Change Log

Author	Date	Changes
Paul	3/2/2012	Added a changelog. ☺
Paul	3/2/2012	Per discussion with Michael, cleaned up the KSSA_CREDIT_PERMISSION table. ALLOWABLE_DEBIT_TYPE_ID. Cleaned up KSSA_ALLOCATION to create an autonumber primary key, and made ACCOUNT_ID -> ACCOUNT FLAG ->FLAG_TYPE, KSSA_INFORMATION.FLAG_ID bcame FLAG_TYPE_ID_FK

Data Models for the Transaction and Associated Classes

General Notes

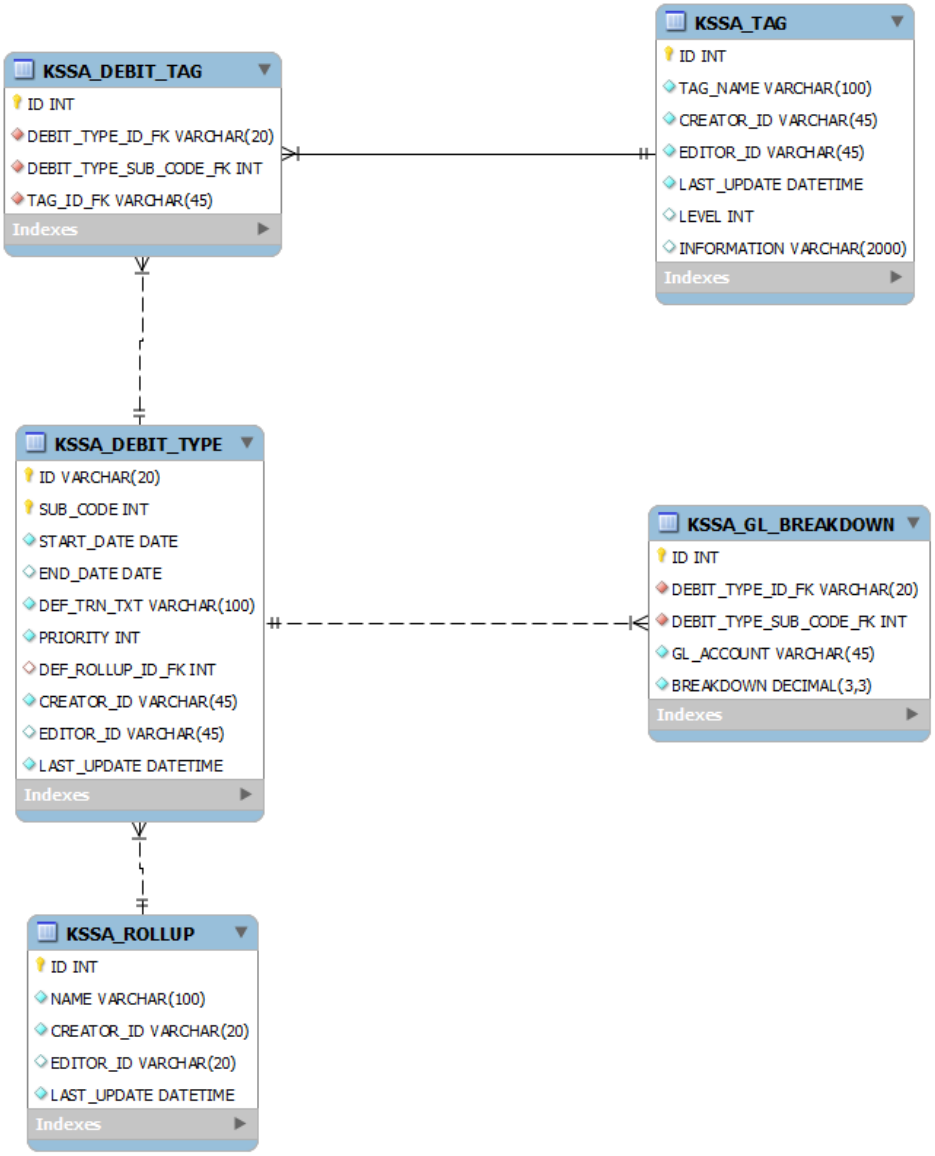
- All tables are preceded with KSSA (Kuali Student – Student Accounts) to prevent table name clashes. This follows the rules established by Rice.
- Appropriate contractions are used as suggested on the Rice and KS wiki, including but not limited to TRN-Transaction and ID- Identifier, AMNT- amount. Foreign keys, where they apply to our tables, are tagged with _FK. A copy of this document is in the dropbox.
- These data models have been designed to support the permanence layer of the Transaction class, its children and its associated classes. Only the permanence layer will access this data structure directly.

There are a number of reused data structures that are not repeated in the document. It is assumed that they are understood.

CREATOR_ID and EDITOR_ID are the identifiers for the entity who creates, and if appropriate, subsequently edits a record. LAST_UPDATE is the date stamp for the last alteration to the structure. LEVEL refers to a general access level that is defined by the institution. It is stored as a plain INT, and a user must have a LEVEL equal to or greater than the LEVEL of the referenced information to be able to view it. Few levels are expected in reality (as roles exist to give much more granular controls) but an example might be 0 for Student (this is expected) 1 for external staff (departments, etc) 2 for internal bursar staff (and default for memos, etc.) and 3 for high-level employees in the bursar's office.

Debit Type Data Model.

This model underlies the DebitType object, which is an associated class with the Debit class. Every Debit HAS_A DebitType. Note that DEBIT_TYPE and its relations are used to create a permanent store for the class DebitType. Although a similar subclass, CreditType exists, they are stored in different table structures.



The debit type table is the master list of types of debits that can be entered on the system. Every debit type (ID) is valid for a certain period of time. START_DATE has to exist, and will default to the date that the type of debit was created. END_DATE is optional. If null, it is assumed that the debit type is valid currently, until it is changed.

With the debit type identified, plus the date of the transaction, the SUBCODE can be determined. The debit type (ID) with the SUBCODE allows us to look up the details of the debit. We can also look up the

default statement text (which can be overridden in the transaction) and the priority of the transaction. Higher priority debits will be paid before lower priority debits. Equal priority debits are paid off FIFO.

ID and SUBCODE can be referenced against the GL_BREAKDOWN table, which will render a list of GL_ACCOUNTS and the BREAKDOWN against that account. The breakdown is a percentage of the total transaction that will be allocated to the general ledger account. Where there is a unary mapping of debit type to general ledger account, there will be only one GL_ACCOUNT and BREAKDOWN will be 100.00%

Where schools divide single transactions over multiple general ledger accounts, these will be listed in this table, with BREAKDOWN amounts to spread the payment. In this case, percentages can be allocated, and there will be one "bucket" account, which receives the remainder of the funds. This prevents problems with fractional currency being unallocated or over allocated or under allocated. For example, a transaction that divides into two general ledger accounts as a 50/50 split would be defined as

ACCOUNT 1 – 50%

ACCOUNT 2 – 0 (Bucket account)

Therefore a $100 transaction would divide as:

50% of $100 = $50. ACCOUNT 1 received $50

ACCOUNT 2 gets the remainder, therefore $100-$50 = $50 to account 2.

For a $99.99 transaction

50% of $99.99 = $49.995. With a rounding up, ACCOUNT 1 would be credited $50.00

ACCOUNT 2 would be credited with $99.99-$50 = $49.99

Comment [i1]: This is a codified business rule, but I am not aware that we have heard of any other process of dividing up payments. I believe this, at the least, supports the needs of UMD.

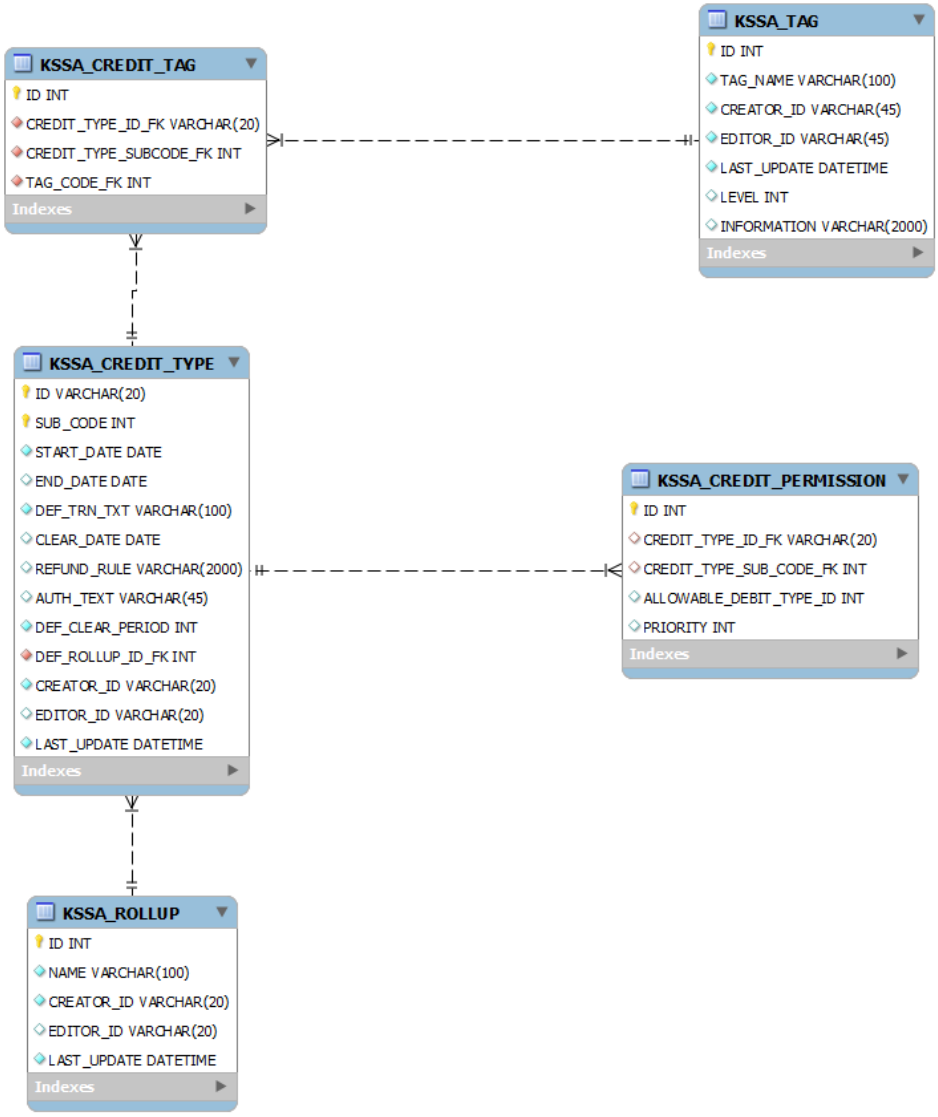
Debit types can also be tagged with multiple attributes from a user defined list (found in TAG). Most often, transactions are categorized using a preset numbering system, (for example, TUT*** are tuition codes, etc.) However, there are times when the transaction codes do not permit flexible categorization for certain reporting purposes. Tags are an optional way to allow control over categories of transactions. Tags have an ID, and a NAME. They also have a LEVEL which defines who can see the tag. If tags are visible (TBD) then a user would need this level to be able to view them. It is assumed that a student is level 0.

INFORMATION Is a plain text description of the tag to give the user more information as to why the tag is used.

Similarity, each debit can be referred to a default rollup, which is a reference to the KSSA_ROLLUP table. This is similar to the KSSA_TAG table.

Credit Type Data Model

This model underlies the CreditType class, which is an associated class with the Credit class. Every Credit object HAS_A CreditType.



Credit types are user configurable types that can be applied to an account. They are able to change over time as with DebitType however, it is expected that they will be more stable, and far less numerous than DebitType. Examples of credit types would be cash, credit card, check, financial aid, etc. There may be a need to handle different types of cash payments differently, so it is envisaged that many different credit types might be created.

Credits have a default statement text, which may be overridden in the Transaction class. They also have a CLEAR_DATE which is a time period, specified in days, after which the payment is considered to be cleared. For example, an institution may implement a 10-day hold on check payments.

The refund rule allows an institution to specify how a transaction may be refunded. Note that if this is only applicable if the IsRefundable flag is set in the Credit object.

The refund rule allows for the following scenarios:

- A refund in cash or equivalent may be issued on this amount after the clearing period (in the case of cash, the clearing period would be 0) (example, cash, checks)
- A refund to the original source may be made after the clearing period. (example, credit cards)
- A refund to the original source may be made for a specified period of time, after which a refund in cash or equivalent may be made. (for example, credit cards.)

Comment [i2]: Exactly how this is to be encoded is not yet decided. I am still working out if we have all the major use cases covered.

Note that the general REFUND_RULE for a CREDIT_TYPE can be overridden in the TRANSACTION table. If TRANSACTION.REFUND_RULE = null, then CREDIT_TYPE.REFUND_RULE is used. Otherwise TRANSACTION.REFUND_RULE Is followed. TRANSACTION.REFUND_RULE permits the same rules as CREDIT_TYPE.REFUND_RULE, as well as also permitting a refund to another KSA account.

AUTH_TXT is a friendly text field to assist the user when processing payments. It stipulates the expected reference that the payment will be. In the case of a credit card, for example, the authorization code from the credit card company might be the reference stored for the transaction. In the case of a check, the bank information and the check number might be stored. This is institution and payment specific.

DEF_CLEAR_PERIOD is the number of days after the entering of a payment, it is considered ‘cleared’ (and therefore refundable). This period is used in logic relating to refunds.

DEF_ROLLUP_ID_FK is a foreign key in the KSSA_ROLLUP table.

The credit type (ID+SUBCODE) are used to link to both the CREDIT_PERMISSION and the CREDIT_TAG tables.

CREDIT_TAG works in much the same way as DEBIT_TAG allowing for a very flexible way of categorizing transactions on the system that does not rely on pre-formatted transaction codes. FRIENDLY_NAME provides a way of giving the credit a name that is more readily understood by a human.

CREDIT_PERMISSION gives a list of allowable debits that can be paid by this type of transaction. This can be done either by listing individual codes, or by using masked codes. For a payment that can be applied to any transaction, the general wildcard would be here.

Priority states the priority of a credit to pay off a certain group of transactions. This is only used if the system needs to break a tie between Debits of the same priority. If two debits have the same priority and the credit is allowed to pay them both, it will pay off the higher priority codes first, before applying the remainder to the remaining codes.

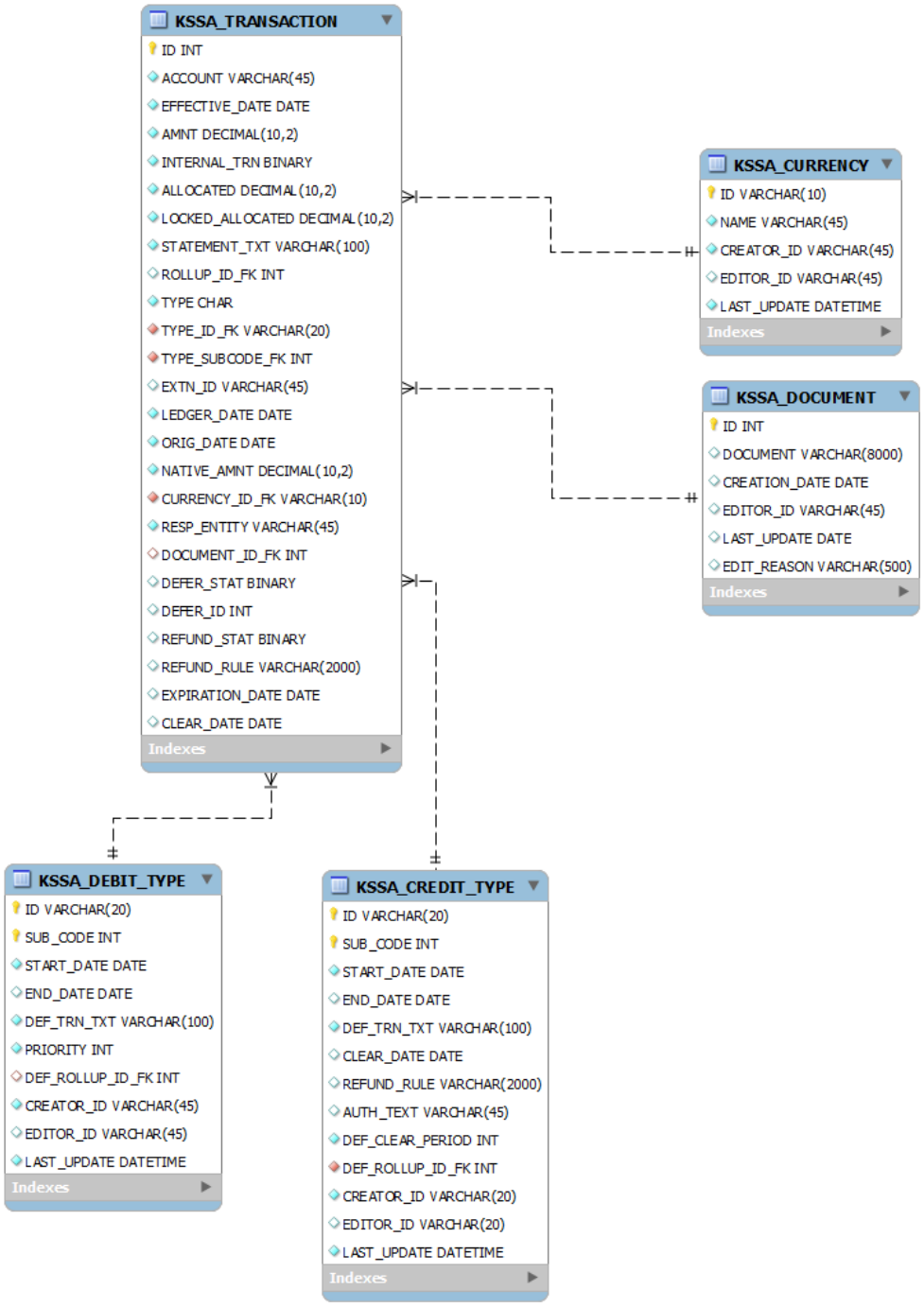
In the KSSA_ROLLUP table ID is the identifier, and NAME Is a simple friendly name that will show when a group of transactions are rolled up. So for example, there might be 15 financial aid transactions, which are rolled up under the name "Financial Aid". Drilling in to the rollup would display the group of transactions.

Transaction Data Model

The transaction class is summarized in two tables. TRANSACTION is identified by ID and lists all the headline information about any transaction. Every Transaction can be either a Credit or a Debit. A Credit can be a Deferment or a Payment, a Debit is a Charge. A debit HAS_A DEBIT_TYPE, a Credit HAS_A CREDIT_TYPE. For clarity, CREDIT_TYPE and DEBIT_TYPE are defined earlier.

Comment [i3]: I envisage masking to work as follows.

- *For any character or number of characters.
- ? For any single character.
- @ for any letter
- # for any number


KSSA_TRANSACTION
ID INT
ACCOUNT VARCHAR(45)
EFFECTIVE_DATE DATE
AMNT DECIMAL(10,2)
INTERNAL_TRN BINARY
ALLOCATED DECIMAL(10,2)
LOCKED_ALLOCATED DECIMAL(10,2)
STATEMENT_TXT VARCHAR(100)
ROLLUP_ID_FK INT
TYPE CHAR
TYPE_ID_FK VARCHAR(20)
TYPE_SUBCODE_FK INT
EXTN_ID VARCHAR(45)
LEDGER_DATE DATE
ORIG_DATE DATE
NATIVE_AMNT DECIMAL(10,2)
CURRENCY_ID_FK VARCHAR(10)
RESP_ENTITY VARCHAR(45)
DOCUMENT_ID_FK INT
DEFER_STAT BINARY
DEFER_ID INT
REFUND_STAT BINARY
REFUND_RULE VARCHAR(2000)
EXPIRATION_DATE DATE
CLEAR_DATE DATE
Indexes
KSSA_CURRENCY
ID VARCHAR(10)
NAME VARCHAR(45)
CREATOR_ID VARCHAR(45)
EDITOR_ID VARCHAR(45)
LAST_UPDATE DATETIME
Indexes
KSSA_DOCUMENT
ID INT
DOCUMENT VARCHAR(8000)
CREATION_DATE DATE
EDITOR_ID VARCHAR(45)
LAST_UPDATE DATE
EDIT_REASON VARCHAR(500)
Indexes
KSSA_DEBIT_TYPE
ID VARCHAR(20)
SUB_CODE INT
START_DATE DATE
END_DATE DATE
DEF_TRN_TXT VARCHAR(100)
PRIORITY INT
DEF_ROLLUP_ID_FK INT
CREATOR_ID VARCHAR(45)
EDITOR_ID VARCHAR(45)
LAST_UPDATE DATETIME
Indexes
KSSA_CREDIT_TYPE
ID VARCHAR(20)
SUB_CODE INT
START_DATE DATE
END_DATE DATE
DEF_TRN_TXT VARCHAR(100)
CLEAR_DATE DATE
REFUND_RULE VARCHAR(2000)
AUTH_TEXT VARCHAR(45)
DEF_CLEAR_PERIOD INT
DEF_ROLLUP_ID_FK INT
CREATOR_ID VARCHAR(20)
EDITOR_ID VARCHAR(20)
LAST_UPDATE DATETIME
Indexes

ACCOUNT references the account to which the transaction is applied.

EFFECTIVE_DATE is the date that the transaction is considered “current” on the account. This is the date around which all processing is based.

AMNT is the value of the transaction in the system specified currency. Note that although the decimalization is overkill for US dollars, the wider field is used to accommodate other currencies.

INTERNAL_TRN is a Boolean that dictates whether a transaction is considered “internal” or not. Internal transactions are generally not presented to customers.

ALLOCATED is the amount (of currency) that is allocated. For a payment, it is the amount of the payment that is allocated to a charge. For a charge, it is the amount of the charge that has been paid.

LOCKED_ALLOCATED is the same as ALLOCATED, except this payment allocation cannot be de-allocated by the automatic payment allocation module. A CSR might choose to lock certain payments and charges together, or a school might choose to freeze all allocations from a previous period to stop the system from de-allocating earlier allocations.

STATEMENT_TXT is the ‘friendly text’ for a transaction that is displayed to explain the purpose of the transaction.

ROLLUP_ID_FK is a field used to group similar transactions in to a rollup. For example, if a number of add/drops occur within a period, all of the charges and refunds might be given the ROLLUP_ID that refers to ”Tuition”. Then, on the initial view of the statement, the word “Tuition” would appear with a net of the values. Further inspection would allow the full list of transactions that make up the rollup appear.

TYPE references the type of transaction; acceptable values are TCP for TRANSACTION->CREDIT->PAYMENT, and TCD for TRANSACTION->CREDIT->DEFERMENT and TDC for TRANSACTION->DEBIT->CHARGE.

TYPE_ID_FK links to the DEBIT_TYPE or CREDIT_TYPE definitions, depending on the value of TYPE.

TYPE_SUBCODE_FK links to the subcode of the DEBIT_TYPE or CREDIT_TYPE definitions, depending on the value of TYPE.

EXTN_ID is the external transaction identifier. For external systems that generate a transaction identifier will populate this with the foreign identifier. If the transaction comes via batch, then the batch id will be here. For payments, the expected authorization code will be inserted here. For example, a credit card payment might use the authorization code as the EXTN_ID.

LEDGER_DATE is the date that the transaction was entered in to the ledger within the KSA-RM system.

ORIG_DATE records when the transaction was generated. This is most useful for transactions occurring in legacy systems that upload their transactions in batches.

NATIVE_AMNT is the amount of the transaction in the native currency of the transaction. Where the native currency and the system currency are the same, NATIVE_AMNT and AMOUNT will be the same.

CURRENCY_ID_FK is the currency identifier for the transaction. In most cases, this will equal the system currency. It is stored as ISO4217 currency codes. CURRENCY_ID links to the CURRENCY table if a friendly version of the currency is needed (E.X. AUD can be referenced to “Australian Dollar”, or appropriate string as required by the language of the country the system is established in.)

RESP_ENTITY is the identifier for the entity who created the transaction.

Comment [i4]: A secondary entity is forming in my mind (but not yet on paper) for a BILLING_AUTHORITY identifier that would provide details of the subunit that billed the account. TBD. That might feed nicely into the external department access.

DOCUMENT_ID_FK is the identifier for an XML document that holds information regarding the transaction. This could be a number of different elements. For example, a bookstore transaction could send the names of the books and their prices in the document. This document is for information presentment and is not intended to be used as part of the accounting process.

DEFER_STAT is a Boolean that is only applicable to debit transactions. It will be null for credits. If True, then this debit has been deferred. The identifier for the deferment will be stored in DEFER_ID.

DEFER_ID is the reciprocating transaction reference if a transaction is deferred. In the case of a deferment transaction, this will point to the transaction which it defers. In the case of a deferred transaction, this will point at the deferment transaction.

REFUND_STAT is a Boolean that is only applicable to credits. It answers the question “is this credit refundable?” If false, the transaction cannot be refunded if it causes a credit balance. This would be the case for types of deposits.

REFUND_RULE defines the refund processing rules. If this is blank, then it defaults to the refund rule defined in CREDIT_TYPE. However, if this value is set, then it overrides the rules in CREDIT_TYPE and takes priority. This rule follows the same format as CREDIT_TYPE refund rule, except it also permits the option of refunding to another KSA account. This override refund rule is required to permit overpayment refunds to other sources. Use cases would be ParentPLUS loans, as well as sponsorship overpayments.

EXPIRATION_DATE is only applicable to DEFERMENT transactions, otherwise it is set to null. On this date, the deferment will be expired.

CLEAR_DATE is only applicable to PAYMENT transactions, otherwise it is set to null. The clear date is used in refund processing as the date as which a transaction is considered to be ‘like cash’. For example, when processing a check payment, an institution may choose not to refund against a check until 10 days have passed.

Currency

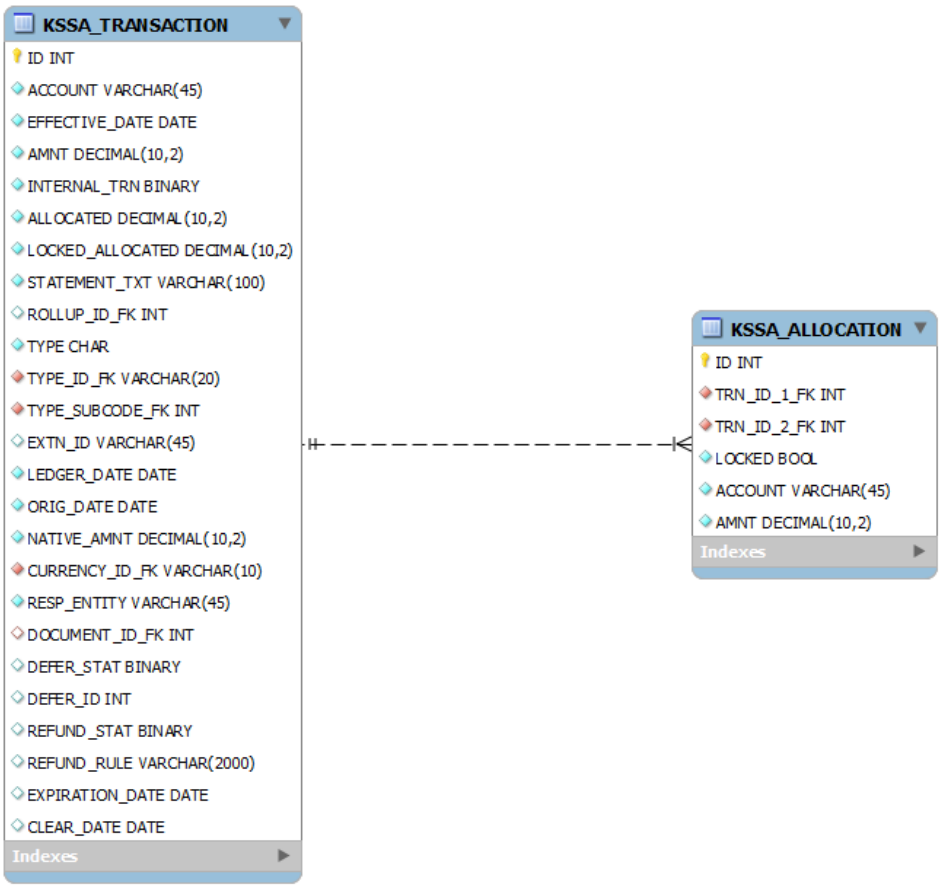
The CURRENCY table is most often used with the TRANSACTION . It simply maps a currency identifier (ID) (in IOS 4217 format) to the friendly name (FRIENDLY_NAME) for the currency. For example, “GBP”, “British Pound”.

Documents

A document is a freeform XML document that is used to describe a transaction. It contains details about the transaction that are useful to the user. An example might be if a student purchases books in the bookstore, the document could contain which books were purchased. This information is not intended to be 'interpreted' by the system, rather provide customer service information to the student.

Each document has an ID, which is associated with the ID in the TRANSACTION table. (TRANSACTION.DOCUMENT_ID_FK) A transaction can have only one document associated with it DOCUMENT Is the XML document. CREATION_DATE shows when the document was added to the transaction (and will almost always be created at the same time as the transaction.) EDITOR_ID identifies who, if anyone, has manually edited the document. LAST_UPDATE shows when the document was last altered. This would most usually be the same as CREATION_DATE. EDIT_REASON is a freeform text string, to explain why the document was altered by the user.

Allocation Data Model



KSSA_ALLOCATION derives its primary key from the two transaction identifiers that make it up plus their locked status. While a transaction might have multiple applications (one payment may be allocated to multiple charges, one charge may be paid off by multiple payments) no group of transactions may be allocated more than once. The only exception to this is a locked allocation might be made on a partial payment, and then a further unlocked allocation might occur in the future. For this reason, the LOCKED_STATUS is part of the PK.

The KSSA_ALLOCATION table is the table of record for allocation information. The TRANSACTION data structure contains grouped information on allocations, but only the ALLOCATION table contains the actual reference of payments and charges. If for whatever reason, the two were to become unsynchronized, the ALLOCATION table would be the table to verify. ACCOUNT_ID references the

account to which the allocations belong. Although ACCOUNT_ID can be derived from TRN_ID_1/2, it is included to assist in speeding lookups for the data.

TRN_ID_1_FK is the identifier of the first transaction in the allocation. Although payment application may default to a certain pattern, there is no requirement that the first column be a charge or a payment, just a valid transaction. The payment application routine will be responsible for verifying that the payment application is appropriate (not allocating a charge to a charge, or a payment to a payment, etc.)

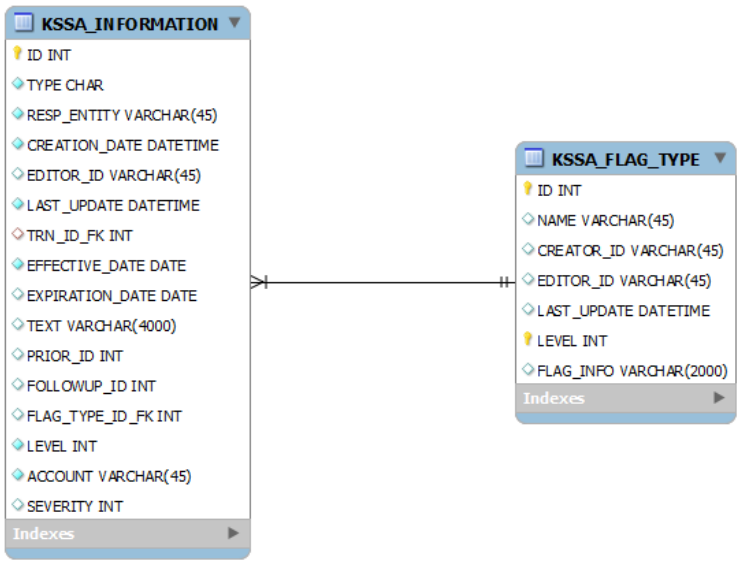
TRN_ID_2_FK is the identifier for the second transaction.

LOCKED is a Boolean value. Locked amounts are not reallocated by the payment application routines. Unlocked transactions may be reallocated at will by payment application. It is primary, as there may be both a locked and an unlocked allocation between the same two transactions.

ACCOUNT_ID_FK is an implied field, but will permit easier searches for allocations. It references the ACCOUNT data model.

AMNT is the amount of the allocation in the system default currency.

Account Information



Account information is the general data format for types of information that can be attached to an account. It supports the Information Class and its children, Memo, FollowUpMemo, Flag and Alert. All information types are associated with an account, and may be optionally associated with a single transaction.

In a practical sense, a Memo is a small piece of text that can be viewed by a CSR to give more information as to why something is happening on an account. It may be generated by the system or by a user, and exists to give a human readable log of actions on the account.

A flag is a predefined, computer readable piece of information on an account. Flags can be read by humans, but can also be used as part of an automated decision making process. For example, there might be an insufficient funds flag. Based on configuration, the system may issue certain types of holds or bars, depending on such a flag.

An alert is a message placed on an account that is displayed when an account is accessed by a CSR. A practical example might be that the address on the account is found to be incorrect. In addition to putting a flag on the account to show an incorrect address, the system might also have an alert that simply informs the CSR that there is a problem with the address on the account. The CSR would then be able to make a decision as to how to proceed with a student, if presented with that information.

TYPE is set to one of M, F or A to define the type of object that is to be created.

RESP_ENTITY is the identifier for the entity that created the account information.

The CREATION_DATE is set when the information is saved to the system. The EFFECTIVE_DATE for the information is the date on which the memo is considered “in force”, allowing information to be placed on the account that does not show until the future. The EXPIRATION_DATE indicated the date after which the information is considered not effective. None of these dates affect the actual existence of the account information (an expired piece of information is not deleted) but they allow a counselor a faster view of currently applicable information, rather than having to trawl through comments that no longer make sense. A comment can be set to expire on its own (by setting the EXPIRATION_DATE when the memo is created) or by calling the expire() method, which will populate the EXPIRATION_DATE with the previous day’s date.

FOLLOW_UP_ID indicates that a memo has had information added to it. This identifier points the system to the next memo in the chain.

EDITOR_ID would indicate that the memo has been edited by a super user. As a general rule, if another user needs to add information to a memo, they would create a follow –up memo, leaving the original memo intact. Certain users will be permitted to edit memos to remove inappropriate content. This user’s ID would be stored in this field.

LAST_UPDATE shows the last time the memo was altered. It will either be the CREATION_DATE, or, if the memo is edited by a super user, then the date of the last edit.

PRIOR_ID indicates that the memo is part of a chain, and links to the memo that precedes it. The existence of this field indicates that the memo is a follow-on memo.

TEXT is the actual text of the memo or the alert. This is a VARCHAR2 field, limiting the size of the memo to 4000 characters. Should more space be needed, a follow-on memo could be created.

TRANSACTION_ID may be populated if the memo relates specifically to a transaction. As an example, if a deferment were issued on an account, the system may prompt the user to enter a reason for the deferment, which would be then linked as a memo to the specific transaction.

FLAG_TYPE_ID_FK is the identifier for a flag that is stored in the table KSSA_FLAG. Flags are entirely user defined. As well as their flag code, flags also have a human-readable FRIENDLY_NAME.

LEVEL indicates the level of user who can view this information. All users of the KSA system have a LEVEL indicator, and anything that is their level or below is visible (for memos, flags, tags, etc.) It is envisaged that students would have a level of 0, so they could view certain flags, tags, and alerts. It is not expected that any memos would be visible to them.

ACCOUNT identifies the account identifier against which the information is lodged.

SEVERITY identifies the severity of the flag. The higher this number, the more severe. The actual meaning of the severity is decided by any rule that acts upon this value.

Flag_Type

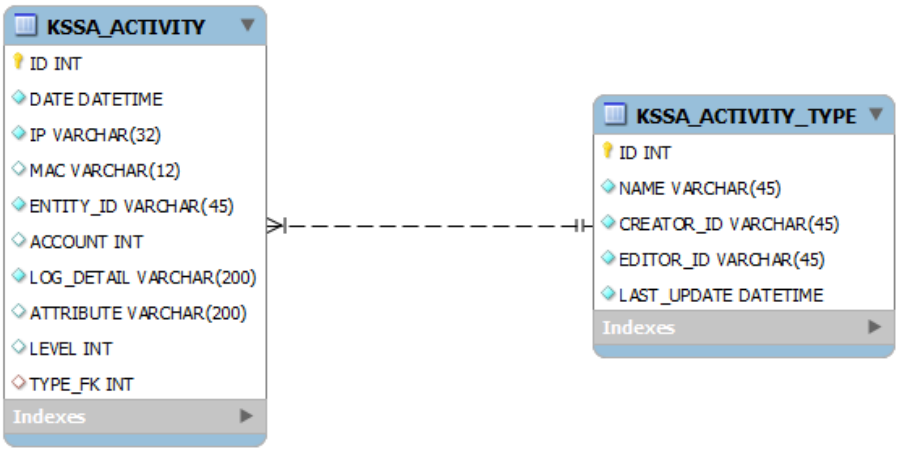
ID is the primary key of the flag.

NAME is the "friendly' human readable name of the flag. For example "Bounced Check"

LEVEL is the required level of the user to view the flag.

FLAG_INFO is a description of the flag with more detail than the high-level NAME field. For example "This user has bounced a check on their account. The bursar's office will not accept checks from those users who have in the past bounced a check. They should use another payment method."

Activity Data Model



KSSA_ACTIVITY persists the objects that make up the activity log, supporting the activity service. ID is an autonumber and serves as the primary key. The log stores the date of the activity, the IP and optionally the MAC address of the originating request, the entity who initiated the activity, the account to which the detail was applied (if applicable) the detail of the activity, a string of text that explains what has happened (example, “A new credit type was created”). Attribute is an optional attribute that explains the log better, so in the previous example, when a new credit type is created, the name of the credit type would be the most appropriate attribute to add to the log.

The level is the level of the activity, which is a numerical value, The lower the value, the more transactional the detail, the higher, the more serious the detail. For example, debugging information might be level 0, whereas a fatal error might be level 5.

Comment [PH5]: I would appreciate feedback as to best practices here. Apache’s logging levels seem reasonably appropriate.

TYPE_FK is a classification of the problem, for example EXCEPTION, SECURITY, etc. These types are defined in the KSSA_ACTIVITY_TYPE table.