Challenge 1: Comma splices and run-on sentences

Before

Outbound processing
XXX Sends request messages out in the form of pacs.008 for both Eligibility and Payment requests, in either situation, the message format is fairly similar aside from a single SvcLvl.Prtry field that either has "VisaElig" present or not. Once the pacs.008 message enters XXX, the IPF session can distinguish the difference and select the appropriate session and in the case of a present VisaElig Prtry field, the session selected will be "XXX General Purpose", however if the field is not present then it will pass through the ISO 20022 Processing session.

The VISA B2B Endpoint also lies on the XXX level, which can interpret the difference between eligibility requests and payments. In each case, the endpoint will allocate specific fields for the request to be formatted into JSON for sending to the VDP, it is important to note the difference, as the VDP will format the message as either an eligibility request or a payment request based off the fields present.

Similarly, the Visa B2B Endpoint can process the message received from VDP and transform the JSON into a pacs.002 response, this response will be matched with the request, either eligibility or payment, and sent back through to XXX. Happy path and error codes are both acceptable, with the reason code and reason description being passed through into the pacs.002 and Further exception processing is not currently supported but expected to be added onto the Endpoint functionality.

Inbound processing
Visa is in charge of sending the inbound payment to us, XXX needs to have a method in place to listen to this, and carry the transaction across to XXX, XXX achieves this using Batch Processing with Fileload.

Providing Visa with the location of the file drop location on our system, they can drop files for processing in our Visa B2B Endpoint, the endpoint will watch the file drop location and can process the file as a payment once it has been dropped without any further interaction. XXX does some validation to determine whether or not the structure of the file is valid. If it is valid, XXX will immediately send a response back to VISA via the file drop location output batch with an acknowledgement pacs.002 message, if the message was invalid, it will send a non-acknowledgement pacs.002 message, no further processing will take place. If the message was valid however, after sending the acknowledgement, XXX will then push the payment pacs.008 structured file to XXX, and wait for the final response.

At this point, XXX will not expect the response immediately, it could take up to a day to process, however once it does XXX will be able to match up the TRN based off the initial request and match the response accordingly. The Final Response is mapped by XXX into a pacs.002 which is dropped into the output batch for consumption by Visa.

After

Outbound processing

XXX sends request messages in pacs.008 format for both Eligibility and Payment requests. In either situation, the message format is similar, other than the SvcLvl.Prtry field, which has "VisaElig" present or not. After the pacs.008 message enters XXX, the IPF session can distinguish the difference and select the appropriate session. If the VisaElig Prtry field is present, the session selected will be "IPF General Purpose." However, if the field is not present, it will pass through the ISO 20022 Processing session.

The VISA B2B Endpoint is also on the XXX level, which interprets the difference between eligibility requests and payments. In each case, the endpoint will allocate specific fields to format the request into JSON to send to the VDP. It is important to note the difference because the VDP will format the message as an eligibility request or a payment request based off the fields present.

Similarly, the Visa B2B Endpoint can process the message received from VDP and transform the JSON into a pacs.002 response. The eligibility or payment request matches the response and sends it back through to XXX. Happy path and error codes are both acceptable, passing the reason code and reason description into the pacs.002. YYY does not support further exception processing but we expect to add it to the Endpoint functionality.

Inbound processing

Visa sends the inbound payment to YYY. XXX uses Batch Processing with Fileload to listen and carry the transaction to XXX.

YYY must provide Visa with the file drop location on their system so Visa can drop files for processing in YYY's Visa B2B Endpoint. The endpoint will watch the file drop location and can process the file as a payment after it dropped without further interaction. XXX does some validation to determine whether the structure of the file is valid. If the file structure is valid, XXX will immediately send a response to VISA through the file drop location output batch with an acknowledgement pacs.002 message. If the message was invalid, it will send a non-acknowledgement pacs.002 message, and processing ends. If the message was valid, after sending the acknowledgement, XXX will push the payment pacs.008 structured file to XXX and wait for the final response.

XXX does not expect an immediate response; it might take up to a day to process. However, after it does, XXX can match the TRN based off the initial request and match the response accordingly. XXX maps the Final Response into a pacs.002 that is dropped into the output batch for consumption by Visa.

Challenge 2: Sentence fragments and dangling modifiers

Before

From YYY:
Transparent Data Encryption (TDE) introduced by Microsoft starting with SQL Server 2008. It provides the possibility to perform full database-level encryption. Until now there were only two options provided by Microsoft for encryption. Cell-level encryption as in SQL Server 2005 or the file-level encryption options provided by the Windows operating system. TDE is the optimal choice for bulk encryption to meet regulatory compliance or corporate data security standards.

Transparent data encryption (TDE) performs real-time I/O encryption and decryption of the data and log files. The encryption uses a database encryption key (DEK). Which is stored in the database boot record for availability during recovery. The DEK is a symmetric key secured by using a certificate stored in the master database of the server or an asymmetric key protected by an EKM module. TDE protects data "at rest," providing the ability to comply with many laws, regulations, and guidelines established in various industries. By using AES and 3DES encryption algorithms without changing existing applications, this enables software developers to encrypt data.

The encryption is performed at the page level. Before being written to disk, they are encrypted. When read into memory, they are decrypted.

The fundamental unit of data storage in SQL Server is the page. The disk space allocated to a data file (.mdf or .ndf) in a database is logically divided into pages numbered contiguously from 0 to n. Disk I/O operations are performed at the page level. That is, SQL Server reads or writes whole data pages.

TDE does not increase the size of the encrypted database.

From ZZZ:
Version 3 of the Generate Funding Token API allows a client to capture funding details on their website and have a funding token generated from the funding details. To represent the funding in a payment API request and/or a recurring payment schedule API request, the token can be used again.

Along with the funding details, the client will pass the wallet ID (billing account number or enrollment profile ID) for which the token will be associated. The wallet ID identifies the owner of the token. If the token is being used for a one-time, on-demand payment and the consumer does not have an enrollment profile. The billing account will be passed as the wallet ID. If the consumer has an enrollment profile, the profile ID will be passed as the wallet ID.

Since the API validates the funding information, it is not necessary to perform any validations on the client's UI application.

To determine if the token should be flagged for multi-use (consumer opted in to retain the funding for use with future payments) or single use (consumer opted to not retain the funding for use with future payments, the API also allows the biller to designate.

After

From YYY:

Microsoft introduced Transparent Data Encryption (TDE) in SQL Server 2008 to perform full database-level encryption. Previously, Microsoft only provided cell-level encryption in SQL Server 2005, and only file-level encryption in Windows. Use TDE when regulatory compliance or corporate data security standards require bulk encryption.

Transparent Data Encryption (TDE) encrypts and decrypts data at rest and log files in real-time I/O. The encryption uses a database encryption key (DEK). The database boot record stores the key for availability during recovery. The DEK is a symmetric key. It is secured by a certificate that the server's master database stores or by an asymmetric key that an EKM module protects. Use TDE to comply with laws, regulations, and guidelines established in various industries. Software developers can encrypt data with AES and 3DES encryption algorithms without changing existing applications.

The encryption is performed at the page level. Pages are encrypted before they are written to disk and are decrypted when read into memory.

The page is the fundamental unit of data storage in SQL Server. The disk space allocated to an .mdf or .ndf file in a database is logically divided into pages numbered contiguously from 0 to n. SQL Server reads or writes whole data pages.

TDE does not increase the size of the encrypted database.

From ZZZ:
Clients can use the Generate Funding Token API version 3 on their website to capture funding details and then generate a funding token. The token can be reused as funding for a payment API request and a recurring payment schedule API request.

The client passes funding details and the wallet ID (billing account number or enrollment profile ID) for which the token will be associated. The wallet ID identifies the token's owner. If the token will be used as a one-time, on-demand payment and the consumer does not have an enrollment profile, the API uses the billing account number as the wallet ID. If the consumer has an enrollment profile, the API passes the profile ID as the wallet ID.

The API validates the funding information, so no validations are needed on the client's application.

The API also allows the biller to designate whether the token must be flagged for multi-use or single use. Multi-use means that the consumer opted in to retain the funding for use with future payments. Single-use means the consumer opted not to retain the funding for use with future payments.

Challenge 3: Concise writing

Before

As a result of the way it performs Multi-Valued Concurrency Control (MVCC), PostgreSQL requires certain maintenance activities that is not needed in other databases, one of which is called “XID Wraparound” Maintenance. This document describes appropriate procedures (aligned with “vacuum” maintenance procedures, another distinctively PostgreSQL maintenance activity).

What XIDs (transaction IDs) are is simple (how they work is described later in this document). There is a global transaction counter variable called “XID” (transaction ID) that is used as the basis for comparing transaction chronology. Its value is incremented for every transaction that makes any database changes (and PostgreSQL stores its value in database rows altered by the transaction, as part of the same chronology-setting mechanism). PostgreSQL depends on the ability to compare a new XID value to XID values that were previously stored.

The challenge is that the XID variable was set at 32-bits in size when PostgreSQL was invented (when computers didn’t support 64-bit data-types), and the value of a 32-bit quantity inevitably “wraps” when its incremented from its maximum value (~4 Billion) to the value zero, then it continues to increment. This makes the simple comparison of XID values challenging, and the wraparound of XID values is inevitable when present in a long-running database.

Without compensating mechanisms, two things are compromised:

- The use of XID as a basis for establishing transaction chronology
- The ability of the database to “house” datasets that have been created by an arbitrary number of independent database transactions (each with its own unique XID value).

PostgreSQL copes with the situation successfully, but the mechanisms require regular “XID Wraparound Maintenance” activities, which must be applied on a regularly recurring basis. The consequence of not doing so will be that as the ~4 Billion transaction limit is approached, PostgreSQL will recognize impending disaster, and will flip the database to read-only mode. To get it back online will require an offline, single-user, standalone vacuum operation that could consume many hours or days of downtime (depending on the size of the database). This consequence is to be avoided at all costs for ACI’s mission-critical applications.

This document describes what configurations and tools to use to achieve “XID Wraparound Maintenance”, and how to configure them.

XID lifecycle

Since insert, update, and delete operations all consume XID values, XIDs evolve over time based on the update activity the table endures.

Each row in a table has an XMIN and an XMAX column added by PostgreSQL, and it is typically hidden from view. When a new row is added to a table, that row’s XMIN column is populated with the XID of the transaction that added it; at this time XMAX column is null. If the row is never changed again that XMIN value will age, in relation to new XID values, until it reaches one of the thresholds mentioned in this document and gets frozen or causes some level of elevated vacuum processing. If the row is subsequently updated, the XMAX column will be updated to the XID of the transaction performing the update, and a new row with the updated values will be added to the table and its XMIN will be set to the XID of the update transaction and its XMAX will be null. At this point the table contains two rows, as soon as the oldest transaction in the system has an XID newer than the value of XMAX in the original row, that row is considered to be dead. No transactions can read that row and it can be removed by autovacuum or vacuum; until the row is removed it is considered to be bloat. Deletes also set the XMAX to the value of the XID of the transaction doing the delete, and the row also stays on the table until it is older than the oldest transaction at which time it becomes bloat until vacuum removes it from the table.

Read-only and insert-only tables will both experience XID aging until the rows are either frozen or deleted. Tables with a high degree of update or delete activity will have less issues with XID aging and more issues with bloat management. The solution to both bloat and XID aging is to autovacuum and/or vacuum, and we will discuss the settings based on the table workload below.

After

Postgres's Multi-Valued Concurrency Control (MVCC) requires "XID Wraparound" and vacuum maintenance.

An XID (transaction ID) value increments when a transaction makes a change to the database. Postgres stores a new version of the transaction's row with that XID value. PostgreSQL compares the new XID value to previous XID values to determine transaction chronology.

Comparing XIDs can be difficult because long-running databases typically have wraparound XID values. The XID is a 32-bit value, so the database can execute over four billion transactions, until the value wraps to zero.

This compromises the following:

- Using XID to establish transaction chronology
- Storing datasets created by many database transactions, each with a unique XID value

PostgreSQL can use MVCC successfully if you perform recurring "XID Wraparound Maintenance." You must do this for ACI's mission critical applications. If you do not, Postgres will switch to read-only mode when it approaches the ~4 billion transaction limit. It can take hours to days of downtime (depending on the database) to get the database back online using an offline, single-user, standalone vacuum operation.

This document describes how to configure and perform "XID Wraparound Maintenance."

XID lifecycle

Insert, update, and delete operations cause XID values to evolve.

PostgreSQL adds a hidden XMIN and XMAX column to each row in the table. When a transaction inserts a row, it saves the XID to the XMIN column; the XMAX column is NULL. If the row never updates again, the XMIN value ages until it:

- reaches the threshold mentioned in this document
- freezes
- causes elevated vacuum processing

However, if a transaction updates the row, it saves the XID to the XMAX column. A new row is inserted to the table and its XMIN value is set to the XID of the update transaction; the XMAX column is null. Now, the table has two rows. When the oldest row has an XID that is newer than the value of XMAX in the original row, the row is dead and considered bloat. No transactions can read that row. Auto vacuum or vacuum must remove the row.

Note: Delete operations set the XMAX column to the delete transaction's XID. The row remains in the table until it is older than the oldest transaction. Then, it becomes bloat until vacuum removes it from the table.
XID-aging occurs for both read-only and insert-only tables until the rows are either frozen or deleted. Tables with a lot of update or delete activity will have less XID aging, but more bloat. Use auto vacuum and vacuum to resolve both bloat and XID aging. See the following table workload for information about the settings.