address table

addressID_parent – addressID of address that was updated to create this new address. In other words, we "updated" the address record of addressID. But rather than updating the addressID record itself, we created a new addressID where the addressID_parent is the old addressID. So if addressID = 5 and we update that record, we insert a new record and thus create a new addressID = 6 where 6 was created from 5. The record represented by addressID 6 has a addressID_record = 5.

addressID_trend – ID of original address that was created. For instances where an address has been updated more than once, we want to know not only which addressID was the immediate parent, but also the original address. In our example above, if we updated addressID = 6 (which was itself created from addressID = 5) we create a new addressID = 7 that has addressID_parent = 6 and addressID_trend = 5, since that is the original addressID for this series of addresses. When a record is the first instance, the addressID_trend = addressID. So if addressID = 5 is the first address listed for a user, then addressID_trend = 5.

i. *Update necessary foreign key values with updated primary key*
If an address is updated, there are records that should maintain a link to the old address and other records that should reference the updated address. For instance, an invoice is associated with a particular billing address and that address is also the billing address for a credit card.

If the billing address is updated, the billing address for the invoice should not change since it was the address at the time of the invoice. In the payment records where the credit card was used to pay, the billing address used also should not change. But going forward, the credit card should reference the new address or the credit card charge may be rejected because of incorrect billing address. So the addressID field in the CreditCard table is updated with the new addressID, i.e., addressID is set to 6 where addressID = 5.

ii. *Copy existing data to replicate settings of original row*
In instances where a record is more than a row in a single table, all records must be replicated when the record is updated and a new row is inserted. For instance, a custom price not only has a row in the Price table, but rows in other price-related tables such as volume discounts and which companies the price applies to. When a price is updated, a new price is created and all rows in the other price-related tables are copied with the new priceID.

Disadvantages:
While the above methods meet the requirements and are as efficient as possible, there are a few disadvantages, such as:

- Each table may have many inactive rows, meaning the tables get larger and contain both active and inactive records, which may slow the database.
- It is sometimes necessary to update an existing foreign key in another table with the primary key value of the updated record.
- It is more confusing than simply backing up archived data in a separate archived table.
- For web services, it is slightly more confusing for how to update an existing address without having to first determine the current addressID. However, the web services interface does offer intelligent method of dealing with this issue that will work for the vast majority of customers with only a single active billing address at a time.

Alternatives:
Several alternatives were considered for archiving data:

I. An archive table where fields are archived individually
II. Separate archive table for each table
III. Updating the existing record and inserting an archived version of the old record
IV. Keeping primaryID the same and using different version numbers

I. An archive table where fields are archived individually
This method was used for the User and Company tables, but it is not appropriate for records where the full meaning of the record is not defined by an individual field. This is true with addresses, phone numbers and credit cards. An area code without the phone number is not useful. If the area code changes, it is likely the phone number changed as well. Storing an updated area code separately from the phone number makes it difficult to view the full phone number.

II. *Separate archive table for each table*
To archive an address, rather than creating a new address row in the Address table, we could have created an AddressLog table to store archived versions of the address. The benefit of this approach is that the addressID referenced in the invoice or credit card rows mentioned earlier would not necessarily have to changed when the address is updated.

However, this is not an ideal solution for 2 reasons: 1) It requires creating a redundant table with the same basic design for dozens of tables; 2) It does not maintain referential integrity, such as viewing the billing address associated with a particular invoice at that time instead of the current billing address. By losing the reference, it is not possible to determine which billing address was in effect at the time except by manually comparing the invoice date and the valid dates of the address.

III. *Updating the existing record and inserting an archived version of the old record*
If an address is updated, rather than inserting a new address with the updated information, one option would be to insert a "new" address with the old address information and update the "current" address with the new information. This means the addressID that is referenced by a credit card does not change. However, this still loses the referential integrity of linking the address to an invoice or an earlier charge. To maintain this relationship, the addressID in the old records must be updated to the "new" record. This is worse than only changing the addressID in those records necessary for future use, such as the address for the credit card.

IV. Keeping primaryID the same and using different version numbers
In the Address table, the primary key addressID is an "identity" or "autonumber" field, which means the database automatically generates the primary key value by incrementing the currently maximum ID. This is significantly more efficient and quicker than doing this manually via code. Assigning the primary ID manually would significantly increase system overhead and could cause database lock conflicts.

While using a version number is useful to track changes, it would be more difficult to reference the current primary ID in queries via a join since the join would not be addressID to addressID, but rather addressID to addressID / addressVersion. This type of join is not as efficient and violates the relational concept.

Also, if selected multiple "current" address, such as the current address of multiple users, if the versions were incremented in the order in which they were created, it would not be possible to efficiently select the "current" address based on the version. A work-around for this is as simple as a bit field that indicates whether a record is the current version or perhaps using a value of 0 for the current version. But that either adds a third field required for a join operation or requires logic to update the version number from 0 to the proper number when updated.

This also does not provide a method to reference the address used by an invoice in a single field. Instead, it requires multiple fields, meaning each foreign key field would need to change from a single field to a dual foreign key.

companyID vs. companyID_author

Definitions:
"client" – The company using the billing software to bill their own customers
"customer" – A customer being billed via the billing software

In some tables, it is necessary to store the ID of multiple companies and/or users. For instance, the Invoice table contains both companyID field and companyID_author fields where companyID is the customer that the invoice is for, and companyID_author is the client who is invoicing the customer. Similarly, the Invoice table contains a userID field which is the contact person at the customer for that invoice and a userID_author field which is the client employee that created or last updated the invoice.

In "setup" tables which store client preferences, such as a product created by the client, the database stores the "companyID" of the company the product belongs to and the "userID" of the user who created or last updated the product. Because there is no companyID or userID necessary to refer to the customer, the companyID and userID fields refer to the client.

In essence, in some tables companyID refers to the client and in other tables it refers to the customer; and userID refers to the customer or the client employee that created that record. This is admittedly confusing, but the generally concept is that if there is only a single companyID or userID necessary in the table, then it is silly to append "_author" to it.

primaryTargetID & targetID

In many tables, the record may refer to one of many possible objects, such as a user, company, group, affiliate, cobrand, etc. For instance, the Note table stores notes for all objects in the system. You can record a note about a particular user, company, invoice, etc. With each note, we record which object the note refers to. However, since the Note may refer to more than a dozen objects, we need a method to determine the type of object it refers to (i.e., a user or company) and the primary ID of that object (companyID = 5). The alternative is to create a separate Note table for each object type, which could mean over a dozen Note tables.

primaryTargetID – An integer field that corresponds to the object type (e.g,. "user").
targetID – An integer field that corresponds to the primary ID of that object (e.g., userID = 5)

The primaryTargetID value for each object (a.k.a., a primary key from a table) is linked via the PrimaryTarget table and is also stored in a function in the code for quick code-level access. When a new table is created, the table information is added as a row in the PrimaryTarget table via an admin screen. When the new table is added, the code-level function is updated with the new object information as well. Other than the system configuration information, this is the only file that may vary across customer

installations, such as if they add their own tables or add new tables in a different order. While this is annoying, as long as the new table/object is added via the admin screen, all is well.

List of affected tables:

User, Company

Address, CreditCard, Phone, Bank
InvoiceLineItem, Subscription
Price, Commission

CustomField,Note,Task,StatusHistory
InvoiceLineItemParameter,
ProductBundle,ProductLanguage
SalespersonCustomer

avFieldArchive
User, Company