Contents

Apply business logic using code

12/8/2021 • 2 minutes to read • Edit Online

Whenever possible, you should first consider applying one of the several declarative process options to define or apply business logic. More information: Apply business logic in Microsoft Dataverse

When a declarative process doesn't meet a requirement, as a developer you have several options. This topic will introduce common options to write code.

Create a plug-in

You can write a custom event handler, known as a plug-in, and register it on the Dataverse server. The plug-in is registered to execute on a specific event of the Dataverse database transaction. When executed, the plug-in can create, read, modify, or delete data being processed during the current database transaction. In this way, plug-ins allow you to customize or extend the data processing of Dataverse.

More information: Write plug-ins to extend business processes

Create a workflow extension

You can write and register custom workflow activities to provide additional actions within the process designer. Your new actions will be available in the workflow designer for users to apply - for example a condition or some new operation. In this way you can add new custom actions in the process designer for users of your environment.

More information: Workflow extensions

See also

Dataverse Developer Overview

Scalable Customization Design in Microsoft Dataverse

12/8/2021 • 7 minutes to read • Edit Online

NOTE

This is the first in a series of topics about scalable customization design. While this content has been divided into separate topics, it presents a wholistic view of concepts, issues, and strategies surrounding the design of scalable customizations. Each topic builds upon concepts established in preceding topics. You can download these topics as a single PDF document if you want to read it offline.

Dataverse is designed to protect itself and its users from long running activities that could affect both the response times for the user making a request and the stability and responsiveness of the system for other users.

A challenge faced by some people implementing Dataverse solutions are errors thrown by the platform or the underlying Microsoft SQL Server database when these protective measures take effect. This is often interpreted as the platform not being able to scale or incorrectly terminating or throttling requests to the system.

This content is based on experiences investigating and addressing the true underlying causes of the majority of these types of challenges. These topics describe how the platform protects itself from the impact of these requests imposed on the system and explains why this behavior is most often the result of custom implementations not understanding the impact on blocking and transaction usage within the platform.

This content also describes how optimizing a custom implementation to avoid these types of behaviors will not only avoid platform errors, but also enable better performance and end user experiences as a result. It provides good design practices and identifies common errors to avoid.

The challenge

Investigating and addressing the challenges in this area typically starts when certain types of errors and symptoms appear in the system. These are often perceived to be problems in the platform and the necessary remedial step is to loosen up the platform constraints that typically trigger a slow running request to become a reported error.

In reality, while the errors could be avoided in the short term by relaxing some of the platform constraints, these constraints are there for good reasons and are designed to prevent an excessively long running action from affecting other users or system performance. While the constraints could be relaxed to avoid the errors, users would still be experiencing slow response times and this would be affecting other users' experience of the system as well.

Therefore, it's preferable to look at the root causes of why these constraints are being triggered and causing errors, and then optimize the code customizations to avoid them. This will provide a more consistent and more responsive system for the users.

Common symptoms

These types of problems typically exhibit a combination of common symptoms as shown in the following table.

SYMPTOM	DESCRIPTION

SYMPTOM	DESCRIPTION
Slow requests	Users see slow response times for the system in particular areas, for example, certain forms and queries
Generic SQL errors	Certain actions respond with a platform error reporting a Generic SQL Error. This often translates at a platform layer to a SQL timeout.
Deadlocks	Platform errors reporting that a deadlock has occurred, which has forced the action to be terminated and rolled back.
Limited throughput	Particularly in batch load scenarios, this often exhibits in really slow throughput being achieved, much slower than should be possible.
Intermittent errors / slow performance	An important indicator of these behaviors is where the same action can be very fast or incredibly slow, and retrying it works much more quickly or avoids an error

In reality a combination of these symptoms can and often would be reported together when these challenges are faced. It's not always the case that these symptoms are an indicator of problems with the design. Other issues, such as disk I/O limitations in the database or a product bug, can cause similar symptoms. But the most common cause of these kinds of symptoms, and therefore one worth checking for, relates directly to the design of the custom implementation and how it affects the system.

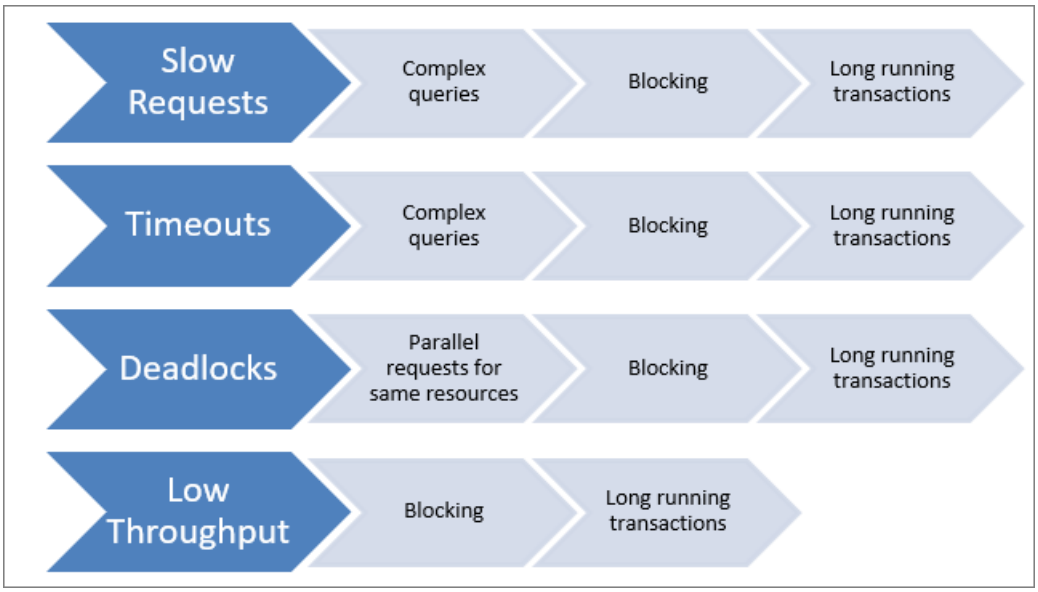
> *Why should we worry? Doesn't Dataverse just take care of this...?*

It does as far as it can... But it uses locking and transactions to protect the system against conflicts when required.

We also provide options for you to make choices about your particular scenario and to decide where it is important to control access to data. But those choices can be made incorrectly, and it is possible to introduce unintended consequences in custom code. These problems typically have an impact on the user experience through slower response times so understanding the implications of certain actions can lead to more consistent and better results for users.

Understanding causes

Common symptoms have causes that force particular requests to run slowly and then to trigger platform constraints. The following diagram shows typical symptoms with some of the common root causes of these symptoms.



The underlying impact of long running transactions, database blocking, and complex queries can all overlap with each other and amplify their effects to cause these symptoms. For example, a series of long running queries that are completely independent of each other may cause slow user response times, but only once they require access to the same resources do the response times become so slow that they become errors.

Design for platform constraints

The Dataverse platform has a number of deliberate constraints it imposes to prevent any one action having too detrimental an impact on the rest of the system and, therefore, on users. While this behavior can be frustrating since it can block specific requests from completing and often leads to questions around whether the constraints can be lifted, this is rarely a good approach when you consider the broader implications.

When the platform is used as intended and an implementation is optimized, it's very rare that there is a scenario where these constraints would be encountered. Running into the constraint is almost always an indication of behaviors that will be tying up resources excessively in the system. This means other requests either from the same user or other users can't be processed. So while it may be possible to loosen the constraint on the request being blocked, what that actually means is that the resources it is consuming are tied up for even longer causing bigger impacts on other users.

At the heart of these constraints is the idea that the Dataverse platform is designed to support a transactional, multi-user application where quick response to user demand is the priority. It's not intended to be a platform for long running or batch processing. It is possible to drive a series of short requests to Dataverse but Dataverse isn't designed to handle batch processing. Equally, where there are activities running large iterative processing, Dataverse isn't designed to handle that iterative processing.

In those scenarios, a separate service can be used to host the long running process, driving shorter transactional requests to Dataverse itself. For example, using Flow or hosting Microsoft SQL Server Integration Services (SSIS) elsewhere and then driving individual create or update requests to Dataverse is a much better pattern than using a plug-in to loop through thousands of records being created in Dataverse.

It is worth being aware of and understanding the platform constraints that do exist, so that you can allow for them in your application design. Also, if you do encounter these errors, you can understand why they are happening and what you can change to avoid them.

CONSTRAINT	DESCRIPTION

CONSTRAINT	DESCRIPTION
Plug-in timeouts	• Plug-ins will time out after 2 minutes • Don't preform long running operations in plug-ins. Protects the platform and the sandbox service and ultimately the user from poor user experience
SQL timeouts	• Requests to SQL Server time out at 30 seconds • Protects against long running requests • Provides protection within a particular organization and its private database • Also provides protection at a database server level against excessive use of shared resources such as processors/memory
Workflow limits	• Operates under a Fair Usage policy • No specific hard limits, but balance resource across organizations • Where demand is low an organization can take full advantage of available capacity. Where demand is high, resources and throughput are shared.
Maximum concurrent connections	• There is a platform default setting of a maximum connection pool limit of 100 connections from the Web Server connection pool in IIS to the database. Dataverse does not change this value • If you hit this, it is an indication of an error in the system; look at why so many connections are blocking • With multiple web servers, each with 100 concurrent connections to the database of typical < 10ms, this suggests a throughput of > 10k database requests for each web server. This should not be required and would hit other challenges well before that
ExecuteMultiple	• The `ExecuteMultiple` message is designed to assist with collections of operations being sent to Dataverse from an external source • The processing of large groups of these requests can tie up vital resources in Dataverse at the expense of more response critical requests by users.
Service Protection Limits	• To ensure consistent availability and performance for everyone we apply some limits to how APIs are used. These limits are designed to detect when client applications are making extraordinary demands on server resources. • More information: Service Protection API Limits

Next steps

To understand how the platform constraints are applied it is necessary to understand database transactions.
More information: Scalable Customization Design: Database transactions

Scalable Customization Design: Database transactions

12/8/2021 • 13 minutes to read • Edit Online

NOTE

This is the second in a series of topics about scalable customization design. To start at the beginning, see Scalable Customization Design in Microsoft Dataverse.

One of the most fundamental concepts behind many of the challenges faced here is that of the database transaction. In Dataverse the database is at the heart of almost all requests to the system and the place data consistency is primarily enforced.

- No Dataverse data operations, either internal or part of code customizations, work completely in isolation.
- All Dataverse data operations interact with the same database resources, either at a data level or an infrastructure level such as processor, memory, or I/O usage.
- To protect against conflicting changes, each request takes locks on resources to be viewed or changed.
- Those locks are taken within a transaction and not released until the transaction is committed or aborted.

Transaction and locking awareness

A common reason that problems can occur in this area is the lack of awareness of how customizations can affect transactions.

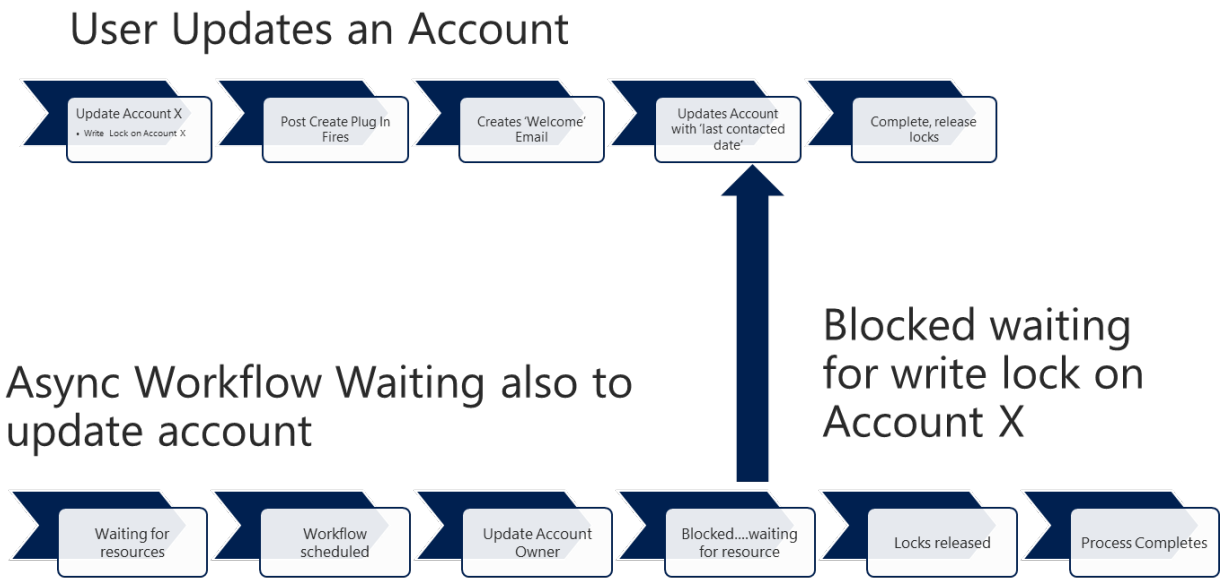
Although the details of how this is done is beyond the scope of this topic, the most simple element to consider is that as Dataverse interacts with data in its database. SQL Server determines the appropriate locks to be taken by transactions on that data such as:

- When retrieving a particular record, SQL Server takes a read lock on that record.
- When retrieving a range of records, in some scenarios it can take a read lock on that range of records or the entire table.
- When creating a record, it generates a write lock against that record.
- When updating a record, it takes a write lock against the record.
- When a lock is taken against a table or record, it's also taken against any corresponding index records.

However, it's possible to influence the scope and duration of these locks. It's also possible to indicate to SQL Server that no lock is required for certain scenarios.

Let's consider SQL Server database locking and the impact of separate requests trying to access the same data. In the following example, creating an account has set up a series of processes, some with plug-ins that are triggered as soon as the record is created, and some in a related asynchronous workflow that is initiated at creation.

The example shows the consequences when an account update process has complex post processing while other activity also interacts with the same account record. If an asynchronous workflow is processed while the account update transaction is still in progress, this workflow could be blocked waiting to obtain an update lock to change the same account record, which is still locked.



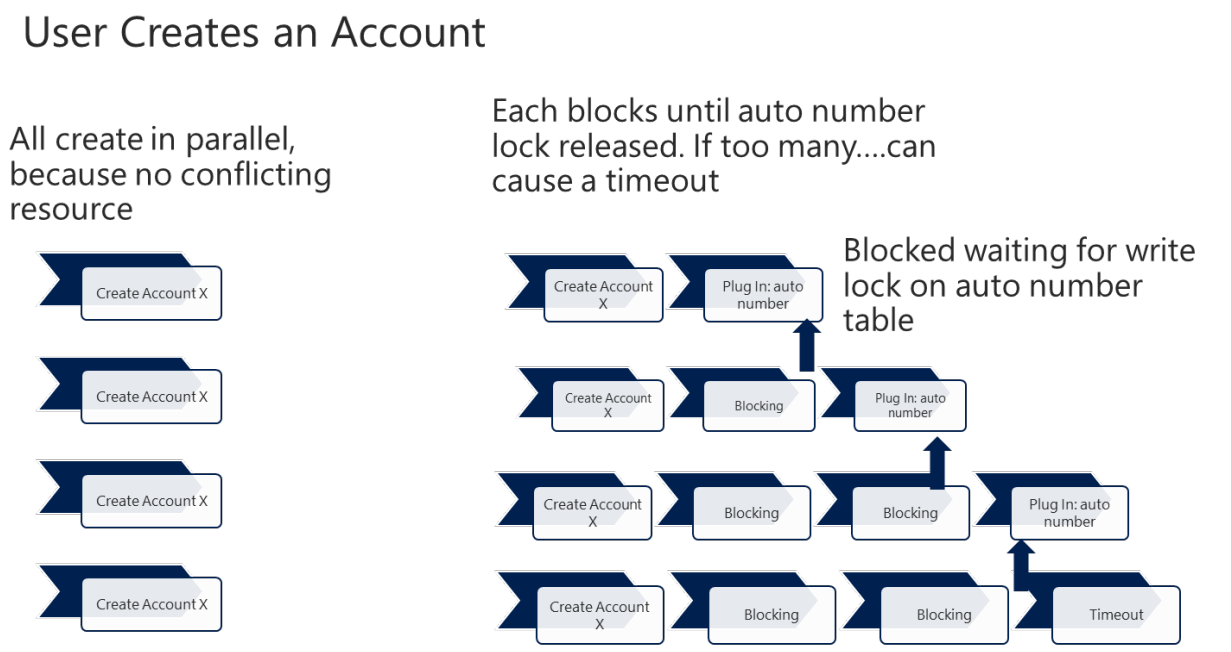
It should be noted that transactions are only held within the lifetime of a particular request to the platform. Locks aren't held at a user session level or while information is being shown in the user interface. As soon as the platform has completed the request, it releases the database connection, the related transaction, and any locks it has taken.

Blocking

While the kind of blocking in the previous example can be inconvenient in and of itself, this can also lead to more serious consequences when you consider that Dataverse is a platform that can process hundreds of concurrent actions. While holding a lock on an individual account record may have reasonably limited implications, what happens when a resource is more heavily contested?

For example, when each account is given a unique reference number it may lead to a single resource that is tracking the used reference numbers being blocked by every account creation process. As described in the Auto-numbering example, if a lot of accounts are generated in parallel, overlapping requests will all need to access that auto-numbering resource and will block it until they complete their action. The longer each account creation process takes, and the more concurrent requests there are, the more blocking occurs.

While the first request to grab the auto-number resource lock can easily be completed, the second request will need to wait for the first to complete before it can check what the next unique reference number is. The third request will have to wait for both the first and second requests to complete. The more requests there are, the longer blocking will occur. If there are enough requests, and each request takes long enough, this can push the later requests to the point that they time out, even though individually they may complete correctly.



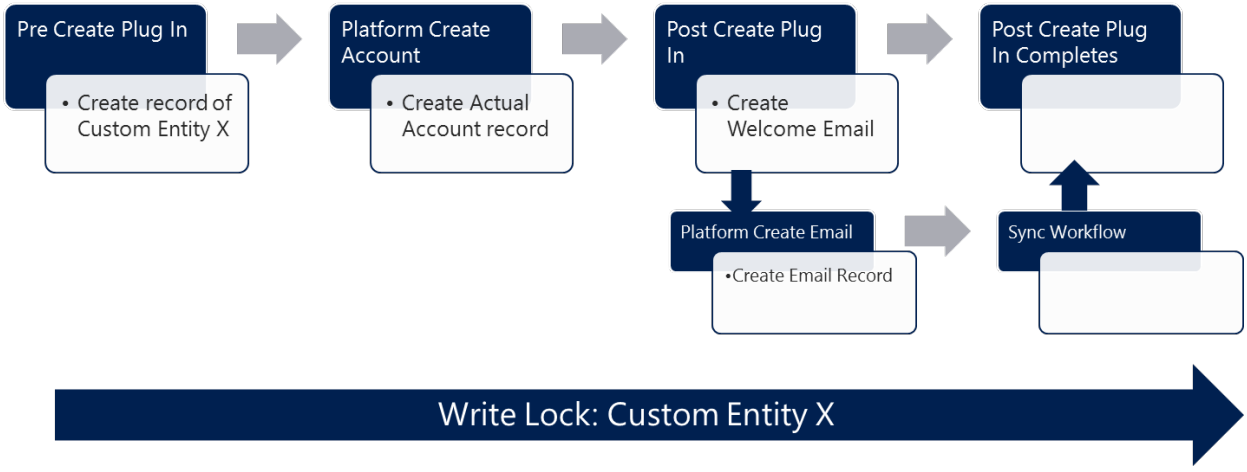
Lock release

There are two primary reasons why a lock isn't released but is held until the transaction is completed:

- The database server holds onto the lock for consistency in case the transaction will later make another request to update the data item.
- The database server also has to allow for the fact that an error or abort command issued later can cause it to roll the entire transaction back, so it needs to hold onto the locks for the entire lifetime of the transaction to ensure consistency.

It is important to recognize that even though your process may have completed any interactions with a particular piece of data, the lock will be held until the entire transaction is complete and committed. The longer the transaction is extended, the longer the lock will be held, preventing other threads from interacting with that data. As will be shown later, this also includes related customizations that work within the same transaction and can significantly extend the lifetime of transactions such as synchronous workflows.

In the following example, the write lock on a custom entity in the pre create plug-in for an account is locked until all logic tied to the creation of the account is completed.



Intermittent errors: timing

Intermittent behavior is an obvious symptom of blocking from concurrent activity. If repeating exactly the same action later succeeds when earlier it failed, there is a very strong likelihood that the error or slowness was

caused by something else occurring at the same time.

That is important to realize as debugging a problem often involves stripping the offending functionality back to the bare minimum. However, when the problem only occurs intermittently, you may need to look at where the failing action is conflicting with another activity in the system, and you need to look at potential contention points. You can mitigate conflict by optimizing an individual process; however, the shorter the processing time, the less likely the activity will conflict with other processes.

Transaction control

While in most cases the way transactions are used can simply be left to the platform to manage, there are scenarios where the logic needed is complex enough that understanding and influence over transactions is required to achieve the desired results. Dataverse offers a number of different customization approaches that impact differently on the way transactions are used.

When you understand how each type of customization participates in the platform transactions, you can model complex scenarios effectively in Dataverse and predict their behavior.

As mentioned earlier, a transaction is only held during the lifetime of a request to the platform, it is not something that is maintained once the platform step is completed. This avoids transactions being held by an external client for long periods and blocking other platform activities.

The job of the platform is to maintain consistency throughout the platform transaction pipeline and where appropriate allow customizations to participate in that same transaction.

How Model-driven Apps use transactions

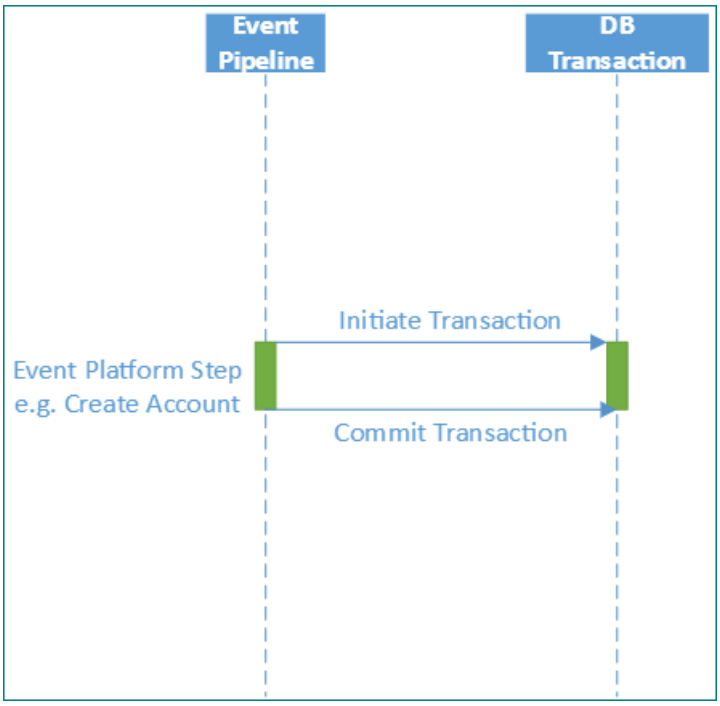
Before understanding how customizations interact with the platform, it is useful to understand how model-driven apps use requests to the platform, and how it affects transaction use.

OPERATION	DESCRIPTION
Forms (Retrieve)	• Takes a read lock on the record shown. • Low impact on other uses.
Create	• Performs a create request through the platform • Low impact on other activities, as a new record nothing else blocking on it • Can potentially block locking queries to the whole table until it is complete. • Often can trigger related actions in customization which can have an impact.
Update	• Performs an update request through the platform. • More likely to have conflicts. An update lock will block anything else updating or reading that record. Also blocks anything taking a broad read lock on that table. • Often triggers other activities.
View (RetrieveMultiple)	• Would think this would block lots of other activity. • Although poor query optimization can affect DB resource usage and possibly hit timeouts.

Event pipeline: platform step

When an event pipeline is initiated, a SQL transaction is created to include the platform step. This ensures that all database activity performed by the platform is acted on consistently. The transaction is created at the start of

the event pipeline and either committed or aborted when the processing is completed, depending on whether it was successful.



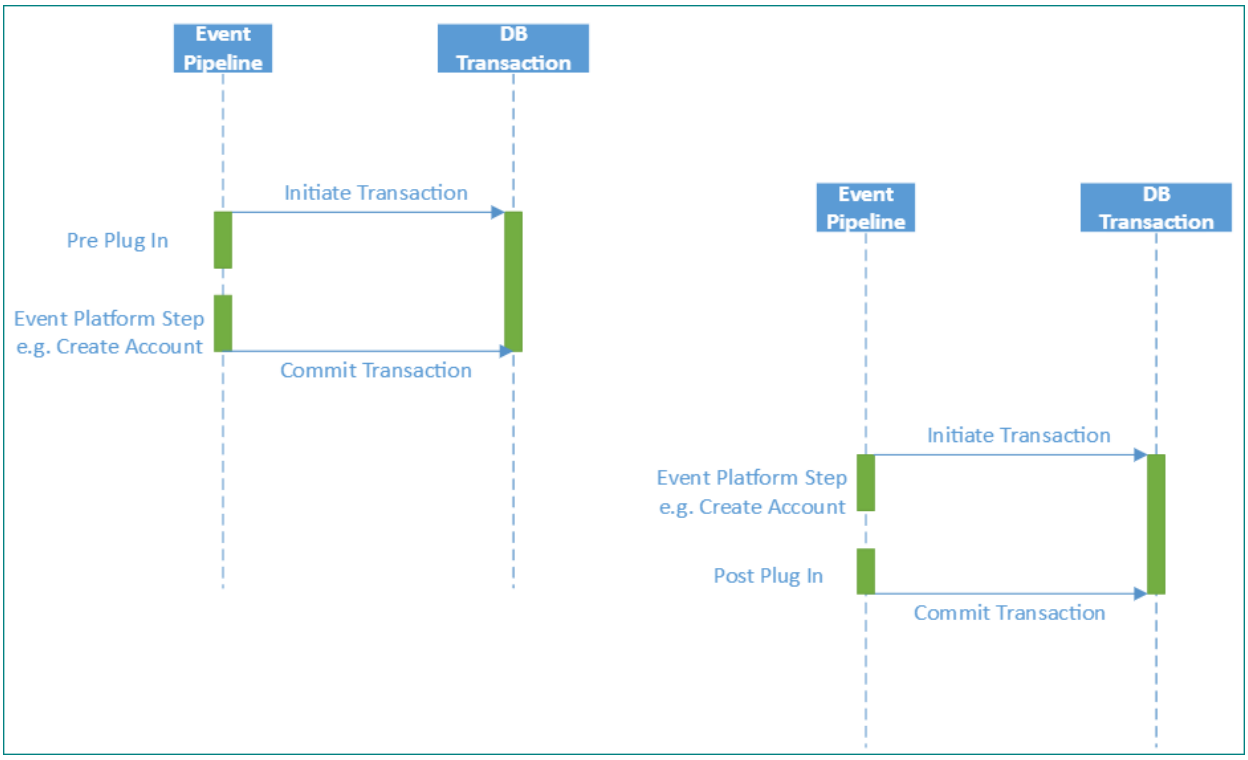
Customization requests

It's also possible to participate in the platform initiated transaction within customizations. Each type of customization participates in transactions in a different way. The following sections will describe each in turn.

- Sync plug-ins (pre or post operation: in transaction context)
- Sync plug-ins (pre and post operation: in transaction context)
- Sync plug-ins (**PreValidation**: outside transaction context)
- Sync plug-ins (**PreValidation**: in transaction context)
- Async plug-ins
- Plug-in transaction use summary
- Synchronous workflows
- Asynchronous workflows
- Custom workflow activity
- Custom actions
- Web service requests

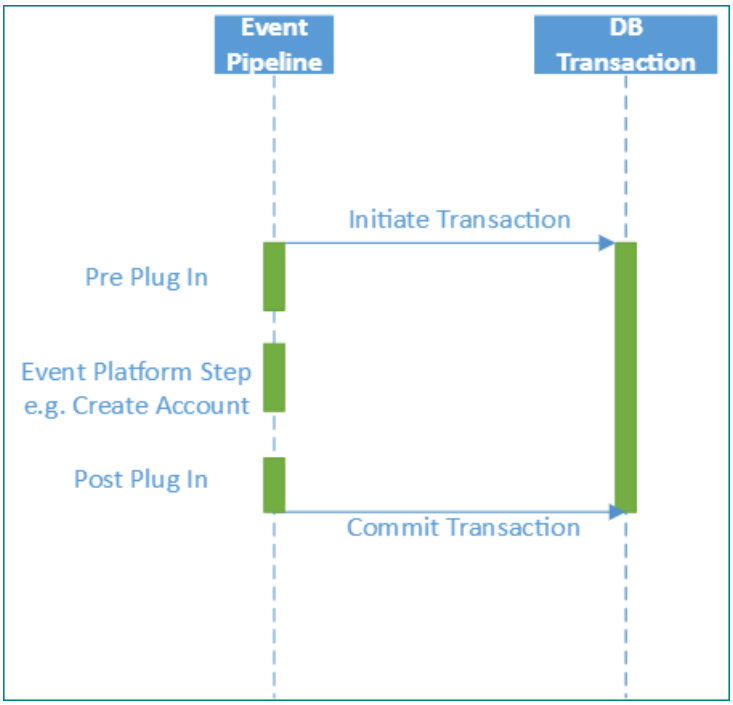
Sync plug-ins (pre or post operation: in transaction context)

When plug-ins are registered for an event, they can be registered against a **PreOperation** or **PostOperation** stage that is within the transaction. Any message requests from the plug-in will be performed within the transaction. This means the lifetime of the transaction, and any locks taken, will be extended.



Sync plug-ins (pre and post operation: in transaction context)

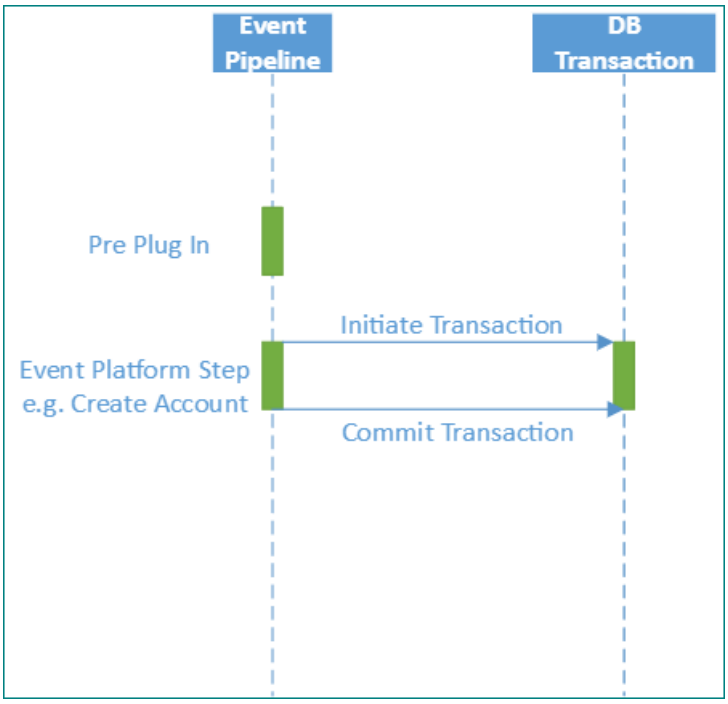
Plug-ins can be registered against both the **PreOperation** and **PostOperation** stages. In this case the transaction can extend even further because it will extend from the start of the **PreOperation** plug-in until the **PostOperation** plug-in completes.



Sync plug-ins (PreValidation: outside transaction context)

A plug-in can also be registered to act outside of the platform transaction by being registered on the **PreValidation** stage.

> **NOTE**
>
> It does NOT create its own transaction. As a result, each message request within the plug-in is acted upon independently within the database.



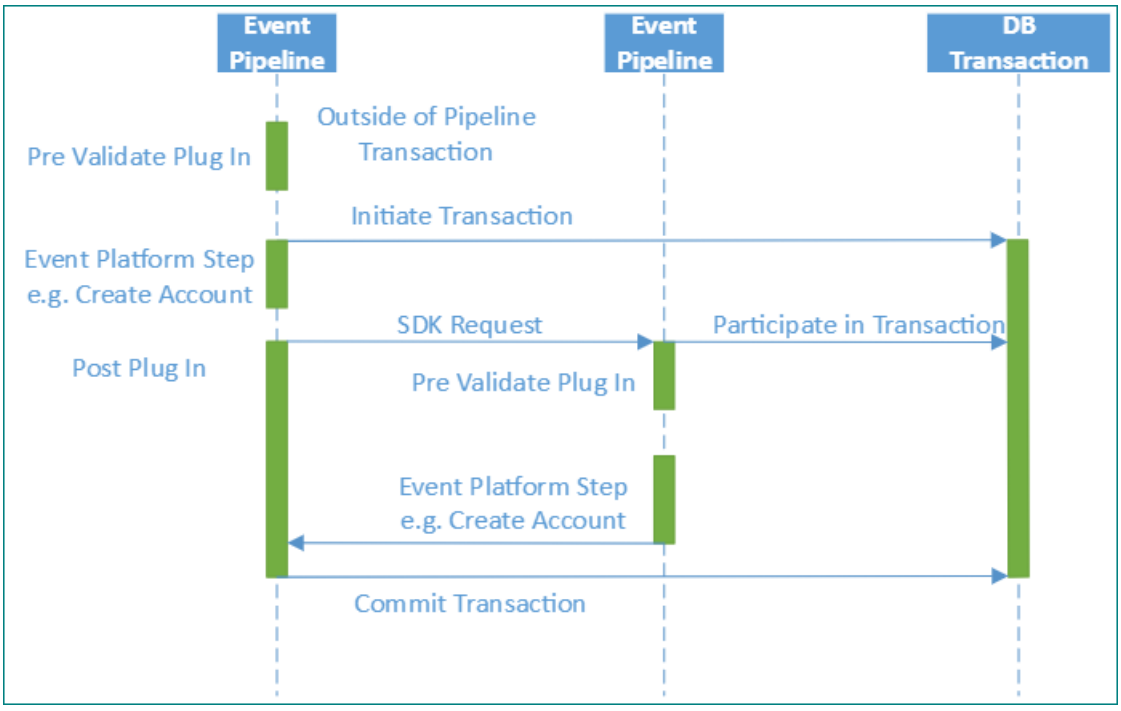
This scenario only applies when the **PreValidation** is called as the first stage of a pipeline event . Even though the plug-in is registered on the **PreValidation** stage, it is possible it will participate in a transaction as the next section describes. It can't be assumed that a **PreValidation** plug-in doesn't participate in a transaction, although it is possible to check from the execution context if this is the case.

Sync plug-ins (PreValidation: in transaction context)

The related scenario occurs when a **PreValidation** plug-in is registered but the related pipeline event is triggered by message request from within an existing transaction.

As the following diagram shows, creating an account can cause a **PreValidation** plug-in to perform initially outside of a transaction when the initial create is performed. If, as part of the post plug-in, a message request is made to create a related child account because that second event pipeline is initiated from within the parent pipeline, it will participate in the same transaction.

In that case, the **PreValidation** plug-in will discover that a transaction already exists and so will participate in that transaction even though it's registered on the **PreValidation** stage.

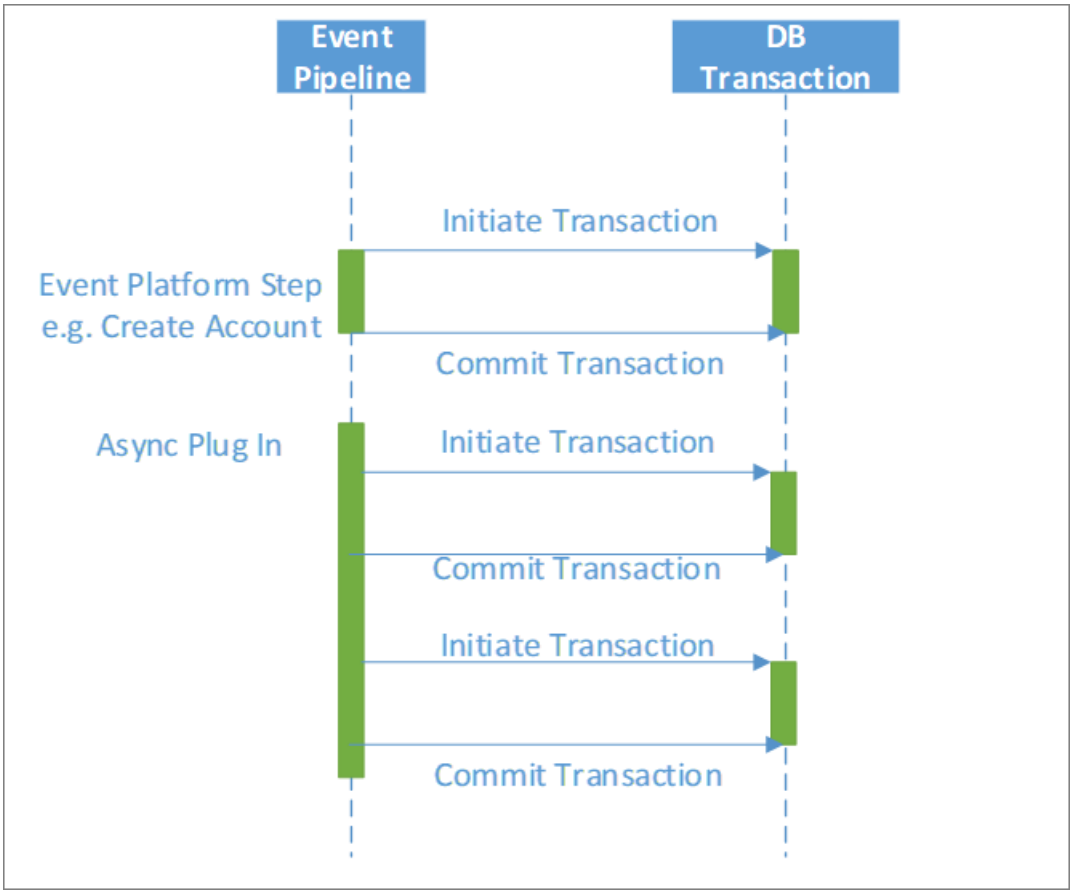


As previously mentioned, the plug-in can check the execution context for the IsInTransaction property, which will indicate if this plug-in is performing within a transaction or not.

Async plug-ins

A plug-in can also be registered to act asynchronously. In this case, the plug-in also acts outside of the platform transaction.

> **NOTE**
> The plug-in doesn't create its own transaction; each message request within the plug-in is acted upon independently.



Plug-in transaction use summary

To summarize:

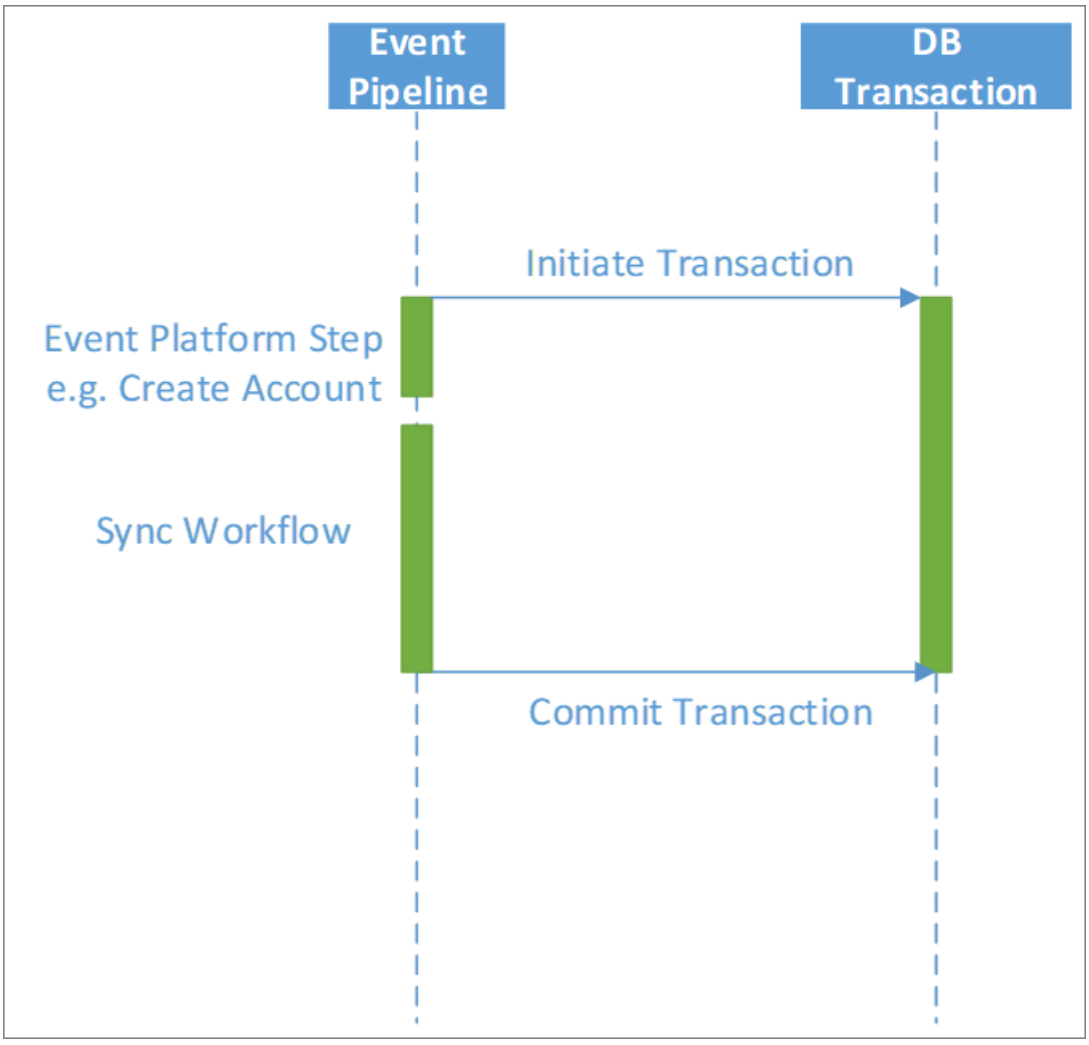
- Synchronous plug-ins typically participate in transactions.
- Async plug-ins never participate in a platform transaction; each request is performed independently.
- **PreValidation** plug-ins don't create a transaction but participate if one already exists.

EVENT	STAGE NAME	TRANSACTION DOES NOT YET EXIST	TRANSACTION ALREADY EXISTS
Pre-Event	**PreValidation**	No transaction is created. Does not participate in transaction; each request uses independent transaction to the database	Participates in existing transaction
Pre-Event	**PreOperation**	Participates in existing transaction	Participates in existing transaction

EVENT	STAGE NAME	TRANSACTION DOES NOT YET EXIST	TRANSACTION ALREADY EXISTS
Post-Event	**PostOperation**	Participates in existing transaction	Participates in existing transaction
Async	N/A	No transaction is created. Does not participate in transaction; each request uses independent transaction to the database	N/A

Synchronous workflows

From the perspective of transactions, synchronous workflows act as pre/post operation plug-ins. They therefore act within the platform pipeline transaction and can have the same effect on the length of the overall transaction.



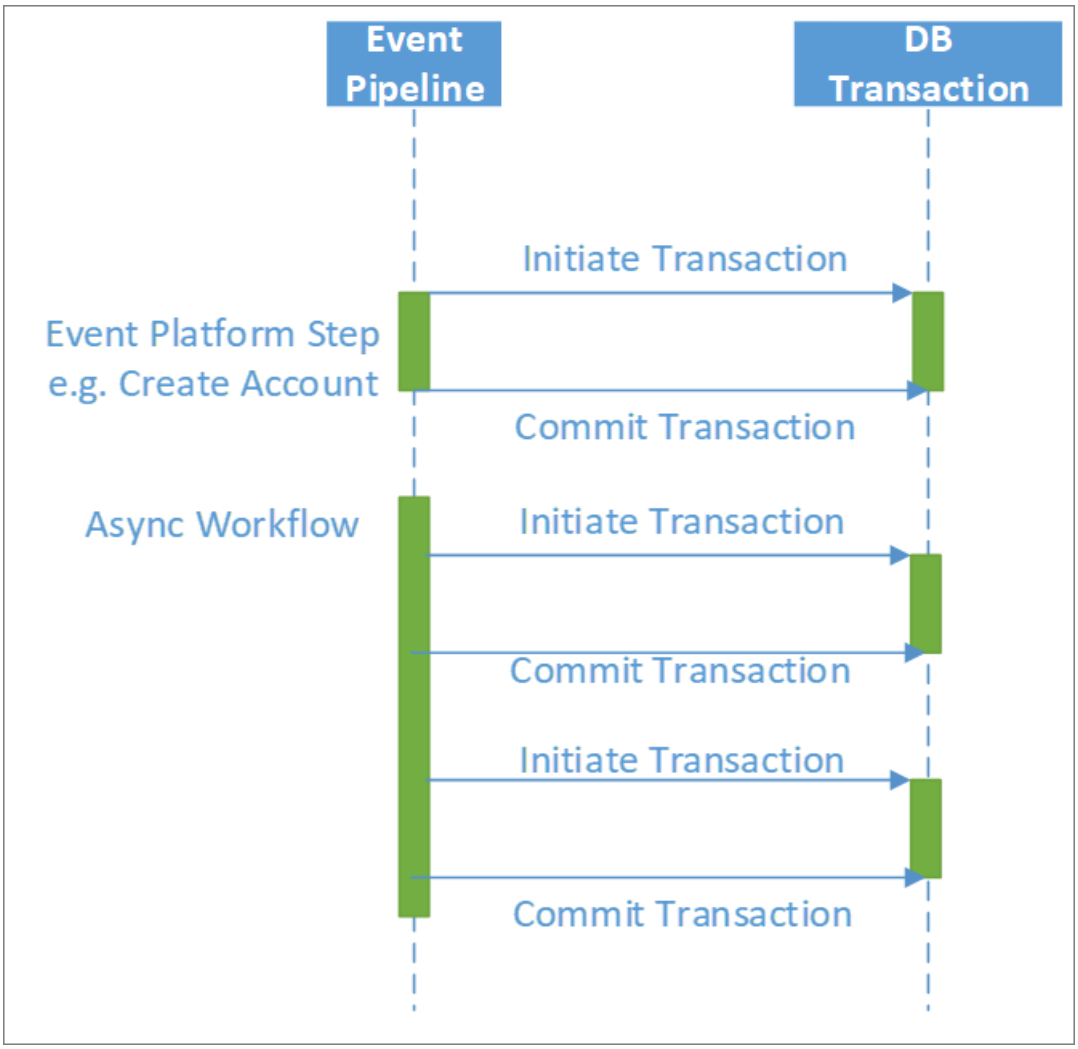
Asynchronous workflows

Asynchronous workflows are triggered outside of the platform transaction.

> **NOTE**
> The workflow also does NOT create its own transaction and therefore each message request within the workflow is acted upon independently.

The following diagram shows the asynchronous workflow acting outside of the platform transaction and each

step initiating its own independent transaction.

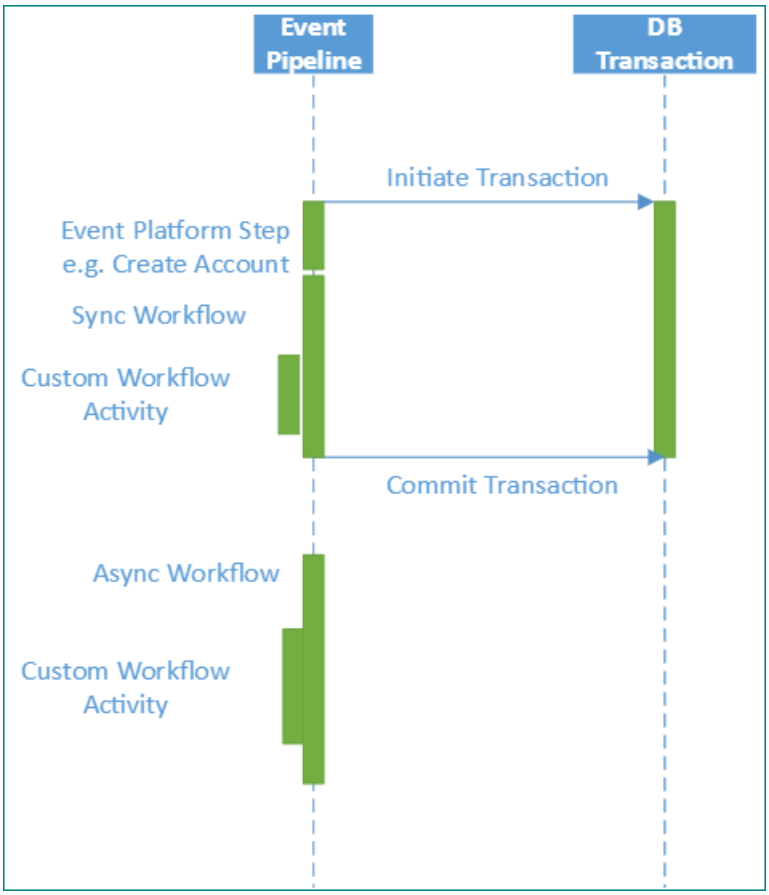


Custom workflow activity

Custom workflow activities act within the parent workflow context.

- Sync workflow: Acts within the transaction
- Async workflow: Acts outside the transaction

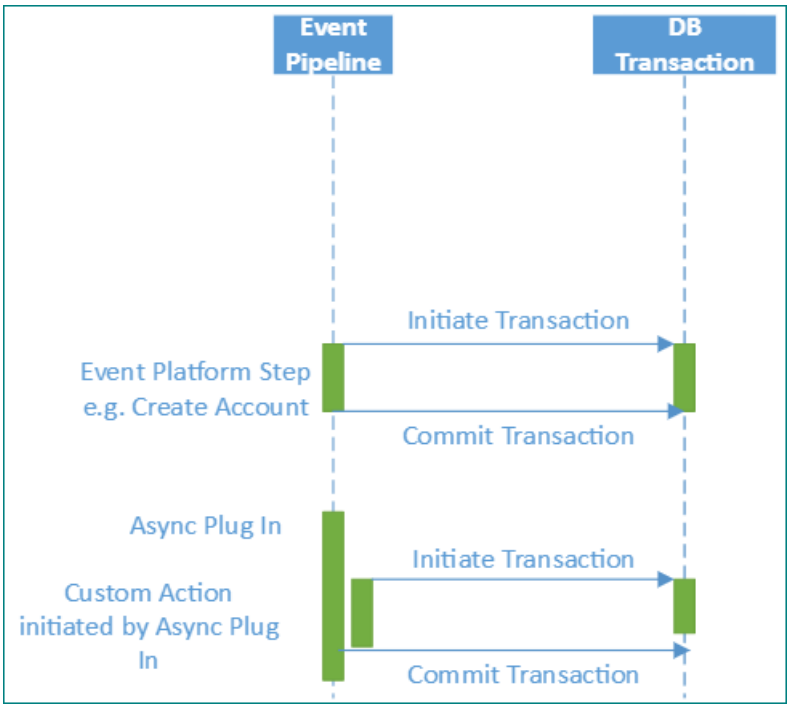
The following diagram shows custom activities first acting within a synchronous workflow and then within an asynchronous workflow.



Custom actions

Custom actions can create their own transactions. This is a key feature. A custom action can create a separate transaction outside of the platform step depending on whether it is configured to Enable Rollback.

- Enable Rollback set
  - If called through a message request from a plug-in running within the transaction, and Enable Rollback is set, the custom action will act within the existing transaction.
  - The custom action will otherwise create a new transaction and run within that.
- Enable Rollback not set
  - The custom action won't act within a transaction.



Web service requests

When requests are made externally through web services, a pipeline is created and transaction handling within the pipeline occurs as previously discussed, but a transaction is not maintained once the response is returned. Since how long it will be until the next request is an unknown, the platform does not allow locking of resources which would block other activities.

When multiple requests are made within a plug-in using the same execution context, it is the common execution context that maintains the transaction reference and in synchronous plug-ins ensures each request is made within the same transaction. The ability to maintain an execution context across requests is not available outside of plug-ins and therefore a transaction cannot be maintained across separate requests made externally.

There are two special messages where multiple actions can be passed to the Dataverse platform as part of a single web service request.

MESSAGE	DESCRIPTION
`ExecuteMultiple`	This allows multiple independent actions to be passed within the same web service request. Each of these requests is performed independently within the platform so there is no transaction context held between requests.
`ExecuteTransaction`	This allows multiple actions to be processed within the same database transaction, in a similar way to multiple message requests made from within a synchronous plug-in. This ability would also have implications similar to multiple message requests; that is, if each action takes a long time (such as by making expensive queries or triggering a long chain of related synchronous plug-ins or workflows) this could lead to blocking issues in the broader platform.

Web API (OData) Requests in plug-ins

Do not use Web API (OData) requests within a plug-in to the same organization as the plug-in. Always use the IOrganizationService methods. This allows for the transaction context to be passed so that the operation can participate in the pipeline transaction.

Next steps

In addition to database transactions, it is important to appreciate the impact of multiple concurrent data operations can have on the system. More information: Scalable Customization Design: Concurrency issues

Scalable Customization Design: Concurrency issues

12/8/2021 • 4 minutes to read • Edit Online

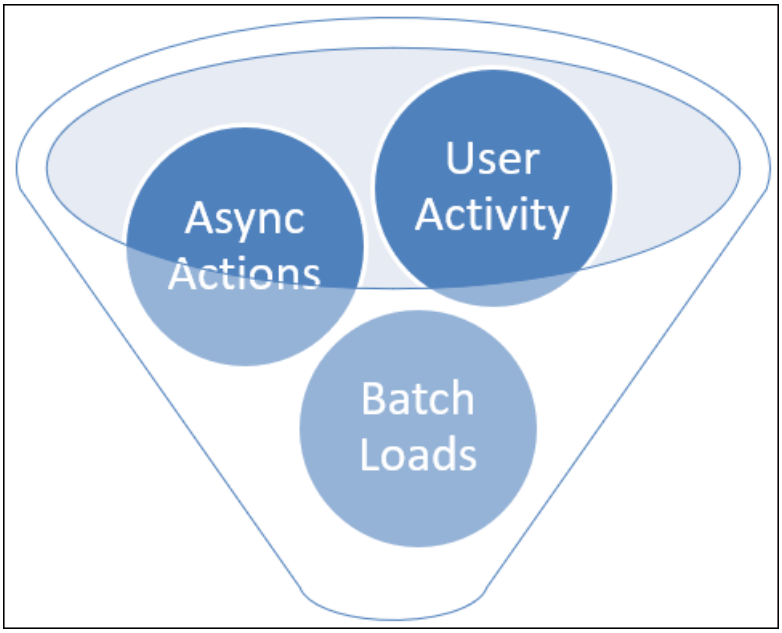
NOTE

This is the third in a series of topics about scalable customization design. To start at the beginning, see Scalable Customization Design in Microsoft Dataverse. The previous topic Scalable Customization Design: Database transactions described how database transactions are applied and the effect they have on different types of customizations.

When you have concurrent requests, the chance of collisions on locks becomes higher. The longer the transactions take, the longer the locks are held. The chances are even higher of collision and the overall impact would be greater on end users.

You also need to be aware of the multiple ways that activity can be driven onto the application, each of which is taking locks that could cause conflict with other actions within the system. In these cases, locking is preventing inconsistencies of data occurring when overlapping actions occur on the same data.

Some key areas to consider design for, and check for if you do see problems, are:



- **User driven activity**: Directly through the user interface.
- **Async actions**: Activity that occurs later as a result of other actions. When this activity will be processed isn't known at the time the initiating action is triggered.
- **Batch activities**: Either driven from within Dataverse such as bulk delete jobs or server side synchronization processing), or driven from external sources such as integration from another system.

Async operations in parallel

A common misconception is that async workflows or plug-ins are processed serially from a queue and there wouldn't be conflict between them. This isn't accurate, since Dataverse processes multiple asynchronous activities in parallel both within each async service instance and across async service instances spread over different servers to increase throughput. Each async service actually retrieves jobs to be performed in batches of approximately 20 per service based on configuration and load.

If you initiate multiple asynchronous activities from the same event on the same record, they're likely to process in parallel. As they fired on the same record, a common pattern is updates back to the same parent record; therefore the conflict opportunity is high.

When a triggering event occurs, such as the creation of an account, asynchronous logic in Dataverse may create entries in the AsyncOperation (System Job) Entity for each process or action to be taken. The Async Service monitors this table, picks up waiting requests in batches, and then processes them. Because the workflows are triggered at the same time, they are highly likely to be picked up in the same batch and processed at the same time.

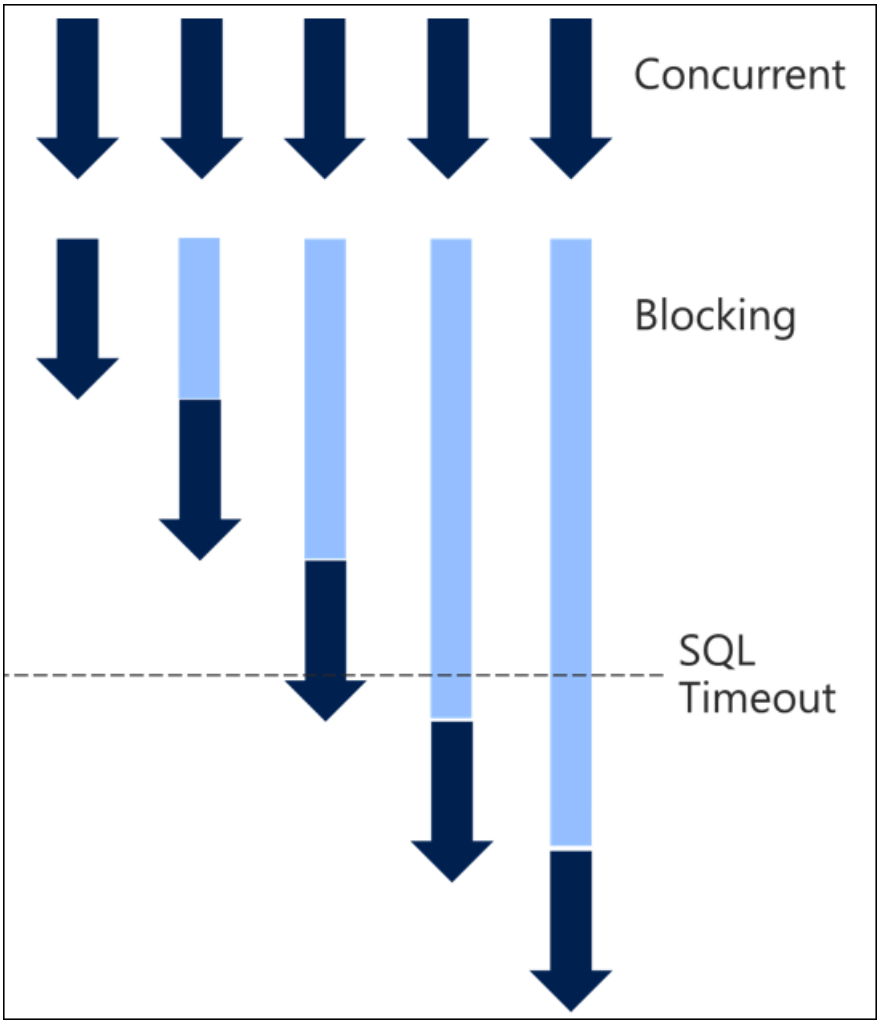
Why it's important to understand transactions

The Auto-numbering example provides a scenario that illustrates how database transactions and concurrency issues need to be considered when designing scalable customizations.

Serialization and Timeouts

A high degree of serialization is typically what turns blocking into timeouts and poor throughput. When you have many concurrent requests, once they serialize and take a long time to process, each request in turn takes longer and longer until you start hitting timeouts and therefore errors.

The plug-in timeout starts from when it is initiated. A SQL timeout is calculated on the database request, so if a query blocks waiting for a long time, it can time out.

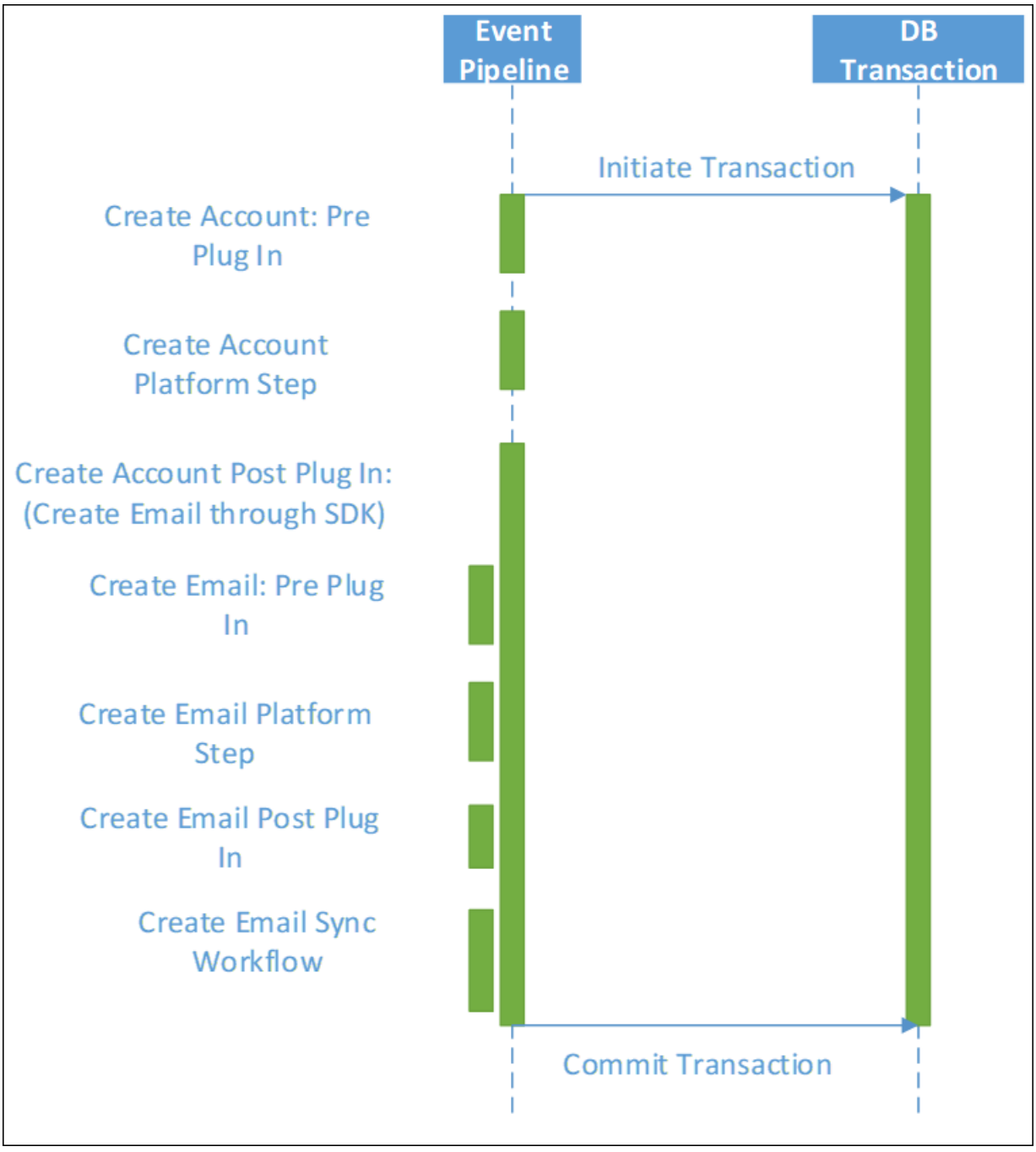


Chain of actions

As well as understanding the specific queries in the directly triggered activities, it is also necessary to consider

where a chain of related events may occur.

Each message request made in a plug-in or as a step in a synchronous workflow not only triggers the direct action but may also cause other synchronous plug-ins and workflows to fire. Each of these synchronous activities will occur in the same transaction, extending the life of that transaction and any locks held possibly much longer than may be realized.



The overall effect may be much greater than initially realized. This can often happen unintentionally where multiple people are building up the implementation, or it evolves over time.

Running into platform constraints

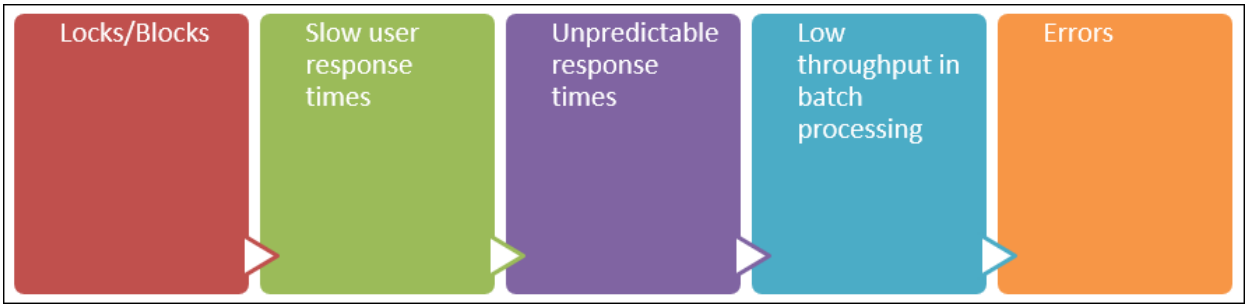
This is where the platform constraints can come in. And in reality this sort of behavior is exactly what the constraints are there to protect the broader system from.

Whenever this level of delay of processing occurs, it will have unintended consequences in other areas of the system and on other users. It's therefore important to prevent this kind of activity from interfering with system performance.

While the easy way to avoid the errors may be to relax the platform constraints, this is not addressing the more fundamental impact on the overall scalability and performance of the system. This needs to be addressed by fixing and preventing the behavior triggering the constraints in the first place.

Impact on usage

What often also has an impact on usage is a cascading series of implications of this behavior.



The initial issue is locks and therefore blocking in the system. This leads to slow user response times, which is then amplified as unpredictable and unreliable user response times, often in a particular area of the system.

In the extreme case or under heavier than normal load, this can then show through in any background batch processing with poor throughput. Eventually it can all escalate into errors occurring in the system.

It is common that when investigating SQL timeout errors, users are also reporting poor and unpredictable response times, and the connection had not been made between these as related issues.

Next steps

Understand design patterns you can apply (or avoid) to minimize performance issues. More information: Scalable Customization Design: Transaction design patterns

Scalable Customization Design: Auto-numbering example

12/8/2021 • 4 minutes to read • Edit Online

> **NOTE**
> This example supports a series of topics about scalable customization design. To start at the beginning, see Scalable Customization Design in Microsoft Dataverse.

One scenario that illustrates the common misunderstanding of how transactions are handled within Dataverse is implementing an automatic numbering scheme.

In this scenario the requirement is typically to:

- Generate a unique number following a particular pattern.
- Allow for many concurrent requests to create the related type of records; for example, accounts that need a unique reference.
- Allow for sequential numbering of the unique numbers.
- Ensure that the number generation is consistent, but scalable, and does not error out under load. It also needs to ensure that duplicate numbers cannot be generated.
- Generate the number on the creation of the relevant record.

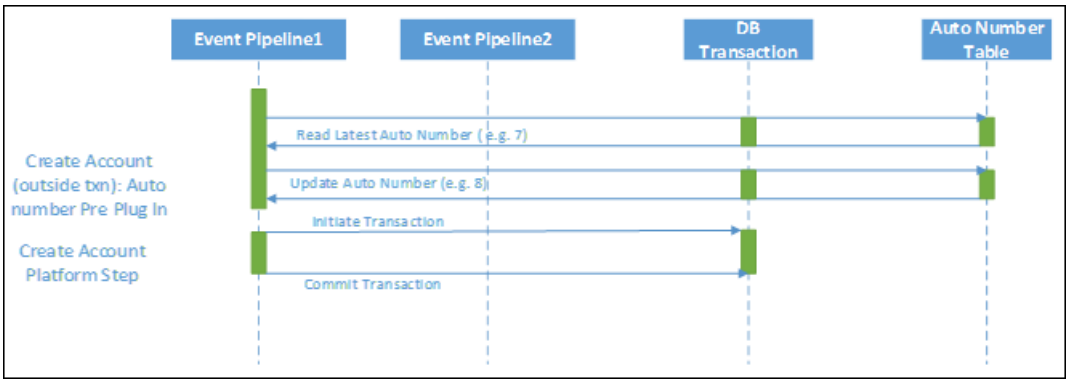
The typical approach involves variations of the following:

- Store the last used number in an auto number index data store; for example, a custom entity with a row per data type.
- Retrieve the last used number and increment that number.
- Record the new number against the newly generated record.
- Store the new number back as being the last used number in the auto number index store.

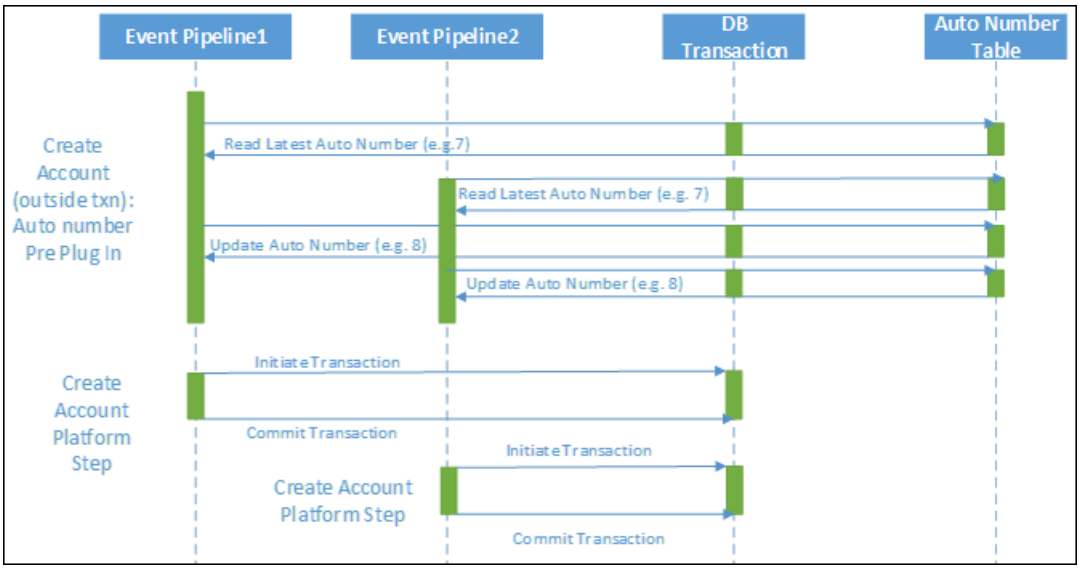
The following sections describe different approaches that can be taken within Dataverse and highlight the implications, showcasing both the importance and benefit of understanding the way transactions are utilized.

Approach 1: Out of a transaction

The simplest approach is to realize that any use of a commonly required resource would introduce the potential for blocking. Since this has an impact on scalability you may decide you want to avoid a platform transaction when generating an auto number. Let's consider the scenario for auto numbering generation outside of the pipeline transaction in a pre-validation plug-in.



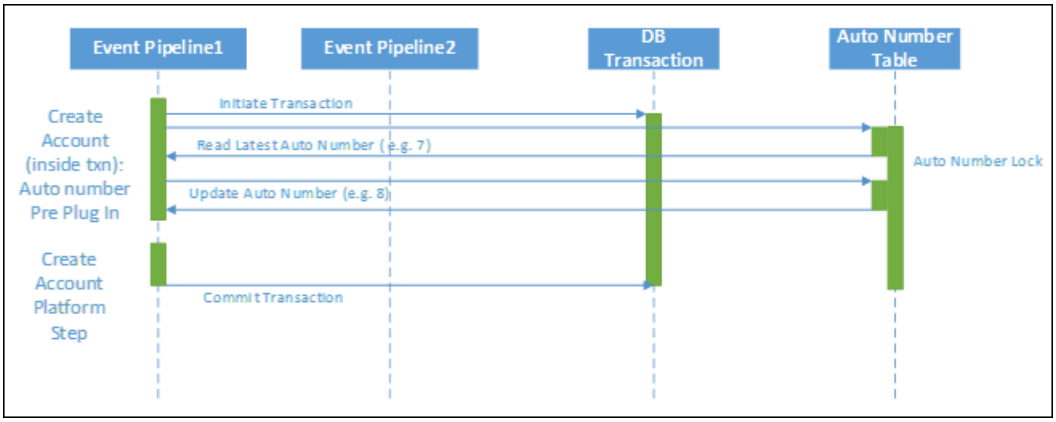
When you run this in isolation it works fine. It doesn't, however, actually protect against concurrency errors. As the following diagram shows, if two requests in parallel both request the latest number and then both increment and update the value, you'll end up with duplicate numbers. Because there is no locking held against the retrieved number, it is possible for a race condition to occur and both threads to end up with the same value.



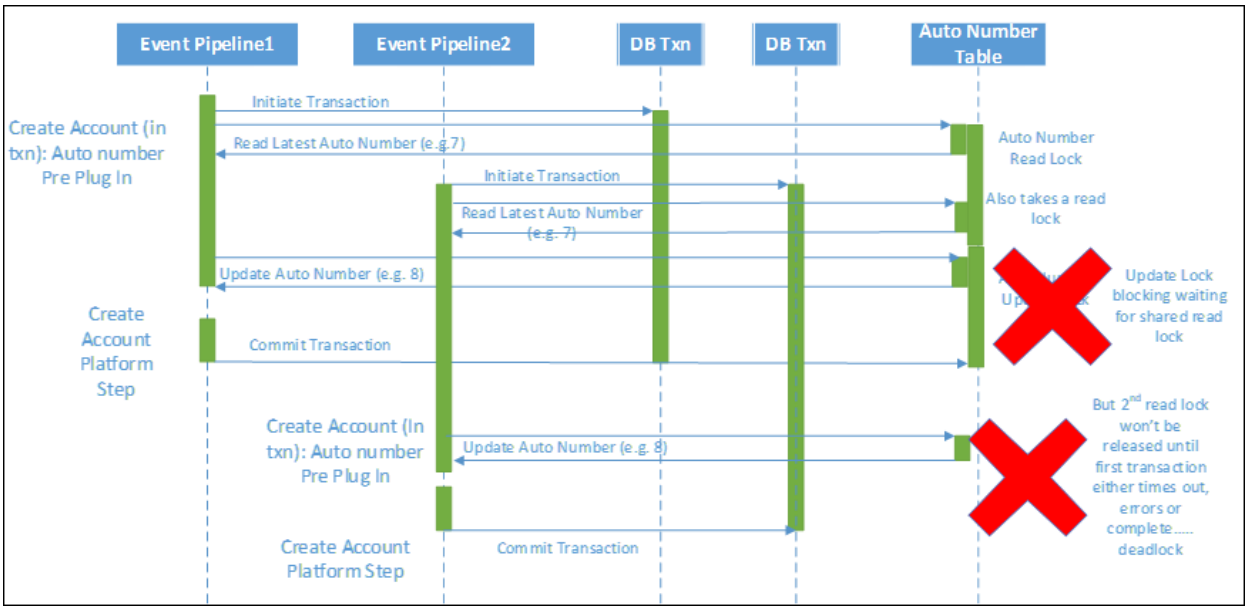
In many cases, even though multiple requests may be occurring, due to the limited window for overlapping this could work fine, but it is relying on luck rather than good design to prevent the duplication.

Approach 2: In a plug-in transaction

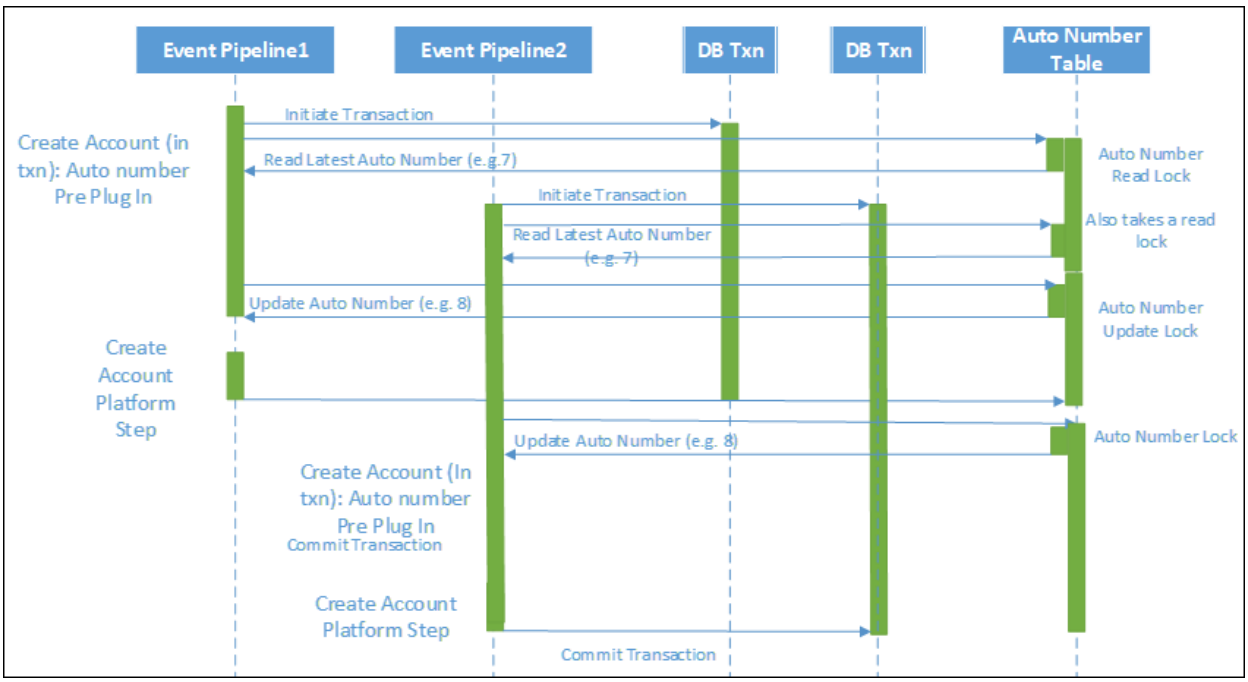
If you do the auto numbering from a plug-in registered within the transaction (txn), surely this works....right?



In the same circumstances of overlapping requests trying to generate numbers at the same time, it would be possible for both requests to be granted a shared read lock on the auto numbering table. Unfortunately, at the point the application tries to upgrade this to an exclusive lock, this would not be possible as there would be another shared read lock preventing this.



Depending on how the queries are being generated, the exact behavior can vary, but relying on those conditions and not being certain of the outcome where the uniqueness is essential isn't ideal. Even if this does not generate a failure, the shared read ability could allow a duplicate number to be generated if the isolation modes aren't correct. As the following diagram shows, both records end up with the same auto number value of 8.

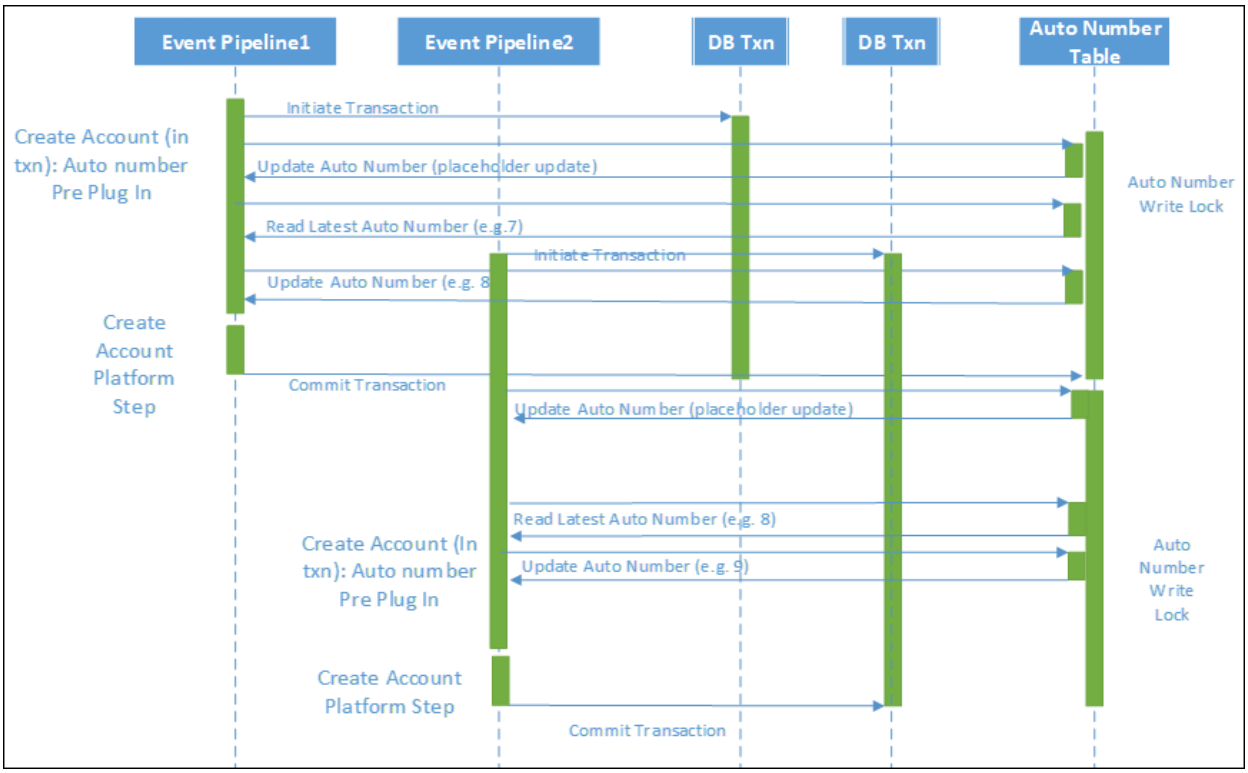


Approach 3: Pre-lock in a plug-in transaction

Understanding the way the transactions work leads to being able to generate a safe way to do this.

In this approach, from the start of the transaction, a placeholder update is performed on the auto numbering record to some field (for example, UpdateInProgress) used purely for the purpose of maintaining consistency. It does this by writing an update indicating an update is about to start. This process then requests and takes an exclusive write lock on that row in the auto numbering table, blocking other processes from starting the auto numbering approach.

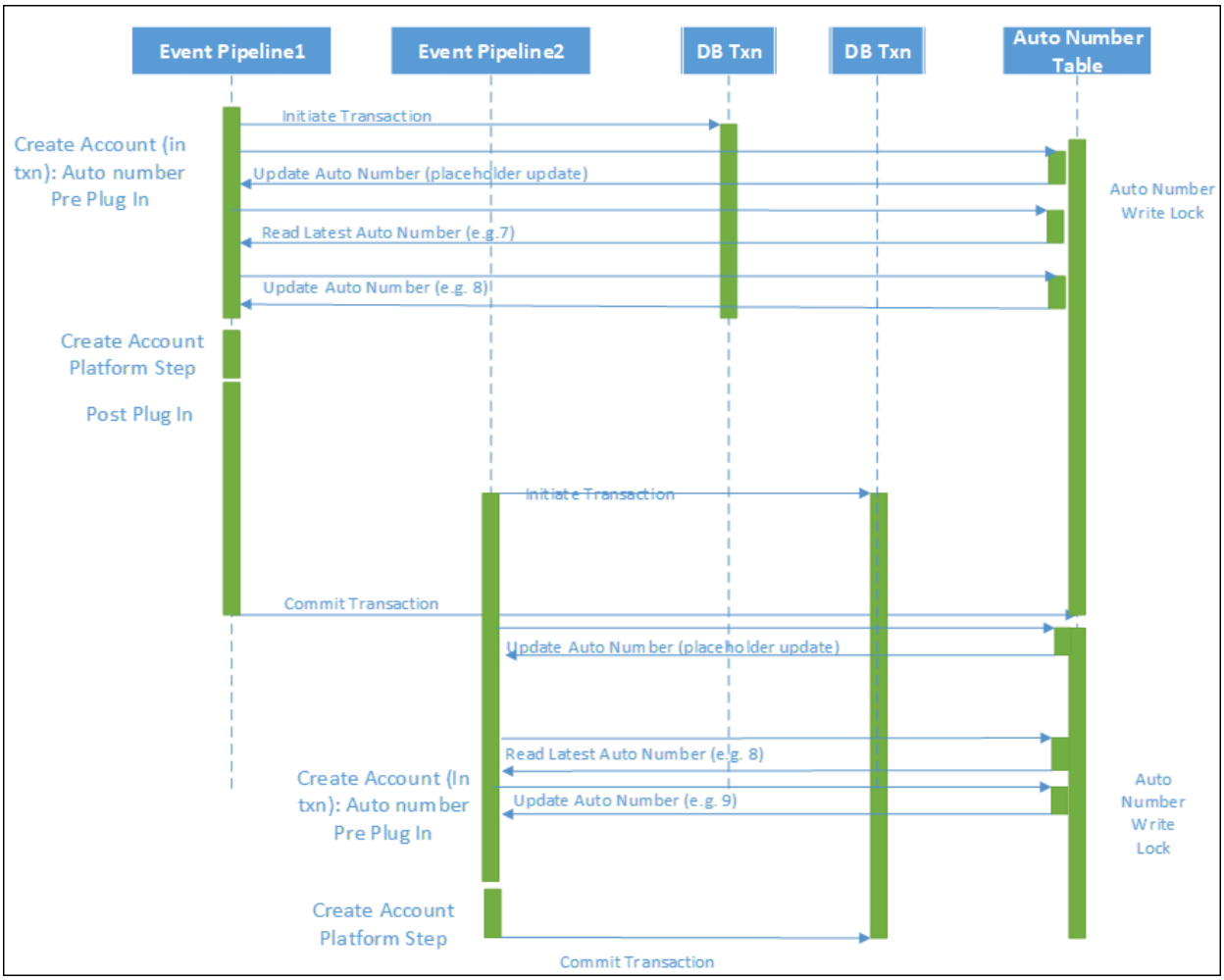
This then allows you to safely increment and write back the updated auto number without any other process being able to interfere.



It does have the implication that this will serialize not only the auto numbering updates but also the account creation requests as both these steps occur in the same platform transaction. If the creation of accounts are quick actions then that may be a perfectly good approach and it ensures that account creation and auto numbering are performed consistently; if one fails they both fail and roll back.

In fact, where the other actions within the transaction are quick, this is the most consistent and efficient approach for implementing auto numbering in customizations.

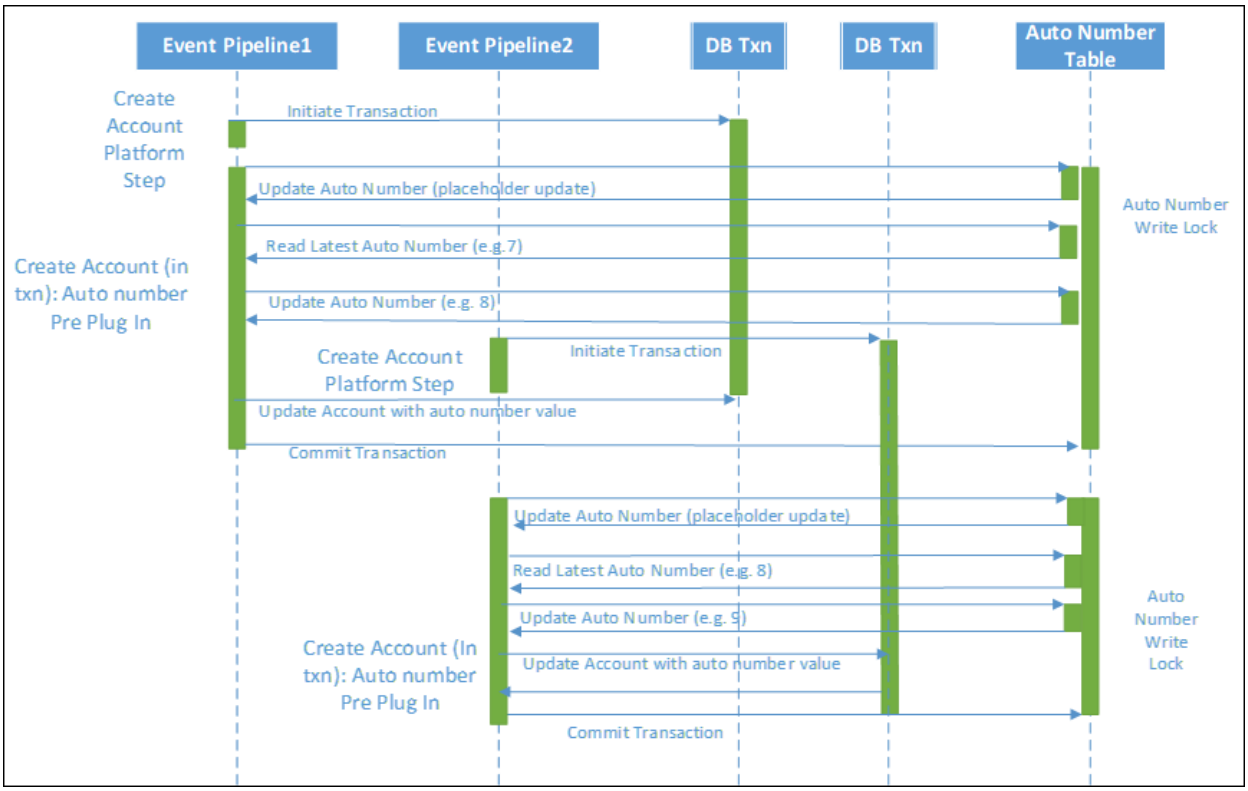
If however, you also introduce other synchronous plug-ins or workflows that each take extended amounts of time to complete, serialization can become a real scalability challenge, as the auto numbering process not only blocks itself but blocks waiting for the other activities to complete.



Normally, generation of the auto number would be done in a pre-event plug-in. You include the number in the input parameters to the create step and avoid a second update in the post processing to record the generated auto number against the account.

With the scalability implications in mind, if there is other complex processing in the account creation process, an alternative would be to move the auto number generation to a post create process, which still ensures a consistent update process. The benefit would be that it reduces the length of time within the transaction that the auto number record lock is held as the lock is only taken towards the end of the process. If the auto numbering table is the most highly contested resource and this approach is taken for all processes accessing it, this reduces the amount of contention overall.

The tradeoff here would be the need to perform an additional update to account, while reducing the overall length of time blocking waiting for the auto numbering record.


Event Pipeline1
Event Pipeline2
DB Txn
DB Txn
Auto Number Table
Create Account Platform Step
Initiate Transaction
Update Auto Number (placeholder update)
Auto Number Write Lock
Read Latest Auto Number (e.g.7)
Create Account (in txn): Auto number Pre Plug In
Update Auto Number (e.g. 8)
Create Account Platform Step
Initiate Transaction
Update Account with auto number value
Commit Transaction
Update Auto Number (placeholder update)
Read Latest Auto Number (e.g. 8)
Auto Number Write Lock
Update Auto Number (e.g. 9)
Create Account (In txn): Auto number Pre Plug In
Update Account with auto number value
Commit Transaction

Scalable Customization Design: Transaction design patterns

12/8/2021 • 16 minutes to read • Edit Online

NOTE

This is the fourth in a series of topics about scalable customization design. To start at the beginning, see Scalable Customization Design in Microsoft Dataverse.

This section describes design patterns to avoid or minimize and their implications. Each design pattern needs to be considered in the context of the business problem being solved and can be useful as options to investigate.

Don't avoid locking

Locking is a very important component of SQL Server and Dataverse, and is essential to healthy operation and consistency of the system. For this reason it is important to understand its implications on design, particularly at scale.

Transaction usage: Nolock hint

One capability of the Dataverse platform that is heavily used by views is the ability to specify that a query can be performed with a nolock hint, telling the database that no lock is needed for this query.

Views use this approach because there is no direct link between the act of launching the view and subsequent actions. A number of other activities could happen either by that user or others in between and it is not practical or beneficial to lock the entire table of data the view shows waiting until the user has moved on.

Because a query across a large data set means that it potentially affects others trying to interact with any of that data, being able to specify that no lock is required can have a significant benefit on system scalability.

When making a query of the platform through the SDK it can be valuable to specify that nolock can be used. It indicates that you recognize this query doesn't need to take a read lock in the database. It's especially useful for queries where:

- There's a broad scope of data
- Highly contested resources are queried
- Serialization isn't important

Nolock shouldn't be used if a later action depends on no change to the results, such as in the auto number locking example earlier.

An example scenario where it can be useful is determining if an email is related to an existing case. Blocking other users from creating new cases to ensure there is no possibility of a case being generated that the email could link to would not be a beneficial level of consistency control.

Instead, making a reasonable effort to query for related cases and attach the email to an existing case or create a new one, while still allowing other cases to be generated, is more appropriate. Particularly, as there is no inherent link in the timing between these two actions, the email could just as easily have come in a few seconds earlier and no link would have been detected.

Whether nolock hints are valid for a particular scenario would typically be based on a judgment of the likelihood

and impact of conflicts occurring and the business implication of not ensuring consistency of actions between a retrieve and subsequent actions. Where no business impact would occur from avoiding locking, using nolocks would be a valuable optimization choice. If there is a potential business impact, the consequence of this can be weighed against the performance and scalability benefits of avoiding locking.

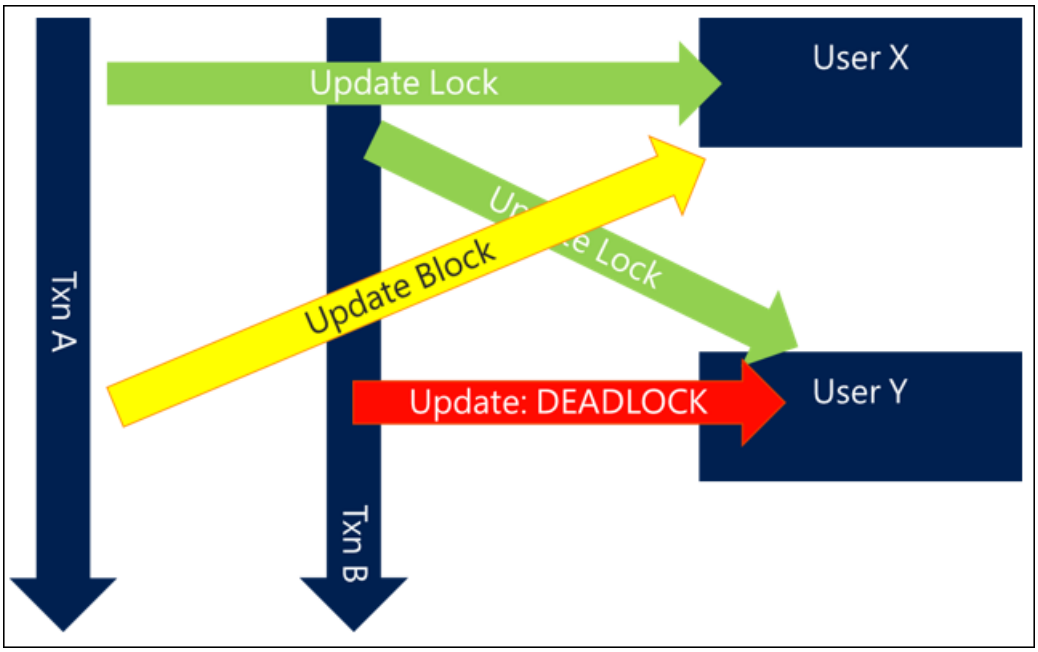
Consider order of locks

Another approach that can be useful in reducing the impact of blocking, and particularly in avoiding deadlocking, is having a consistent approach to the ordering of locks in an implementation.

A simple and common example is when updating or interacting with groups of users. If you have requests that update related users (such as adding members to teams or updating all participants in an activity), not specifying an order can mean that if two concurrent activities try to update the same users, you can end up with the following behavior, which results in a deadlock:

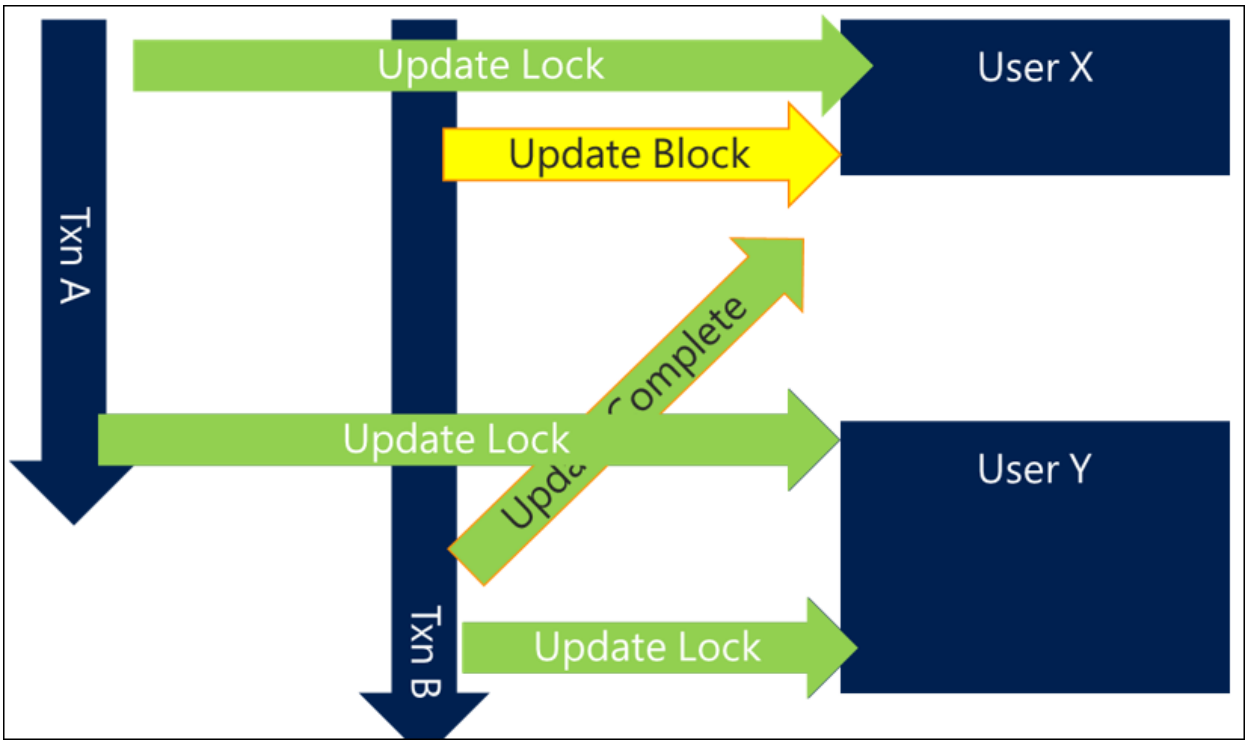
- Transaction A tries to update User X and then User Y
- Transaction B tries to update User Y and then User X

Because both requests start together, Transaction A is able to get a lock on User X and Transaction B is able to get a lock on User Y, but as soon as each of them try to get a lock on the other user they block and then deadlock.



Simply by ordering the resources you access in a consistent way you can prevent many deadlock situations. The ordering mechanism is often not important as long as it is consistent and can be done as efficiently as possible. For example, ordering users by name or even by GUID can at least ensure a level of consistency that avoids deadlocks.

In a scenario using this approach, Transaction A would get User X, but Transaction B would also try now to get User X rather than User Y first. While that means Transaction B blocks until Transaction A completes, this scenario avoids the deadlock and is completed successfully.



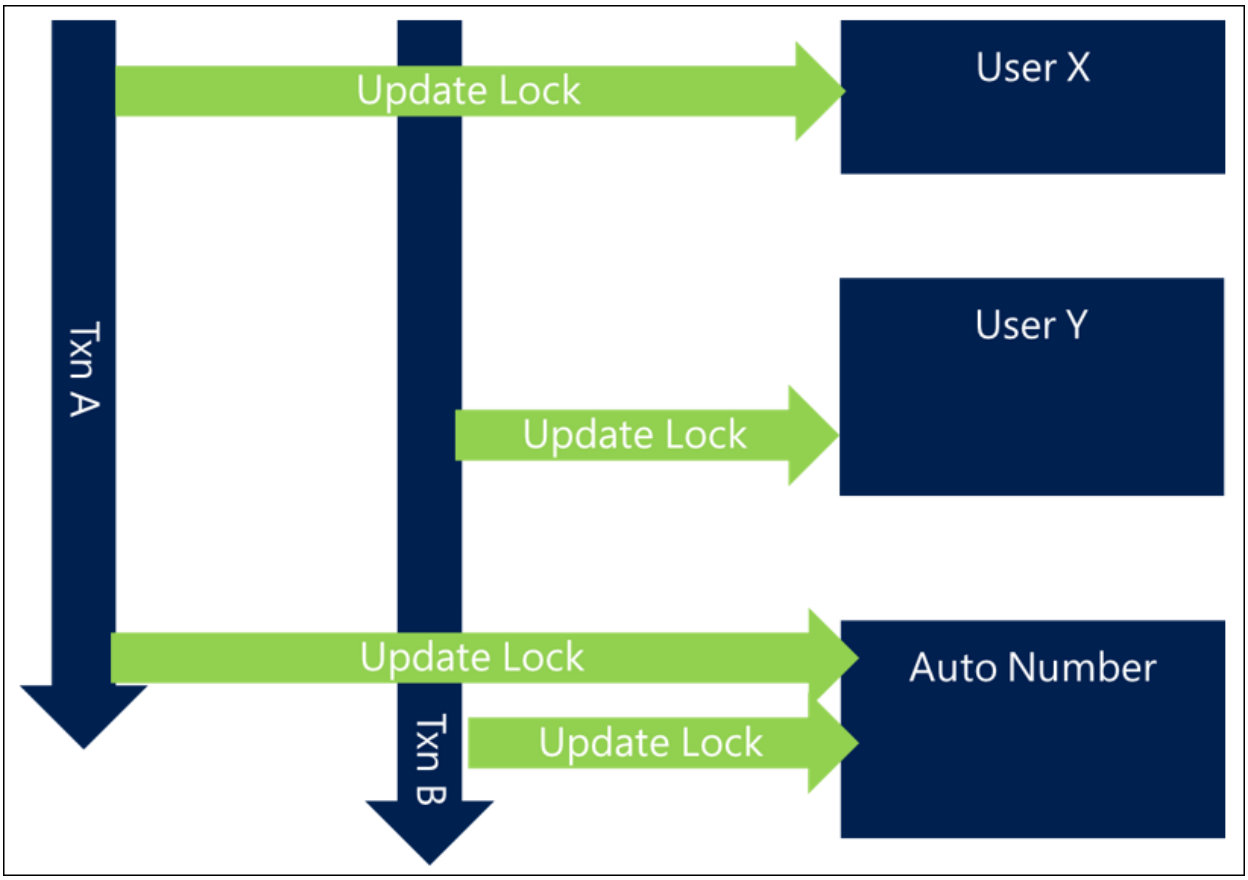
In more complex and efficient scenarios, it may be that you lock least commonly referenced users first and more frequently referenced users last, which leads to the next design pattern.

Hold contentious locks for shortest period

There are scenarios, such as the auto numbering approach, where there is no way around the fact that there is a heavily contested resource that needs to be locked. In that case, the blocking problem can't be completely avoided, but can be minimized.

When you have heavily contested resources, a good design is to not include the interaction with that resource at the functionally logical point in the process, but move the interaction with it to as close to the end of the transaction as possible.

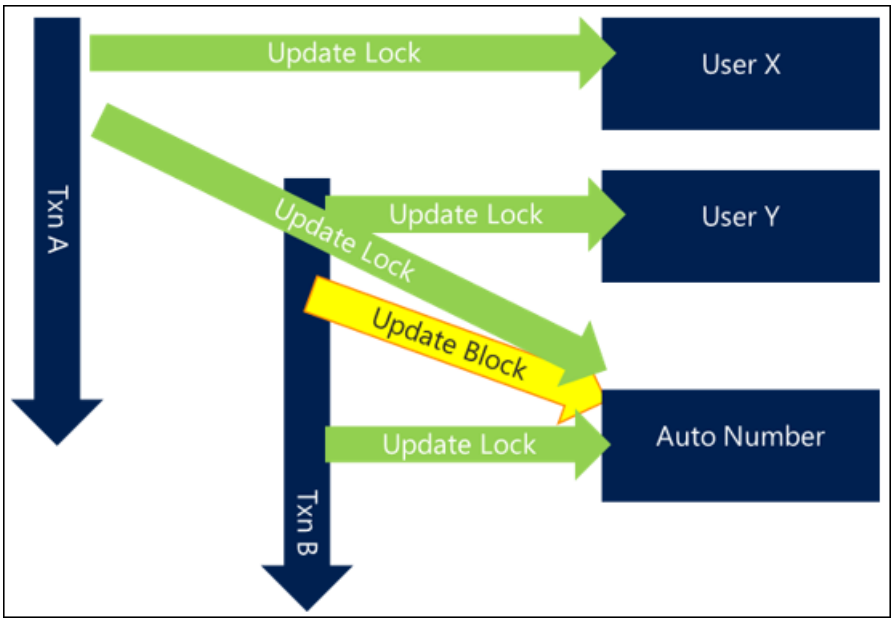
With this approach, although there will still be some blocking on this resource, it reduces the amount of time that resource is locked, and therefore decreases the likelihood and time in which other requests are blocked while waiting for the resource.



Reduce length of transactions

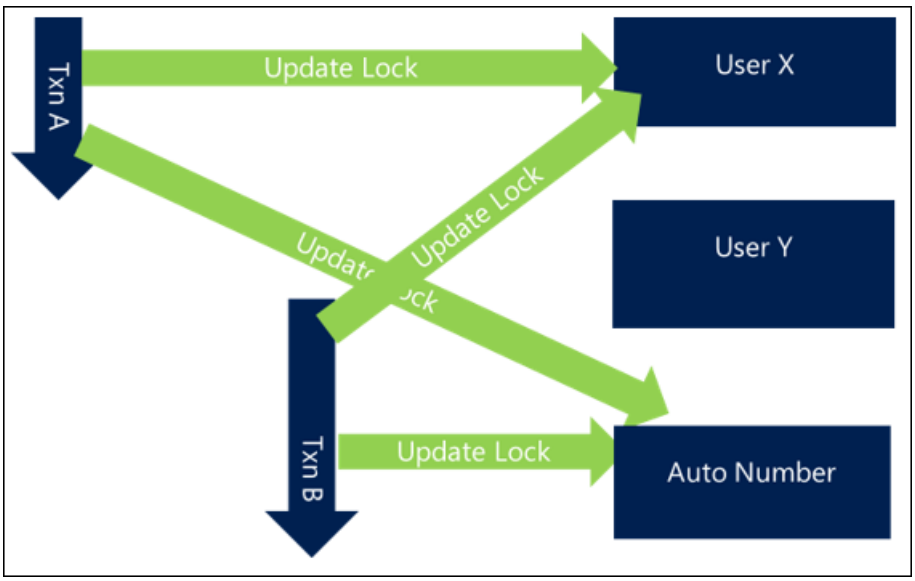
In a similar way, a lock only becomes a blocking issue if two processes need access to the same resource at the same time. The shorter the transaction that holds a lock, the less likely that two processes, even if they access the same resource, will need it at exactly the same time and cause a collision. The less time the transactions are held, the less likely blocking will become a problem.

In the following example, the same locks are taken but other processing within the transaction means that the overall length of the transaction is extended, leading to overlapping requests for the same resources. This means that blocking occurs and each request is slower overall.



By shortening the overall length of the transaction, the first transaction completes and releases its locks before the second request even starts, meaning there is no blocking and both transactions complete efficiently.

Other activities within a request that extend the life of a transaction can increase the chance of blocking, particularly when there are multiple overlapping requests, and can lead to a significantly slower system.



There are a number of ways that the transaction length can be reduced.

Optimize requests

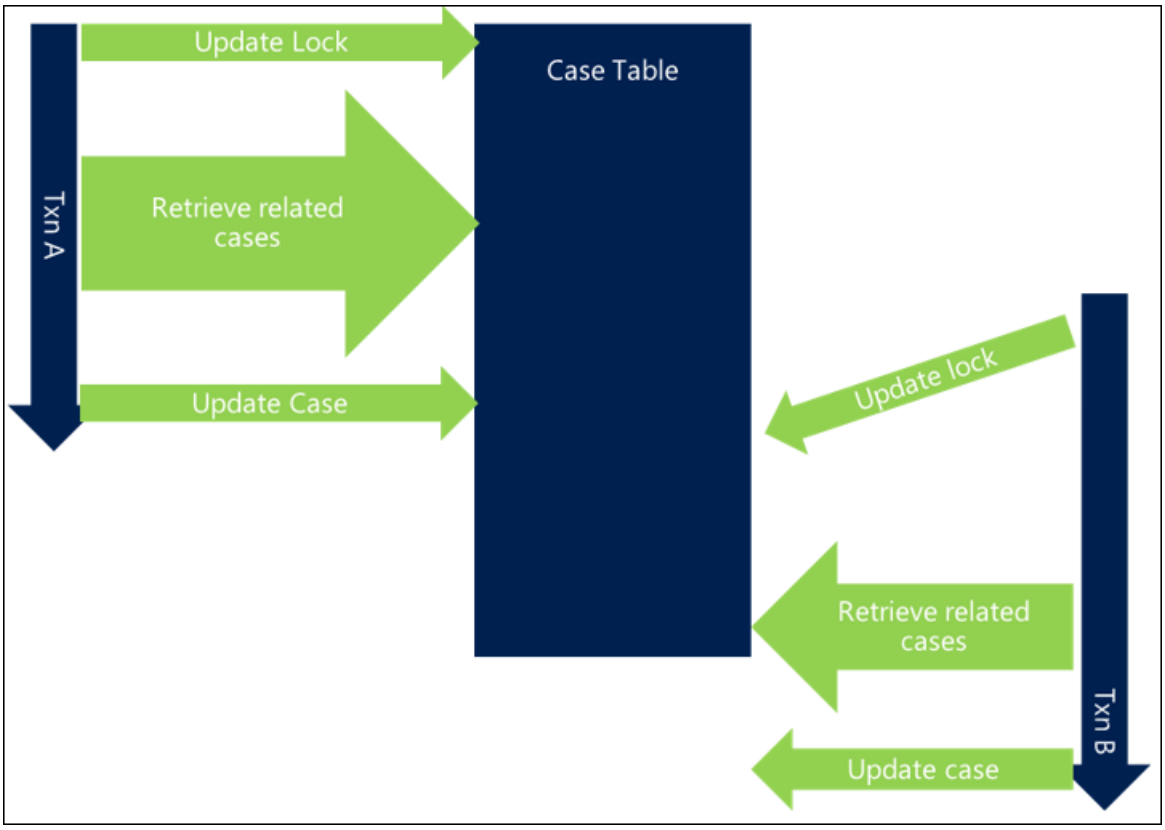
Each transaction is made up of a series of database requests. If each request is made as efficiently as possible, this reduces the overall length of a transaction and reduces the likelihood of collision.

Review each query you make to determine whether:

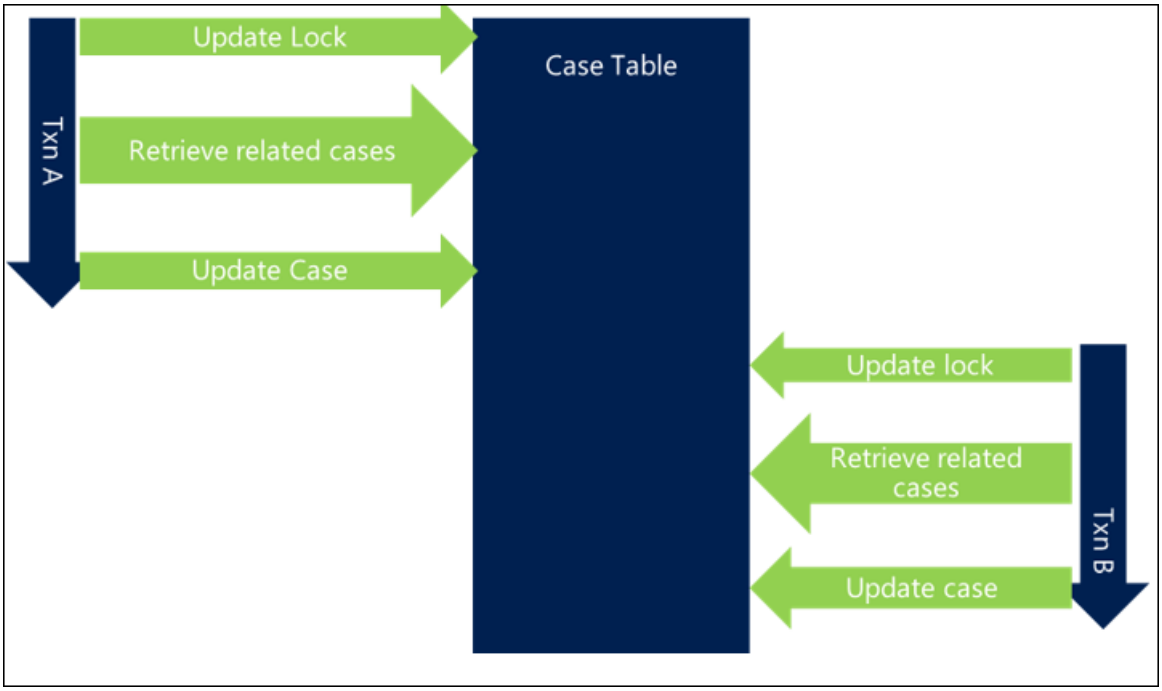
- Your query only asks for what it needs, for example, columns, records, or entity types.
  - This maximizes the chance that an index can be used to efficiently service the query
  - It reduces the number of tables and resources that need to be accessed, reducing the overhead on other resources in the database server and reducing the query time
  - It avoids potential blocking on resources that you don't need, particularly where a join to another table is asked for but could be avoided or is unnecessary
- An index is in place to assist the query, you are querying in an efficient way, and an index seek rather than scan is taking place

  It's worth noting that introducing an index doesn't avoid locking on create/update of records in the underlying table. The entries in the indexes are also locked when the related record is updated as the index itself is subject to change. The existence of indexes doesn't avoid this problem completely.

In the following example, the retrieval of related cases isn't optimized and adds to the overall transaction length, introducing blocking between threads.



By optimizing the query, there's less time spent performing it, and the chance of collision is lower, thereby reducing blocking.



Making sure that the database server can process your query as efficiently as possible can significantly decrease the overall time of your transactions and reduce the potential for blocking.

Reduce chain of events

As was shown in earlier examples, the consequences of long chains of related events can have a material impact on the overall transaction time and therefore the potential for blocking arises. This is particularly the case when triggering synchronous plug-ins and workflows, which then trigger other actions, and they in turn trigger further synchronous plug-ins and workflows.

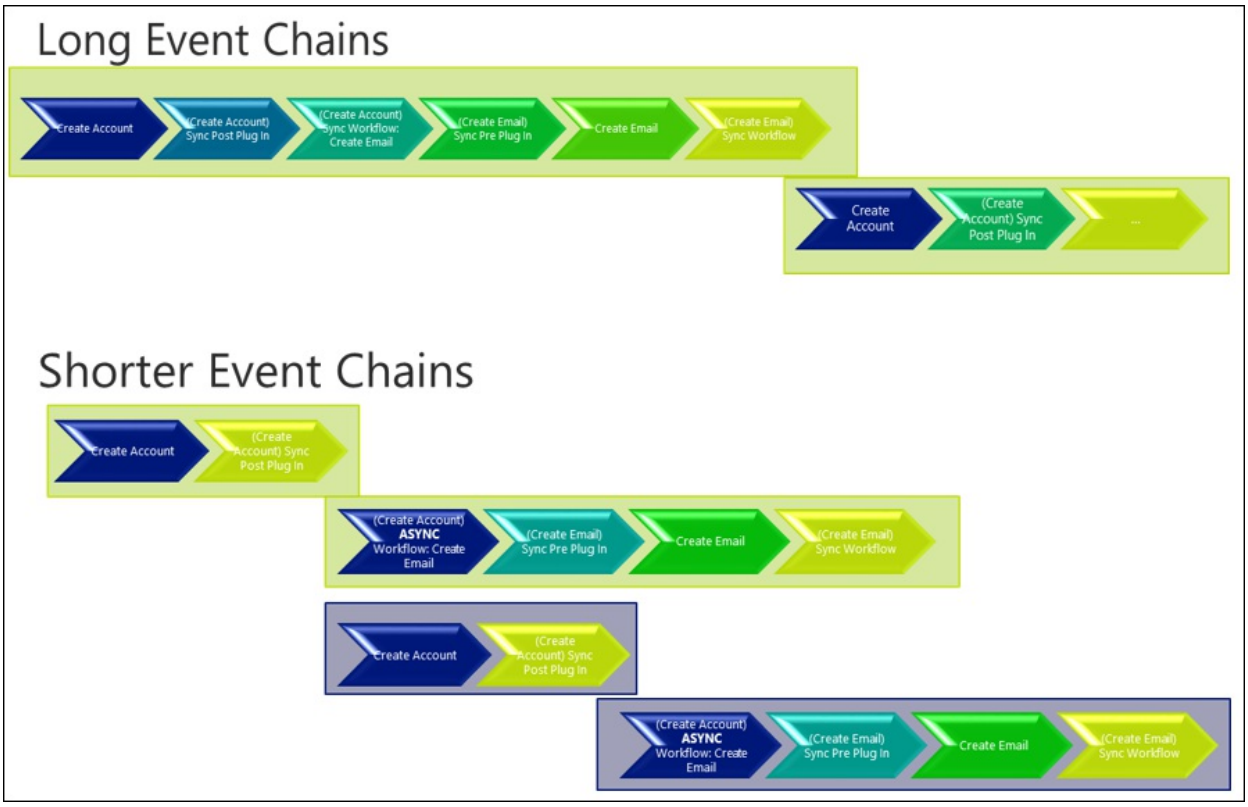
Carefully reviewing and designing an implementation to avoid long chains of events occurring synchronously

can be beneficial in reducing the overall length of a transaction. This allows any locks that are taken to be released more quickly and reduce the potential for blocking.

It also reduces the likelihood of secondary locks becoming a major concern. In the auto numbering example on account creation, the primary issue initially is access to the auto numbering table, but when many different actions are performed in one sequence, a secondary blockage, such as updates to a related user record, may start to surface also. Once multiple contested resources are involved, avoiding blocking becomes even harder.

Considering whether some activities need to be synchronous or asynchronous can mean the same activities are achieved but have less initial impact. Particularly for longer running actions or those depending on heavily contested resources, separating them from the main transaction by performing them in an asynchronous action can have significant benefits. This approach will not work if the action needs to complete or fail with the broader platform step, such as updating a police crime report with the next auto number value ensuring a continuous, sequential number scheme is maintained. In those scenarios other approaches to minimize the impact should be taken.

As the following example shows, simply by moving some actions out to an asynchronous process, which means the actions are performed outside of the platform transaction, can mean that the length of the transaction is shorter and the potential for concurrent processing increases.



Avoid multiple updates to the same record

When designing multiple layers of functional activity, while it is good practice to break down the needed actions to logical and easily followed flows of activity, in many cases this would lead to multiple, separate updates to the same record.

In the case handling scenario, first updating a case with a default owner based on the customer it is raised against and then later having a separate process to automatically send communications to that customer and update the last contact date against the case is a perfectly logical thing to do functionally.

The challenge, however, is that this means there are multiple requests to Dataverse to update the same record, which has a number of implications:

- Each request is a separate platform update, adding overall load to the Dataverse server and adding time to

the overall transaction length, increasing the chance of blocking.

- It also means that the case record will be locked from the first action taken on that case, meaning that the lock is held throughout the rest of the transaction. If the case is accessed by multiple parallel processes, that could cause blocking of other activities.

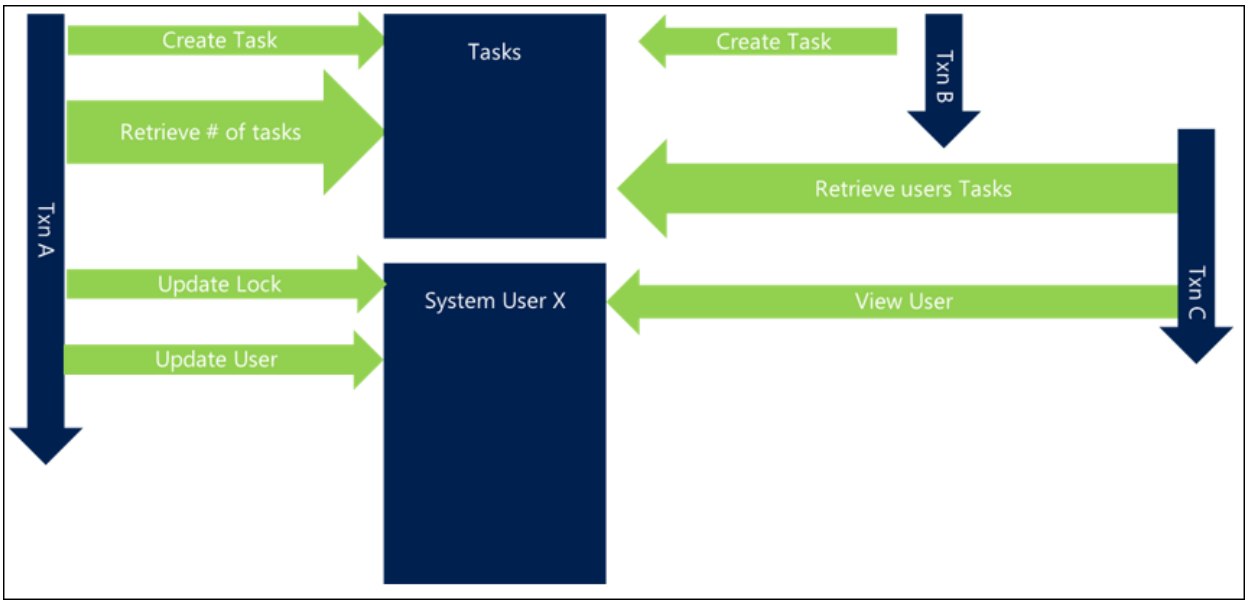
Consolidating updates to the same record to a single update step, and later in the transaction, can have a significant benefit to overall scalability, particularly if the record is heavily contested or accessed by multiple people quickly after creation, for example, as with a case.

Deciding whether to consolidate updates to the same record to a single process would be based on balancing the complexity of implementation against the potential for conflict that separate updates could introduce. But for high volume systems, this can be very beneficial for highly contested resources.

Only update things you need to

While it is important not to reduce the benefit of a Dataverse system by excluding activities that would be beneficial, often requests are made to include customizations that add little business value but drive real technical complexity.

If every time we create a task we also update the user record with the number of tasks they currently have allocated, that could introduce a secondary level of blocking as the user record would also now be heavily contended. It would add another resource that each request may need to block and wait for, despite not necessarily being critical to the action. In that example, consider carefully whether storing the count of tasks against the user is important or if the count can be calculated on demand or stored elsewhere such as using hierarchy and rollup field capabilities in Dataverse natively.



As will be shown later, updating system user records can have negative consequences from a scalability perspective.

Multiple customizations triggered on same event

Triggering multiple actions on the same event can result in a greater chance of collision as by the nature of the requests those actions are likely to interact with the same related objects or parent object.

This is a pattern that should be carefully considered or avoided as it is very easy to overlook conflicts, particularly when different people implement the different processes.

When to use different types of customization

Each type of customization has different implications for use. The following table highlights some common patterns, when each should be considered and used, and when it isn't appropriate for use.

Often a compromise between different behaviors may need to be considered so this gives guidance of some of the common characteristics and scenarios to consider but each scenario needs to be evaluated and the right approach chosen based on all the relevant factors.

PRE/POST STAGE	SYNC/ASYNC	TYPE OF CUSTOMIZATION	WHEN TO USE	WHEN NOT TO USE
Pre Validation	Sync	Plug-in	Short term validation of input values	Long running actions. When creating related items that should be rolled back if later steps fail.
Pre Operation	Sync	Workflow/Plug-in	Short term validation of input values. When creating related items that should be rolled back as part of platform step failure.	Long running actions. When creating an item and the resulting GUID will need to be stored against the item the platform step will create/update.

PRE/POST STAGE	SYNC/ASYNC	TYPE OF CUSTOMIZATION	WHEN TO USE	WHEN NOT TO USE
Post Operation	Sync	Workflow/ Plug-in	Short running actions that naturally follow the platform step and need to be rolled back if later steps fail, for example, creation of a task for the owner of a newly created account. Creation of related items that need the GUID of the created item and that should roll back the platform step in the event of failure	Long running actions. Where failure should not affect the completion of the platform pipeline step.
Not in event pipeline	Async	Workflow/ Plug-in	Medium length actions that would impact on the user experience. Actions that cannot be rolled back anyway in the event of failure. Actions that should not force the rollback of the platform step in the event of failure.	Very long running actions. These shouldn't be managed in Dataverse. Very low cost actions. The overhead of generating async behavior for very low cost actions may be prohibitive; where possible do these synchronously and avoid the overhead of async processing.
N/A Takes context of where it is called from		Custom Actions	Combinations of actions launched from an external source, for example, from a web resource	When always triggered in response to a platform event, use plug-in/workflow in those cases.

Plug-ins/workflows aren't batch processing mechanisms

Long running or volume actions aren't intended to be run from plug-ins or workflows. Dataverse isn't intended to be a compute platform and especially isn't intended as the controller to drive big groups of unrelated updates.

If you have a need to do that, offload and run from a separate service, such as an Azure worker role.

Setting up security

A very common escalation area is scalability of setting up security. This is a costly operation, so when done in volume can always cause challenges if not understood and carefully considered.

Team setup

- Always add users in the same order: avoid deadlocks
- Only update users if they need to be updated: avoid invalidating users' caches unnecessarily

Owner v. access teams

- If users' teams change regularly, be careful about using owner teams heavily; every time they change they invalidate the user's cache in the web server
- Ideally make changes when the user isn't working, reduce impact, such as overnight

Lots of team memberships/ BUs

- Consider carefully scenarios with lots of teams/BUs add to complexity of calculation

Cascading behavior

- Consider cascading sharing, for example, assignment

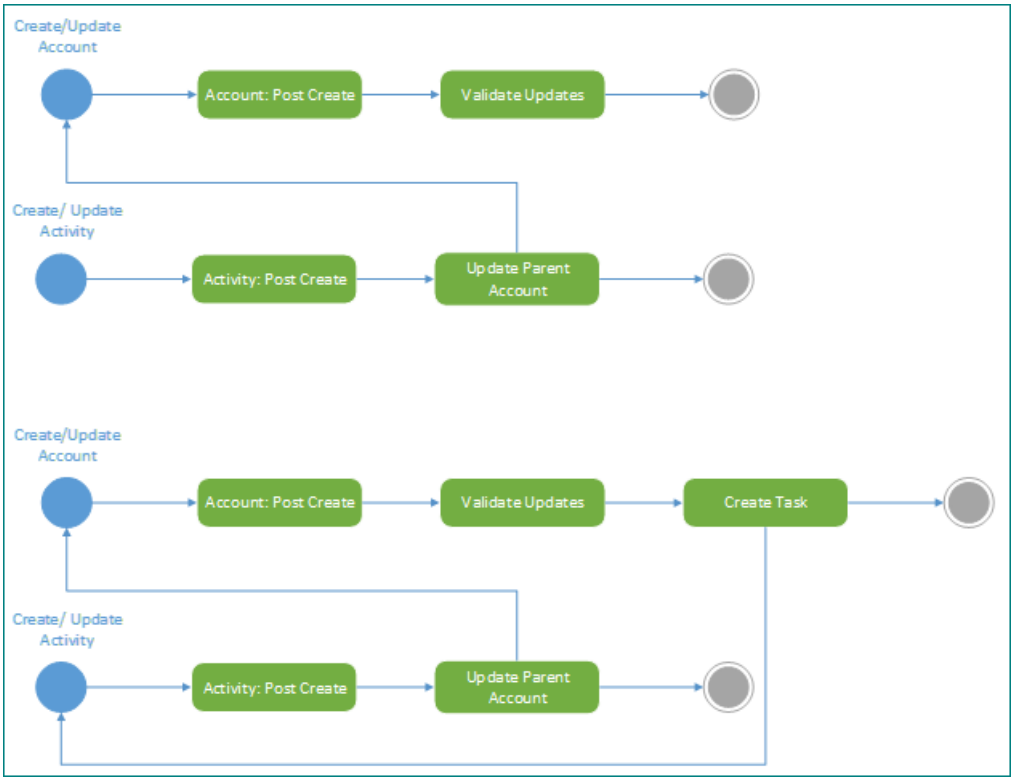
Careful updating of user records

- Don't regularly update system user records unless something fundamental has changed as this forces the user cache to be reloaded and the security privileges to be recalculated, an expensive activity
- Don't use system user to record how many open activities the user has, for example

Diagram related actions

An activity that is very beneficial as a preventative measure, as well as a tool for diagnosing blocking problems, is to diagram related actions triggered in the Dataverse platform. When doing this it helps to highlight both intentional and unintentional dependencies and triggers in the system. If you aren't able to do this for your solution, you might not have a clear picture of what your implementation actually does. Creating such a diagram can expose unintended consequences and is good practice at any time in an implementation.

The following example highlights how initially two processes work perfectly well together but in ongoing maintenance the addition of a new step to create a task can create an unintended loop. Using this documentation technique can highlight this at the design stage and avoid this affecting the system.



Review system captured statistics

There are a number of ways to determine what is happening if the problem occurs outside of the database layer. The first is analysis of plug-in performance. The PluginTypeStatistic Entity can be queried to give an indication of how often the plug-in is running, and statistics on how long it typically takes to run.

When certain errors are occurring, using the server trace files to understand where related problems may be occurring in the platform can also be useful. More information: Use Tracing

Summary

The content in Scalable Customization Design in Dataverse and the subsequent topics Database transactions, Concurrency issues, and this one have described the following concepts with examples and strategies that will help you understand how to design and implement scalable customizations for Dataverse.

Some key things to remember include the following:

Locks/ transactions

- Locks and transactions are essential to a healthy system
- But when used incorrectly, can lead to problems

Platform constraints

- Platform constraints often exhibit in the form of errors
- But rarely is the constraint the cause of the problem
- They're there to protect the platform and other activity from being affected

Design for transaction use

- If implementations are designed with transaction behavior in mind, this can lead to much greater scalability and improved user performance