Coin Management

A key concept when programming on Sui is that of owned objects. Address-owned objects are important in that they allow for highly parallelizable transactions. And they also logically map to assets or resources that someone exclusively owns. Coins are a typical case of owned object usage, with cash being a real-life reference. The owned objects paradigm, however, and particularly as related to coins, is somewhat of a divergence from other blockchains which have a concept of balance. In other words, in other systems, especially account based systems, coins are held in a single location (field) which can be thought of as a balance in a bank account.

Because Sui uses owned objects instead of a balance, it is common to own a number of coins, at times even a significant number of them. Some scenarios necessitate merging some or all of those coins into a single object. At times, merging coins together might even be required because the amount necessary to execute a transaction is more than any single coin the sender owns, thus making merging an inevitable step.

The Sui SDKs (TypeScript and Rust) manage coins on your behalf, removing the overhead of having to deal with coin management manually. The SDKs attempt to merge coins whenever possible and assume that transactions are executed in sequence. That's a reasonable assumption with wallet-based transactions and for common scenarios in general. Sui recommends relying on this feature if you do not have a need for heavy parallel or concurrent execution.

When executing a transaction Sui allows you to provide a number of coins as payment. In other words, the payment can be a vector of coins rather than a single coin. That feature, known as gas smashing, performs merging of coins automatically, and presents the PTBs you write with a single gas coin that can be used for other purposes besides just gas.

Basically, you can provide as many coins as you want (with a max limit defined in the protocol configuration) and have all of them merged (smashed) into the first coin provided as payment. That coin, minus the gas budget, is then available inside the transaction and can be used in any command. If the coin is unused it is returned to the user.

Gas smashing is an important feature - and a key concept to understand - to have for the optimal management of coins. See Gas Smashing for more details.

Gas smashing works well for Coin objects, which is the only coin type that can be used for gas payment.

Any other coin type requires explicit management from users. PTBs offer a mergeCoins command that you can use to combine multiple coins into a single one. And a splitCoins as the complementary operation to break them up.

From a cost perspective, those are very cheap transactions, however they require a user to be aware of their coin distribution and their own needs.

Merging coins, and particularly Coin , into a single coin or a very small number of coins might prove problematic in scenarios where heavy or high concurrency is required.

If you merge all Coin into a single one, you would need to sequentially submit every transaction. The coin - being an owned object - would have to be provided with a version and it would be locked by the system when signing a transaction, effectively making it impossible to use it in any other transaction until the one that locked it was executed. Moreover, an attempt to sign multiple transactions with the same coin might result in equivocation and the coin being unusable and locked until the end of the epoch.

So when you require heavy concurrency, you should first split a coin into as many coins as the number of transactions to execute concurrently. Alternatively, you could provide multiple and different coins (gas smashing) to the different transactions. It is critically important that the set of coins you use in the different transactions has no intersection at all.

The possible pitfalls in dealing with heavy concurrency are many. Concurrency in transaction execution is not the only performance bottleneck. In creating and submitting a transaction, several round trips with a Full node might be required to discover and fetch the right objects, and to dry run a transaction. Those round trips might affect performance significantly.

Concurrency is a difficult subject and is beyond the scope of this documentation. You must take maximum care when dealing with coin management in the face of concurrency, and the right strategy is often tied to the specific scenario, rather than universally available.

SDK usage

The Sui SDKs (TypeScript and Rust) manage coins on your behalf, removing the overhead of having to deal with coin management manually. The SDKs attempt to merge coins whenever possible and assume that transactions are executed in sequence. That's a

reasonable assumption with wallet-based transactions and for common scenarios in general. Sui recommends relying on this feature if you do not have a need for heavy parallel or concurrent execution.

When executing a transaction Sui allows you to provide a number of coins as payment. In other words, the payment can be a vector of coins rather than a single coin. That feature, known as gas smashing, performs merging of coins automatically, and presents the PTBs you write with a single gas coin that can be used for other purposes besides just gas.

Basically, you can provide as many coins as you want (with a max limit defined in the protocol configuration) and have all of them merged (smashed) into the first coin provided as payment. That coin, minus the gas budget, is then available inside the transaction and can be used in any command. If the coin is unused it is returned to the user.

Gas smashing is an important feature - and a key concept to understand - to have for the optimal management of coins. See Gas Smashing for more details.

Gas smashing works well for Coin objects, which is the only coin type that can be used for gas payment.

Any other coin type requires explicit management from users. PTBs offer a mergeCoins command that you can use to combine multiple coins into a single one. And a splitCoins as the complementary operation to break them up.

From a cost perspective, those are very cheap transactions, however they require a user to be aware of their coin distribution and their own needs.

Merging coins, and particularly Coin , into a single coin or a very small number of coins might prove problematic in scenarios where heavy or high concurrency is required.

If you merge all Coin into a single one, you would need to sequentially submit every transaction. The coin - being an owned object - would have to be provided with a version and it would be locked by the system when signing a transaction, effectively making it impossible to use it in any other transaction until the one that locked it was executed. Moreover, an attempt to sign multiple transactions with the same coin might result in equivocation and the coin being unusable and locked until the end of the epoch.

So when you require heavy concurrency, you should first split a coin into as many coins as the number of transactions to execute concurrently. Alternatively, you could provide multiple and different coins (gas smashing) to the different transactions. It is critically important that the set of coins you use in the different transactions has no intersection at all.

The possible pitfalls in dealing with heavy concurrency are many. Concurrency in transaction execution is not the only performance bottleneck. In creating and submitting a transaction, several round trips with a Full node might be required to discover and fetch the right objects, and to dry run a transaction. Those round trips might affect performance significantly.

Concurrency is a difficult subject and is beyond the scope of this documentation. You must take maximum care when dealing with coin management in the face of concurrency, and the right strategy is often tied to the specific scenario, rather than universally available.

Gas Smashing

When executing a transaction Sui allows you to provide a number of coins as payment. In other words, the payment can be a vector of coins rather than a single coin. That feature, known as gas smashing, performs merging of coins automatically, and presents the PTBs you write with a single gas coin that can be used for other purposes besides just gas.

Basically, you can provide as many coins as you want (with a max limit defined in the protocol configuration) and have all of them merged (smashed) into the first coin provided as payment. That coin, minus the gas budget, is then available inside the transaction and can be used in any command. If the coin is unused it is returned to the user.

Gas smashing is an important feature - and a key concept to understand - to have for the optimal management of coins. See Gas Smashing for more details.

Gas smashing works well for Coin objects, which is the only coin type that can be used for gas payment.

Any other coin type requires explicit management from users. PTBs offer a mergeCoins command that you can use to combine multiple coins into a single one. And a splitCoins as the complementary operation to break them up.

From a cost perspective, those are very cheap transactions, however they require a user to be aware of their coin distribution and their own needs.

Merging coins, and particularly Coin , into a single coin or a very small number of coins might prove problematic in scenarios where

heavy or high concurrency is required.

If you merge all Coin into a single one, you would need to sequentially submit every transaction. The coin - being an owned object - would have to be provided with a version and it would be locked by the system when signing a transaction, effectively making it impossible to use it in any other transaction until the one that locked it was executed. Moreover, an attempt to sign multiple transactions with the same coin might result in equivocation and the coin being unusable and locked until the end of the epoch.

So when you require heavy concurrency, you should first split a coin into as many coins as the number of transactions to execute concurrently. Alternatively, you could provide multiple and different coins (gas smashing) to the different transactions. It is critically important that the set of coins you use in the different transactions has no intersection at all.

The possible pitfalls in dealing with heavy concurrency are many. Concurrency in transaction execution is not the only performance bottleneck. In creating and submitting a transaction, several round trips with a Full node might be required to discover and fetch the right objects, and to dry run a transaction. Those round trips might affect performance significantly.

Concurrency is a difficult subject and is beyond the scope of this documentation. You must take maximum care when dealing with coin management in the face of concurrency, and the right strategy is often tied to the specific scenario, rather than universally available.

Generic coins

Gas smashing works well for Coin objects, which is the only coin type that can be used for gas payment.

Any other coin type requires explicit management from users. PTBs offer a mergeCoins command that you can use to combine multiple coins into a single one. And a splitCoins as the complementary operation to break them up.

From a cost perspective, those are very cheap transactions, however they require a user to be aware of their coin distribution and their own needs.

Merging coins, and particularly Coin , into a single coin or a very small number of coins might prove problematic in scenarios where heavy or high concurrency is required.

If you merge all Coin into a single one, you would need to sequentially submit every transaction. The coin - being an owned object - would have to be provided with a version and it would be locked by the system when signing a transaction, effectively making it impossible to use it in any other transaction until the one that locked it was executed. Moreover, an attempt to sign multiple transactions with the same coin might result in equivocation and the coin being unusable and locked until the end of the epoch.

So when you require heavy concurrency, you should first split a coin into as many coins as the number of transactions to execute concurrently. Alternatively, you could provide multiple and different coins (gas smashing) to the different transactions. It is critically important that the set of coins you use in the different transactions has no intersection at all.

The possible pitfalls in dealing with heavy concurrency are many. Concurrency in transaction execution is not the only performance bottleneck. In creating and submitting a transaction, several round trips with a Full node might be required to discover and fetch the right objects, and to dry run a transaction. Those round trips might affect performance significantly.

Concurrency is a difficult subject and is beyond the scope of this documentation. You must take maximum care when dealing with coin management in the face of concurrency, and the right strategy is often tied to the specific scenario, rather than universally available.

Concurrency

Merging coins, and particularly Coin , into a single coin or a very small number of coins might prove problematic in scenarios where heavy or high concurrency is required.

If you merge all Coin into a single one, you would need to sequentially submit every transaction. The coin - being an owned object - would have to be provided with a version and it would be locked by the system when signing a transaction, effectively making it impossible to use it in any other transaction until the one that locked it was executed. Moreover, an attempt to sign multiple transactions with the same coin might result in equivocation and the coin being unusable and locked until the end of the epoch.

So when you require heavy concurrency, you should first split a coin into as many coins as the number of transactions to execute concurrently. Alternatively, you could provide multiple and different coins (gas smashing) to the different transactions. It is critically important that the set of coins you use in the different transactions has no intersection at all.

The possible pitfalls in dealing with heavy concurrency are many. Concurrency in transaction execution is not the only performance bottleneck. In creating and submitting a transaction, several round trips with a Full node might be required to discover and fetch the right objects, and to dry run a transaction. Those round trips might affect performance significantly.

Concurrency is a difficult subject and is beyond the scope of this documentation. You must take maximum care when dealing with coin management in the face of concurrency, and the right strategy is often tied to the specific scenario, rather than universally available.