Delta encodings
business logic ◆ database

Heinrich Apfelmus

bobkonf, 17 March 2023

Code

Image by @rahjada2.0

Code

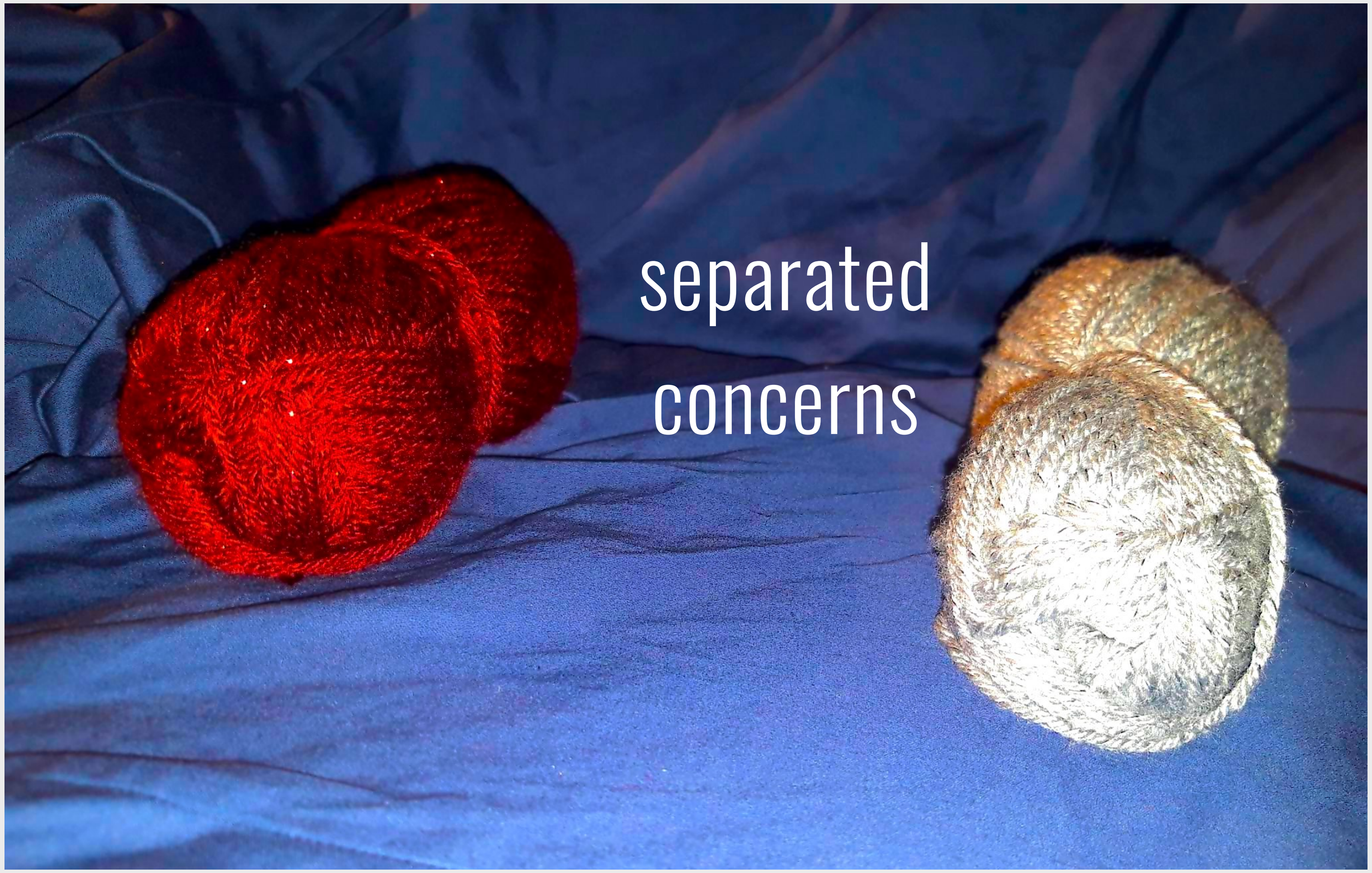


Image by @rahjada2.0

Separate concerns!

Concerns

business logic

```
data Set T

list   :: Set T → [T]
filter :: Set T → Set T
```

Concerns

business logic

```
data Set T

list   :: Set T → [T]
filter :: Set T → Set T
```

database operations

table	

Concerns

business logic

```
data Set T

list   :: Set T → [T]
filter :: Set T → Set T
```

database operations

```
load   :: IO (Set T)
write  :: Set T → IO ()
```

Concerns

business logic

```
data Set T

list   :: Set T → [T]
filter :: Set T → Set T
```

database operations

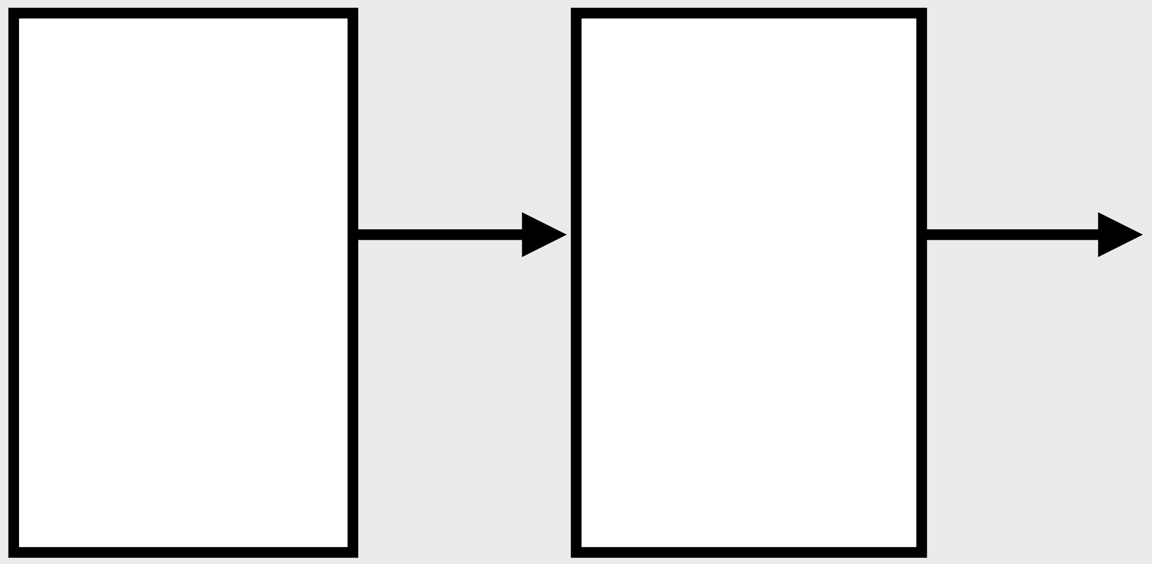
```
load   :: IO (Set T)
write  :: Set T → IO ()

update :: ???
```

Case Study

Submit transactions to a blockchain

Business logic

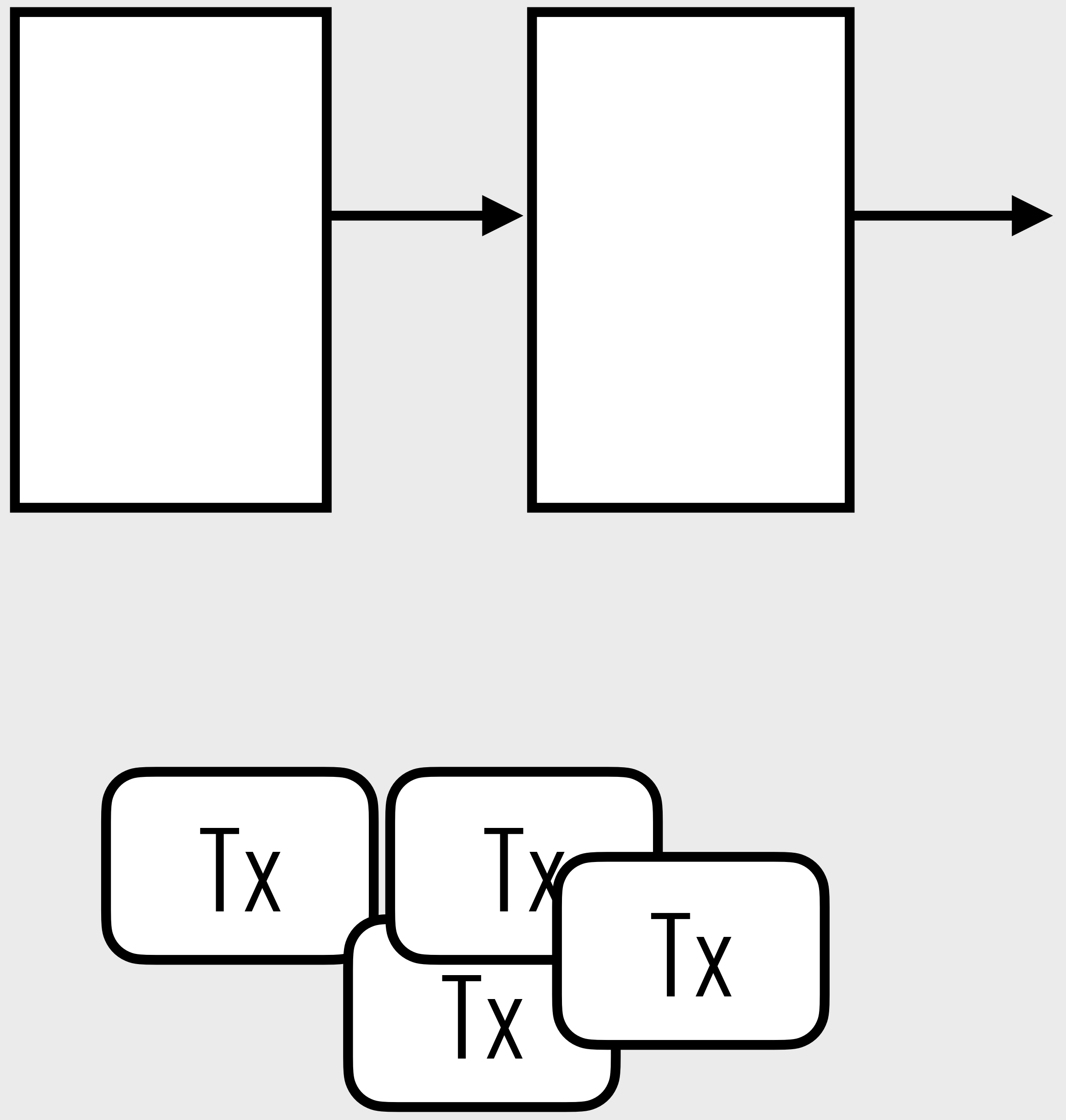
blockchain



Business logic

blockchain

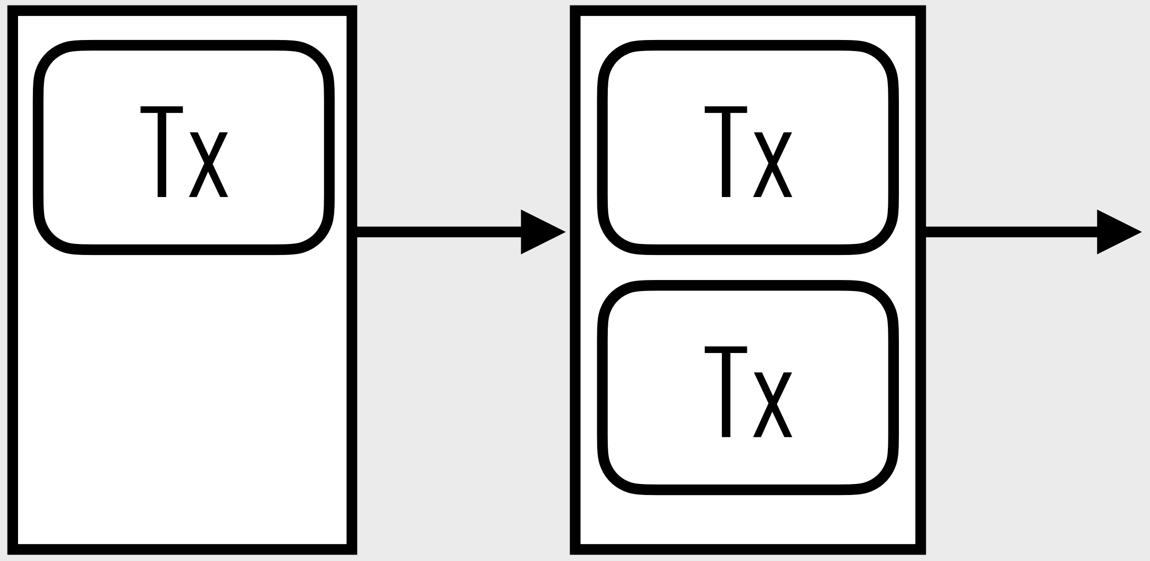
transactions



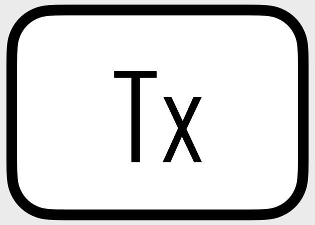
submitted

Business logic

blockchain



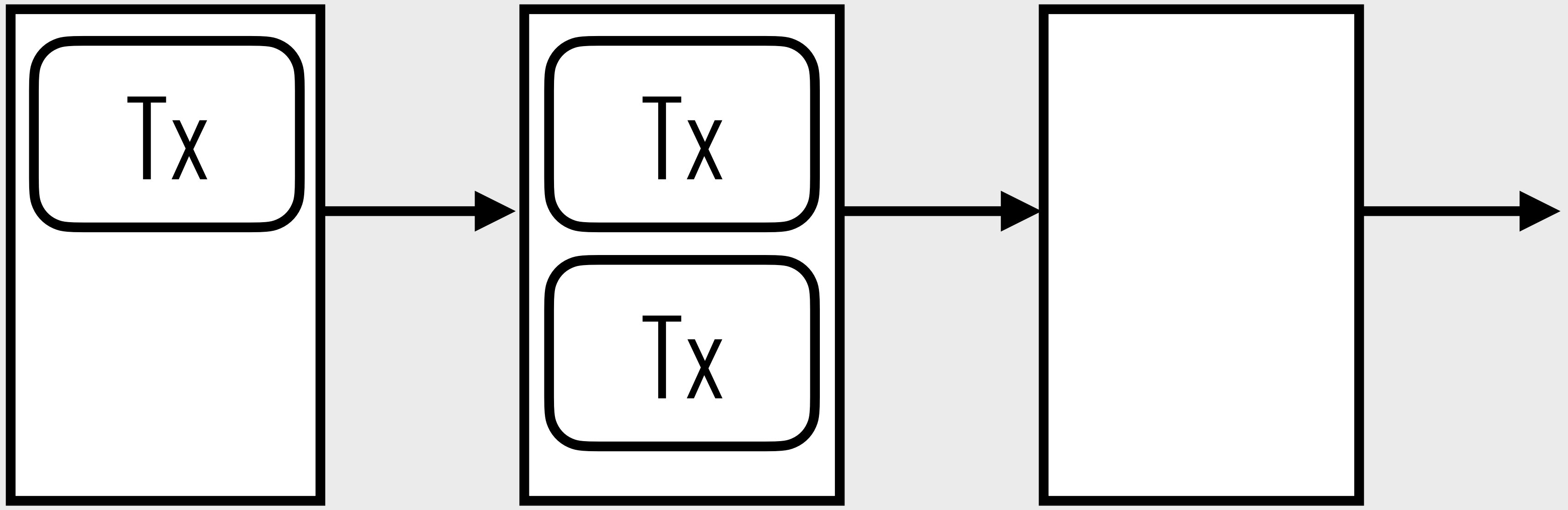
transactions



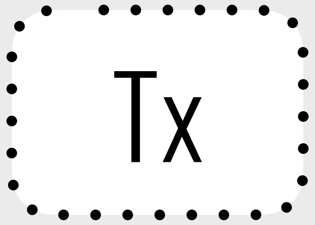
submitted

Business logic

blockchain

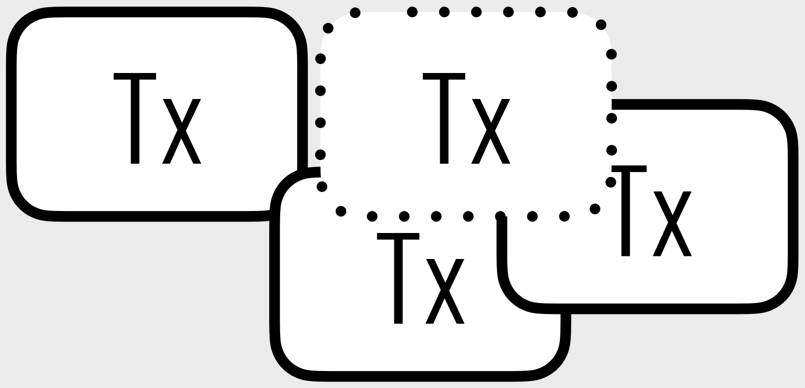


transactions



expired

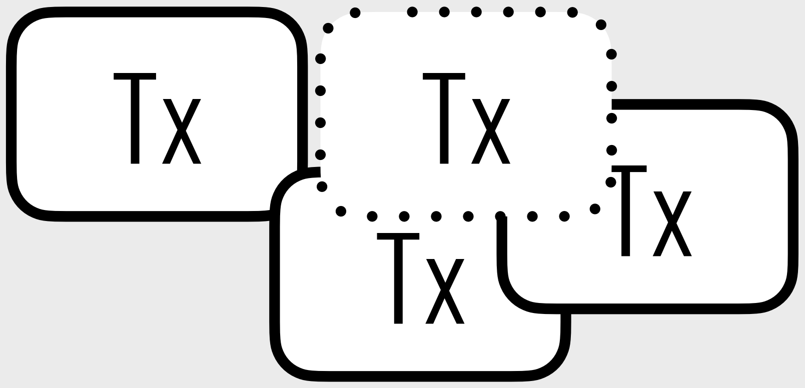
Business logic



submitted & expired

Business logic

```
data Set Tx

list   :: Set Tx → [Tx]
expire :: Set Tx → Set Tx
```



submitted & expired

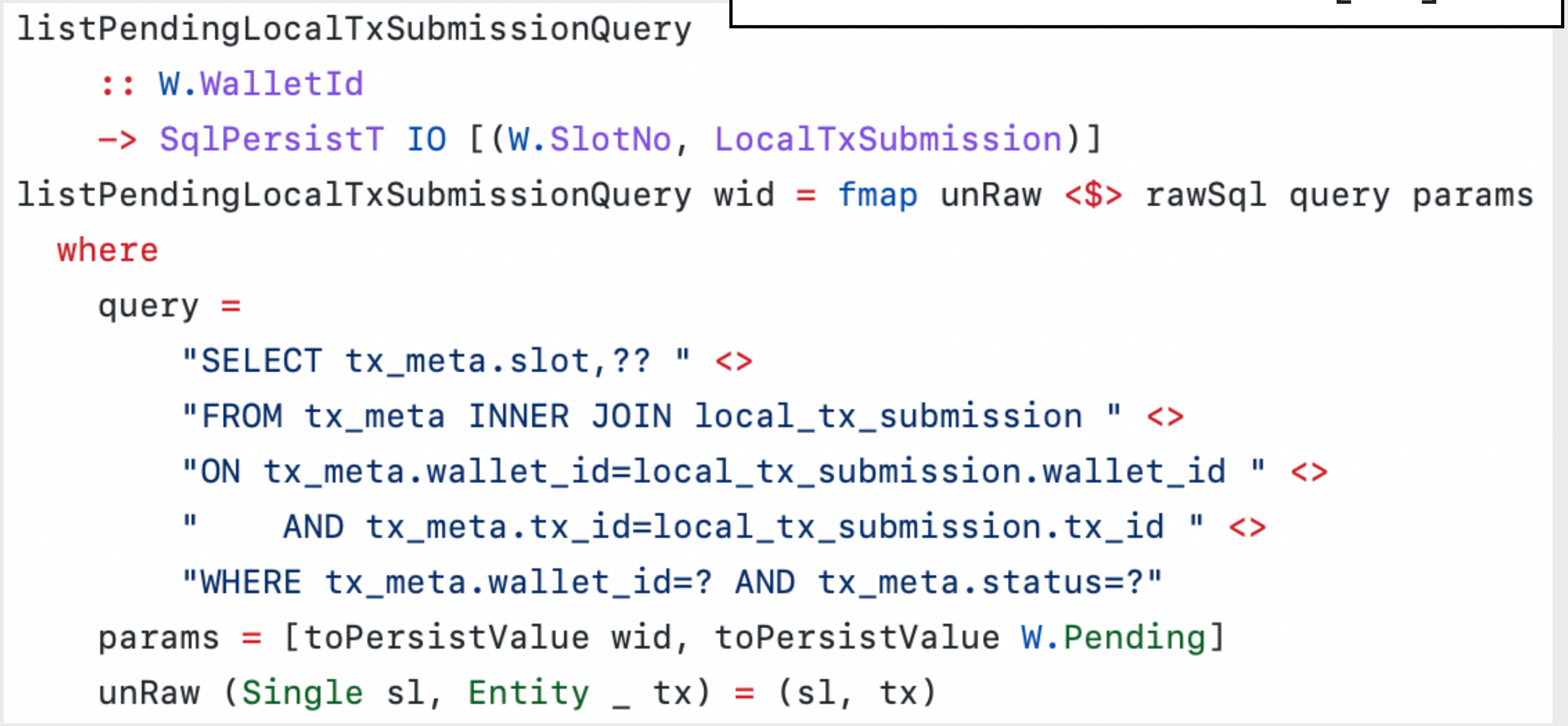
Monolith

cardano-wallet

~350	.hs files
~160k	lines of code and tests

Monolith

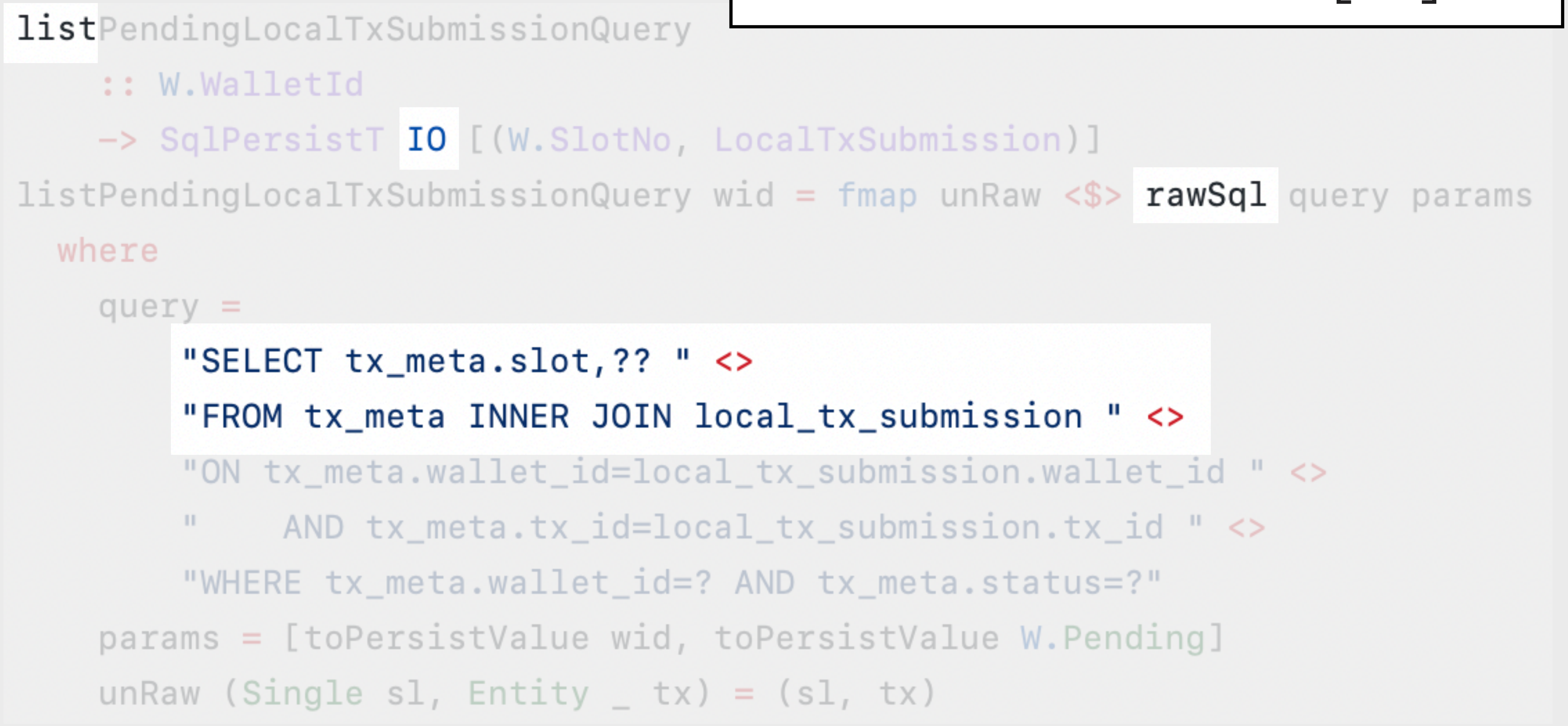
```
list    :: Set Tx → [Tx]
```



```haskell
listPendingLocalTxSubmissionQuery
    :: W.WalletId
    -> SqlPersistT IO [(W.SlotNo, LocalTxSubmission)]
listPendingLocalTxSubmissionQuery wid = fmap unRaw <$> rawSql query params
  where
    query =
        "SELECT tx_meta.slot,?? " <>
        "FROM tx_meta INNER JOIN local_tx_submission " <>
        "ON tx_meta.wallet_id=local_tx_submission.wallet_id " <>
        "    AND tx_meta.tx_id=local_tx_submission.tx_id " <>
        "WHERE tx_meta.wallet_id=? AND tx_meta.status=?"
    params = [toPersistValue wid, toPersistValue W.Pending]
    unRaw (Single sl, Entity _ tx) = (sl, tx)
```

Monolith

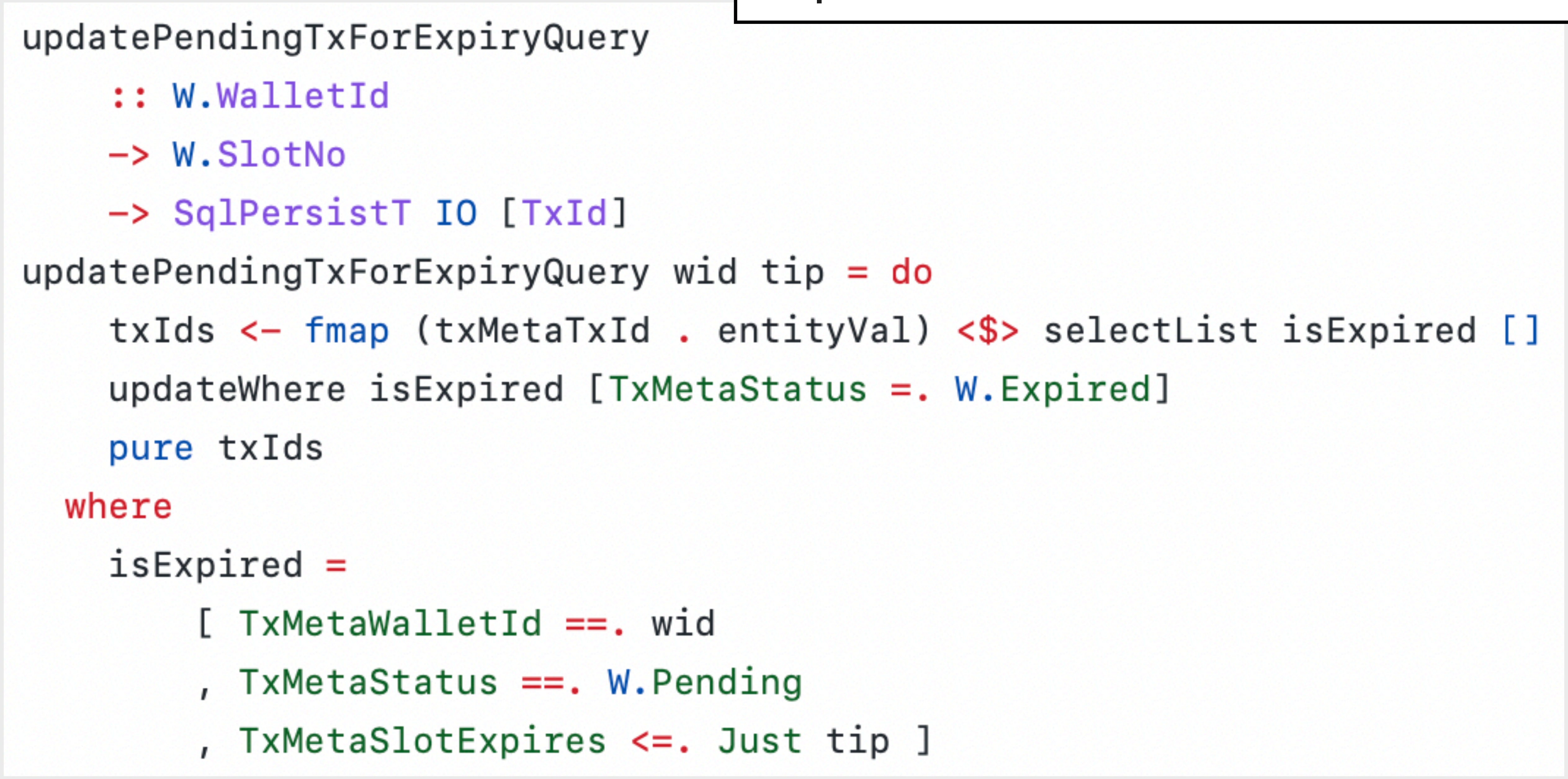
```
list    :: Set Tx → [Tx]
```



```
listPendingLocalTxSubmissionQuery
    :: W.WalletId
    -> SqlPersistT IO [(W.SlotNo, LocalTxSubmission)]
listPendingLocalTxSubmissionQuery wid = fmap unRaw <$> rawSql query params
  where
    query =
        "SELECT tx_meta.slot,?? " <>
        "FROM tx_meta INNER JOIN local_tx_submission " <>
        "ON tx_meta.wallet_id=local_tx_submission.wallet_id " <>
        "    AND tx_meta.tx_id=local_tx_submission.tx_id " <>
        "WHERE tx_meta.wallet_id=? AND tx_meta.status=?"
    params = [toPersistValue wid, toPersistValue W.Pending]
    unRaw (Single sl, Entity _ tx) = (sl, tx)
```

Monolith

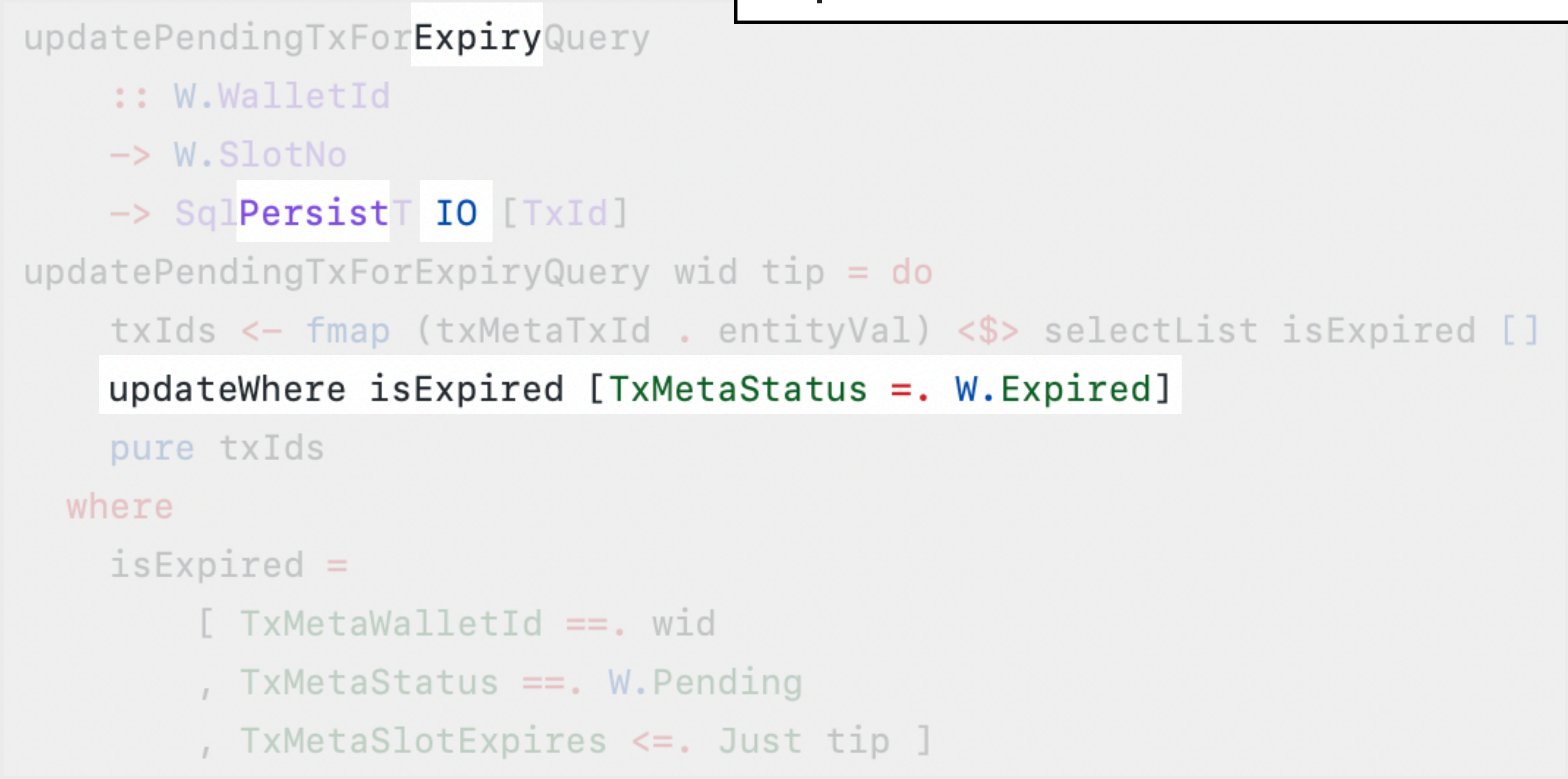
```
expire :: Set Tx → Set Tx
```



```
updatePendingTxForExpiryQuery
    :: W.WalletId
    -> W.SlotNo
    -> SqlPersistT IO [TxId]
updatePendingTxForExpiryQuery wid tip = do
    txIds <- fmap (txMetaTxId . entityVal) <$> selectList isExpired []
    updateWhere isExpired [TxMetaStatus =. W.Expired]
    pure txIds
  where
    isExpired =
        [ TxMetaWalletId ==. wid
        , TxMetaStatus ==. W.Pending
        , TxMetaSlotExpires <=. Just tip ]
```

Monolith

```
expire :: Set Tx → Set Tx
```



```
updatePendingTxForExpiryQuery
    :: W.WalletId
    -> W.SlotNo
    -> SqlPersistT IO [TxId]
updatePendingTxForExpiryQuery wid tip = do
    txIds <- fmap (txMetaTxId . entityVal) <$> selectList isExpired []
    updateWhere isExpired [TxMetaStatus =. W.Expired]
    pure txIds
  where
    isExpired =
        [ TxMetaWalletId ==. wid
        , TxMetaStatus ==. W.Pending
        , TxMetaSlotExpires <=. Just tip ]
```

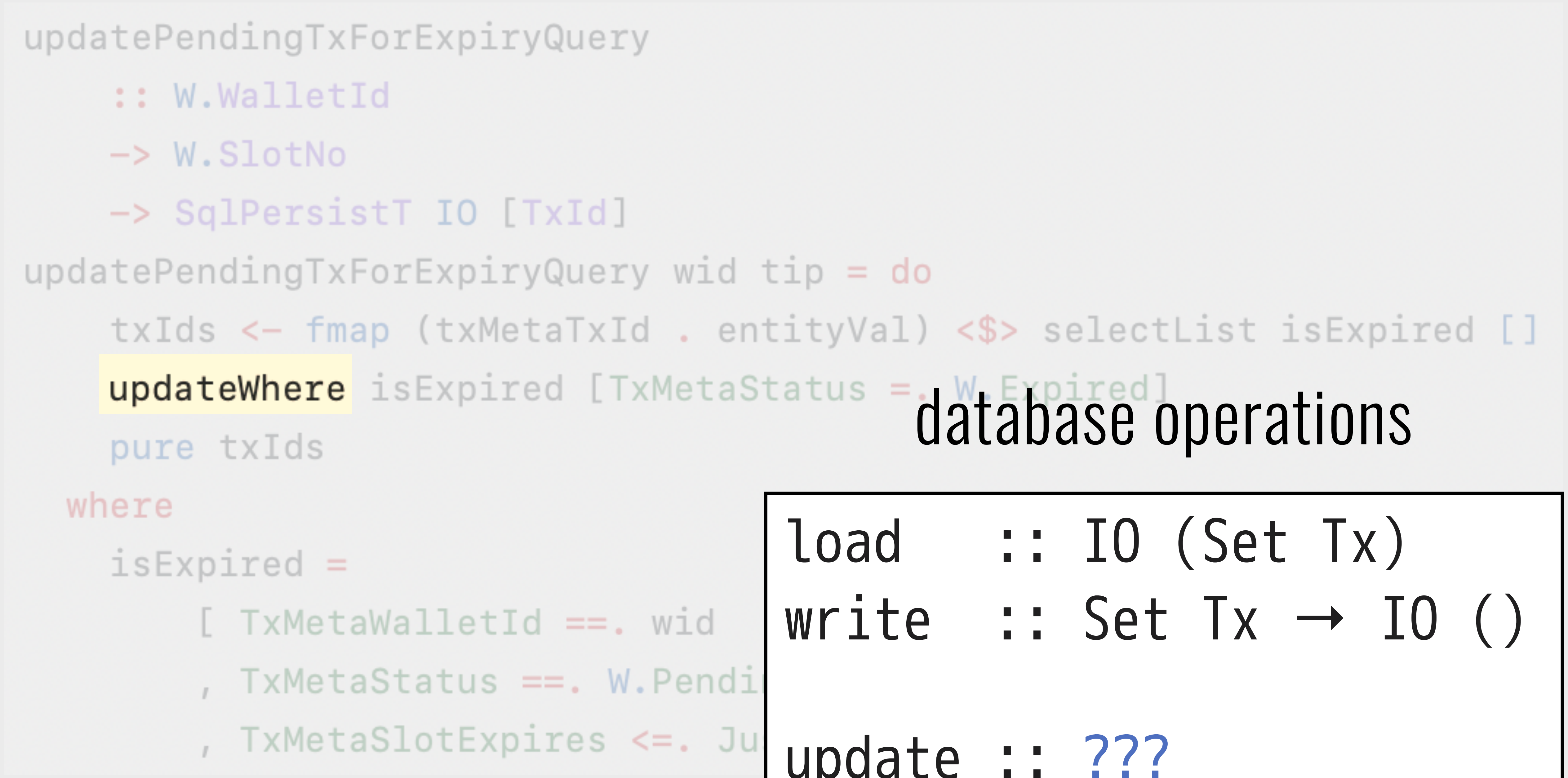
Monolith

```
updatePendingTxForExpiryQuery
    :: W.WalletId
    -> W.SlotNo
    -> SqlPersistT IO [TxId]
updatePendingTxForExpiryQuery wid tip = do
    txIds <- fmap (txMetaTxId . entityVal) <$> selectList isExpired []
    updateWhere isExpired [TxMetaStatus =. W.Expired]
    pure txIds
  where
    isExpired =
        [ TxMetaWalletId ==. wid
        , TxMetaStatus ==. W.Pending
        , TxMetaSlotExpires <=. Just tip ]
```

database operations

```
load    :: IO (Set Tx)
write   :: Set Tx → IO ()
```

Monolith



```
updatePendingTxForExpiryQuery
    :: W.WalletId
    -> W.SlotNo
    -> SqlPersistT IO [TxId]
updatePendingTxForExpiryQuery wid tip = do
    txIds <- fmap (txMetaTxId . entityVal) <$> selectList isExpired []
    updateWhere isExpired [TxMetaStatus =. W.Expired]
    pure txIds
  where
    isExpired =
        [ TxMetaWalletId ==. wid
        , TxMetaStatus ==. W.Pendi
        , TxMetaSlotExpires <=. Ju
```

database operations

```
load    :: IO (Set Tx)
write   :: Set Tx → IO ()

update  :: ???
```

Separate concerns

updateWhere

business logic

```
expire :: Set Tx
       → DeltaSet Tx
```

database operations

```
update :: DeltaSet Tx
       → IO ()
```

Separate concerns

◆ business logic — support efficient updates

updateWhere

business logic

database operations

```
expire :: Set Tx
       → DeltaSet Tx
```

```
update :: DeltaSet Tx
       → IO ()
```

Delta encodings

Delta encodings

business logic

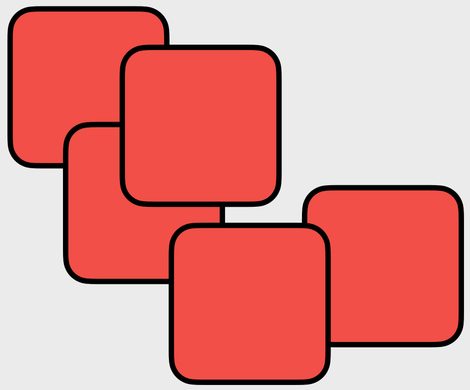
```
expire :: Set Tx
       → DeltaSet Tx
```

database operations

```
update :: DeltaSet Tx
       → IO ()
```

DeltaSet1

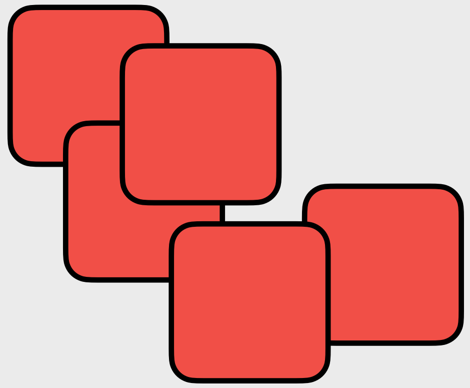
Set a



DeltaSet1

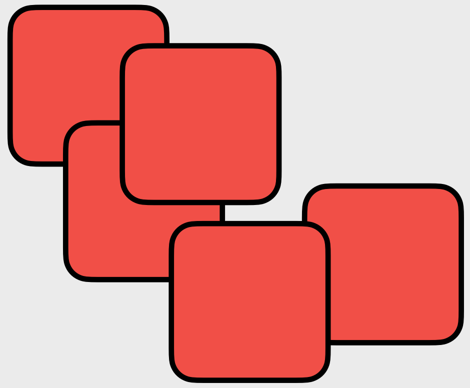
```
data DeltaSet1 a
    = Insert a
    | Delete a
```

Set a



DeltaSet1

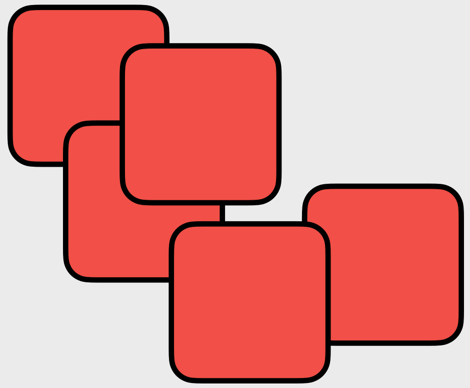
```
data DeltaSet1 a
    = Insert a
    | Delete a
```

Set a 

```
apply :: DeltaSet1 a → (Set a → Set a)
apply (Insert a) = Data.Set.insert a
apply (Delete a) = Data.Set.delete a
```

class Delta

```
data DeltaSet1 a
    = Insert a
    | Delete a
```

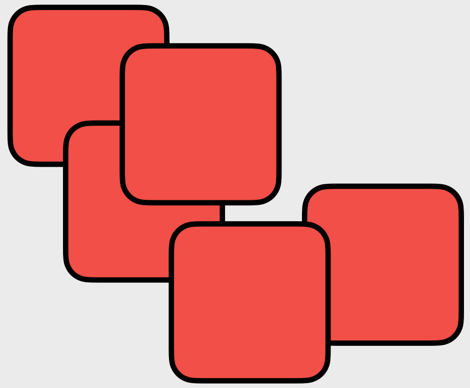
Set a 

```
class Delta da where
    type Base :: Type → Type
    apply :: da → (Base a → Base a)
```

class Delta

```
data DeltaSet1 a
    = Insert a
    | Delete a
```

Set a



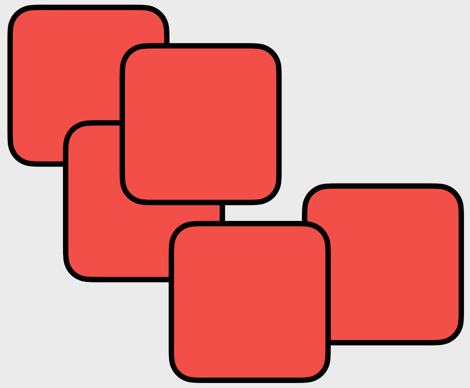
```
class Delta da where
    type Base :: Type → Type
    apply :: da → (Base a → Base a)
```

```
instance Delta (DeltaSet1 a) where
    type Base (DeltaSet1 a) = Set a
    apply (Insert a) = Data.Set.insert a
    apply (Delete a) = Data.Set.delete a
```

List of Deltas

```
data DeltaSet a
  = [DeltaSet1 a]
```

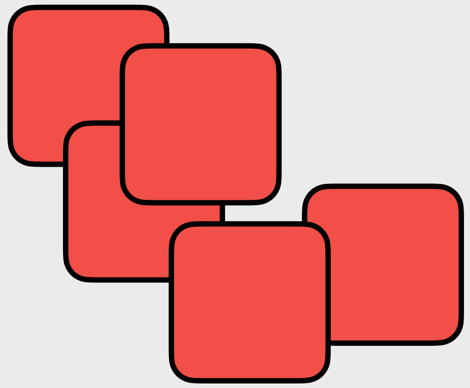
Set a



List of Deltas

```
data DeltaSet a
  = [DeltaSet1 a]
```

Set a

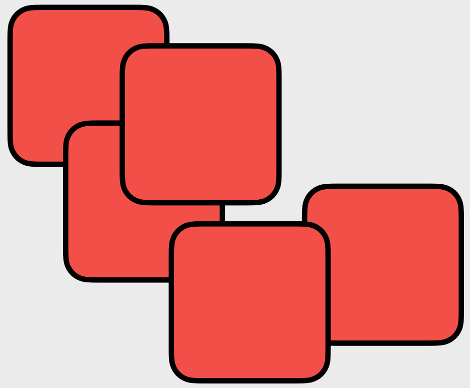


- one Base type, many Delta

List of Deltas

```
data DeltaSet a
  = [DeltaSet1 a]
```

Set a



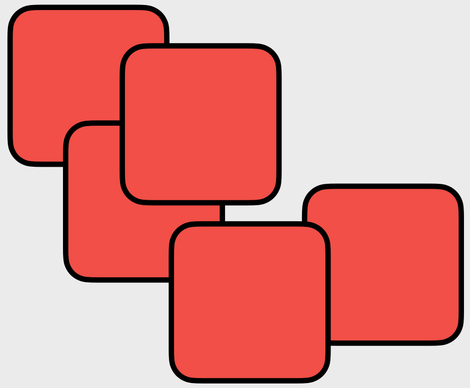
- one Base type, many Delta

```
instance Delta da => Delta [da] where
    type Base [da] = Base da
    apply das a = foldr apply a das
```

List of Deltas

```
data DeltaSet a
  = [DeltaSet1 a]
```

Set a



- one Base type, many Delta

```
instance Delta da => Delta [da] where
    type Base [da] = Base da
    apply das a = foldr apply a das
```

```
apply (xs ++ ys) = apply xs . apply ys
```

Example

```
expire :: Time → Set Tx → DeltaSet Tx
```

Example

```
expire :: Time → Set Tx → DeltaSet Tx
```

```
expire now xs =
    [Delete y | y ← filter isExpired xs]
 where
    isExpired x = expiryDate x < now
```

Example

```
expire :: Time → Set Tx → DeltaSet Tx
```

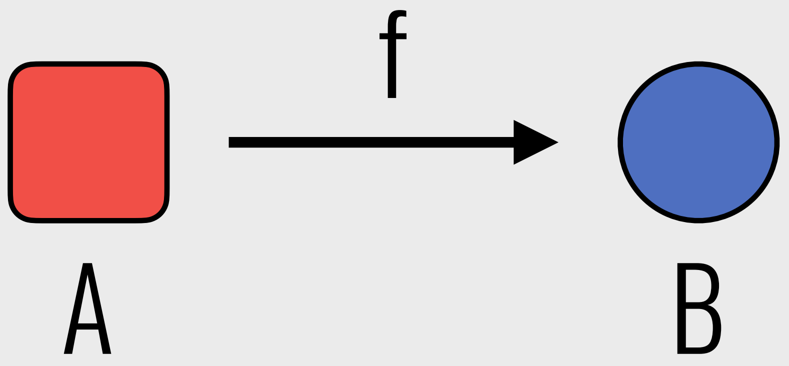
```
expire now xs =
    [Delete y | y ← filter isExpired xs]
 where
    isExpired x = expiryDate x < now
```

Example

```
expire :: Time → Set Tx → DeltaSet Tx
```

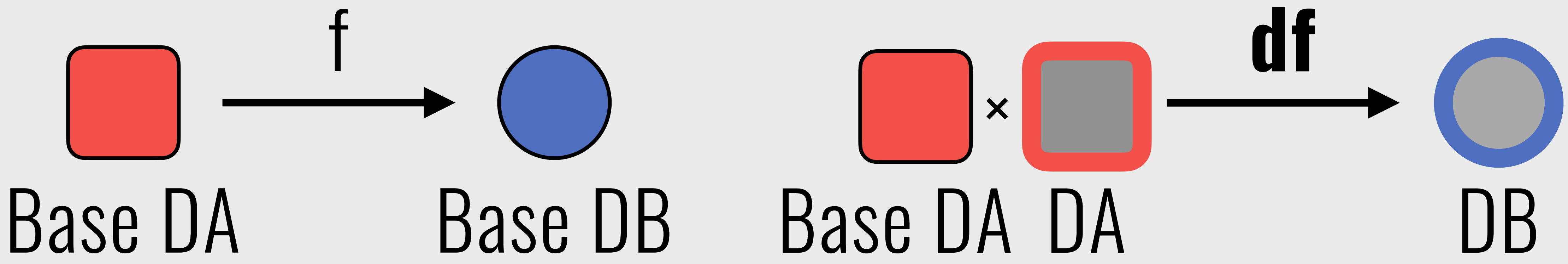
```
expire now xs =
    [Delete y | y ← filter isExpired xs]
 where
    isExpired x = expiryDate x < now
```

Category: Haskell



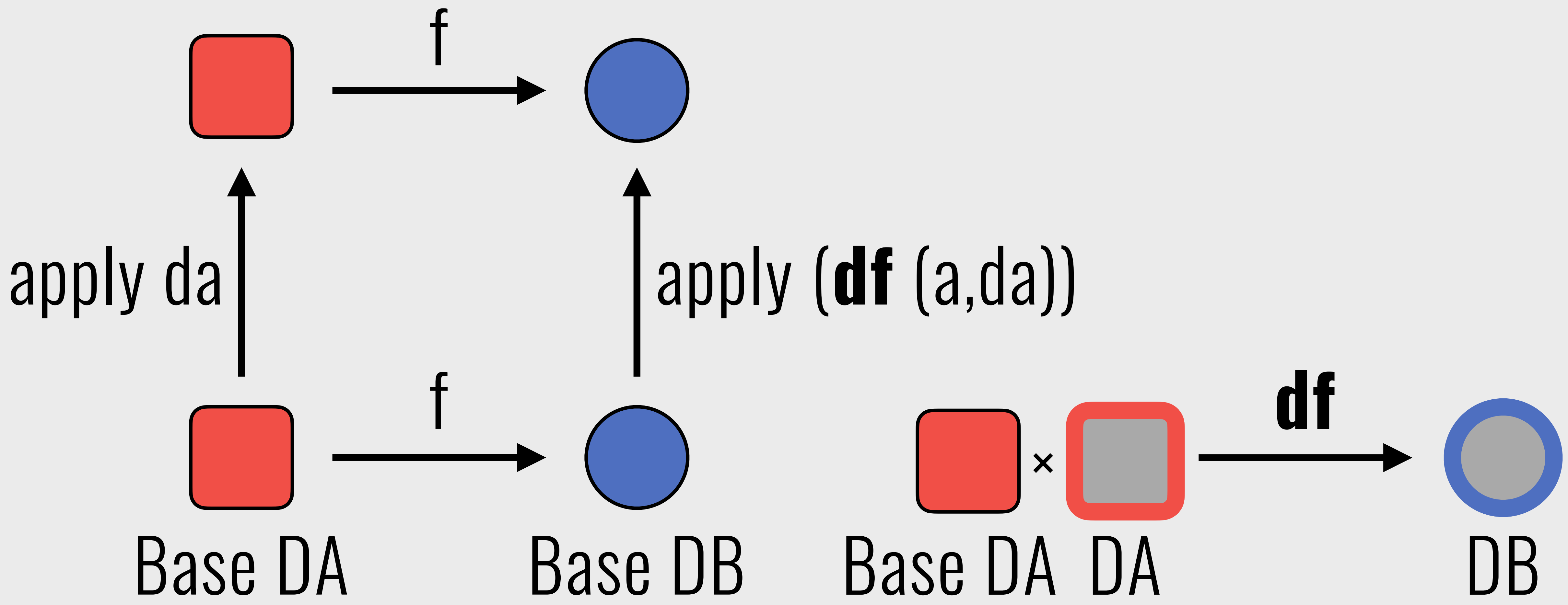
Morphism = function f

Category: Delta



Morphism = (f, **df**)

Category: Delta



Morphism = (f, **df**)

Database

Database

business logic

```
expire :: Set Tx
       → DeltaSet Tx
```

database operations

```
update :: DeltaSet Tx
       → IO ()
```

Database

database operations

```
load   :: IO (Set Tx)
write  :: Set Tx → IO ()

update :: DeltaSet Tx
       → IO ()
```

Store

“facility for storing one value of type (Base delta)”

```
data Store m delta =
   Store
   { load   :: m (Base delta)
   , write  :: Base delta → m ()
   , update :: delta → m ()
   }
```

Store

“facility for storing one value of type (Base delta)”

```
data Store m delta =
   Store
   { load   :: m (Base delta)
   , write  :: Base delta → m ()
   , update :: delta → m ()
   }
```

example

```
delta = DeltaSet Tx
Base delta = Set Tx
```

Store

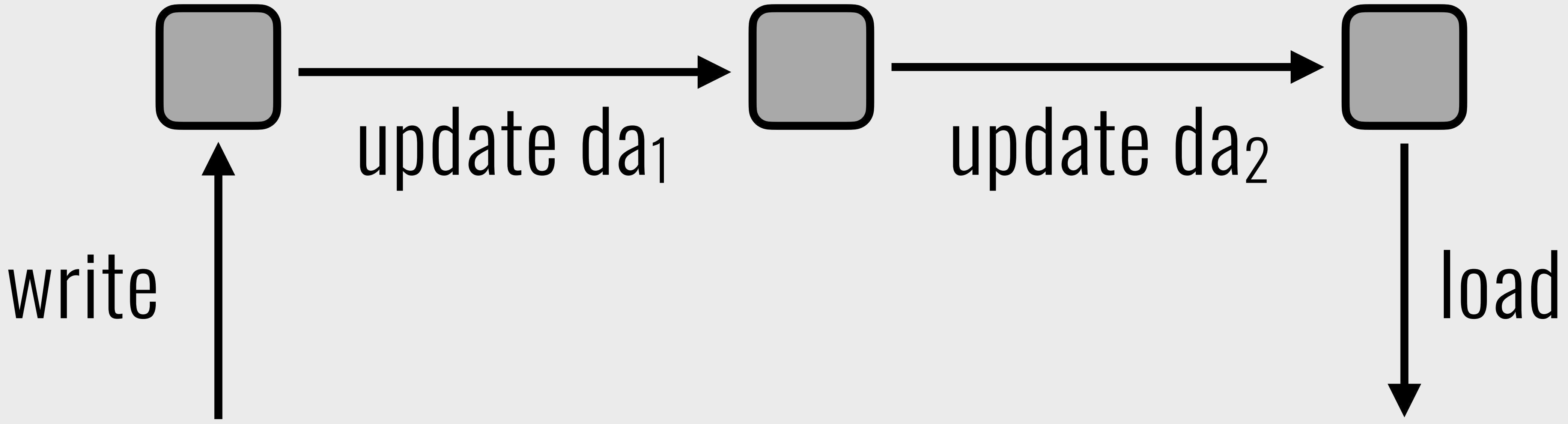
“facility for storing one value of type (Base delta)”

```
data Store m delta
```

- first-class
- stored value :: type

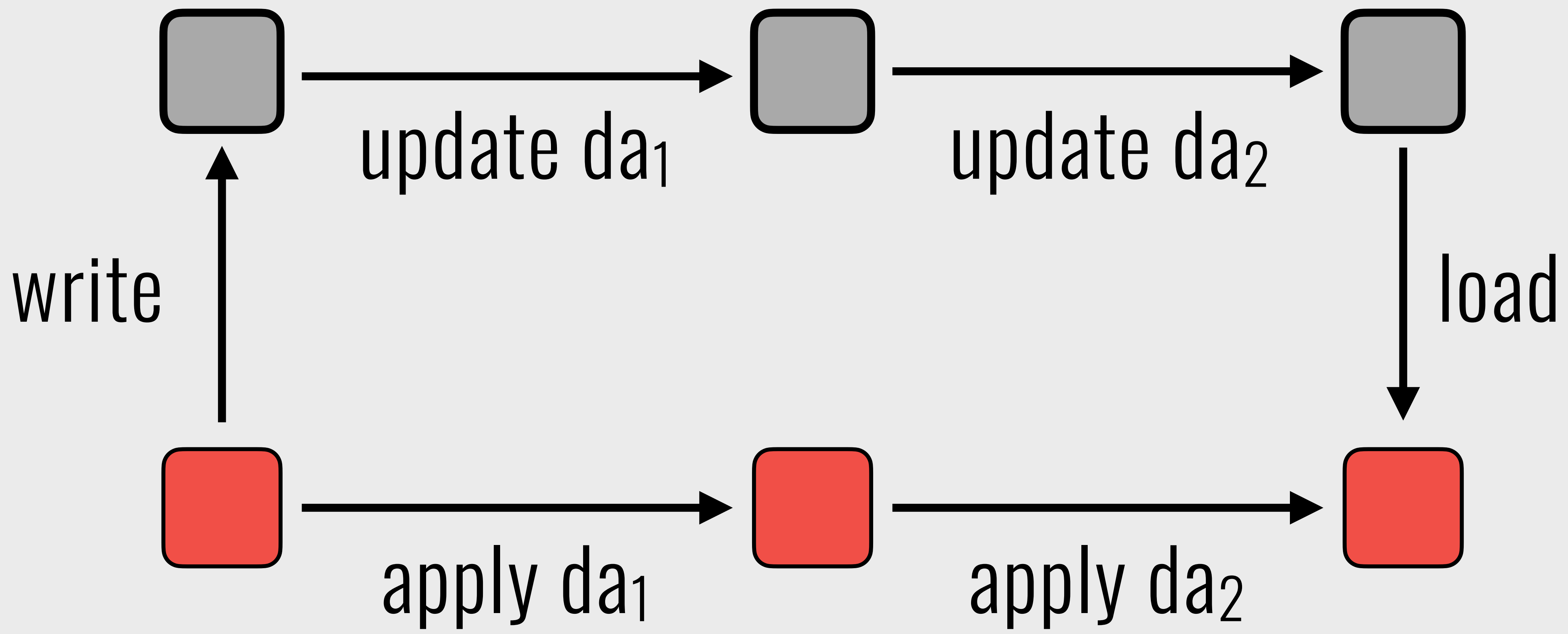
Store, testing

```
data Store m delta
```



Store, testing

```
data Store m delta
```



$$\begin{tikzcd}
\square \arrow[r, "\text{update } da_1"'] & \square \arrow[r, "\text{update } da_2"'] & \square \arrow[d, "\text{load}"] \\
\blacksquare \arrow[u, "\text{write}"] \arrow[r, "\text{apply } da_1"'] & \blacksquare \arrow[r, "\text{apply } da_2"'] & \blacksquare
\end{tikzcd}$$

Store, wallet

```
data Store m delta
```

for:

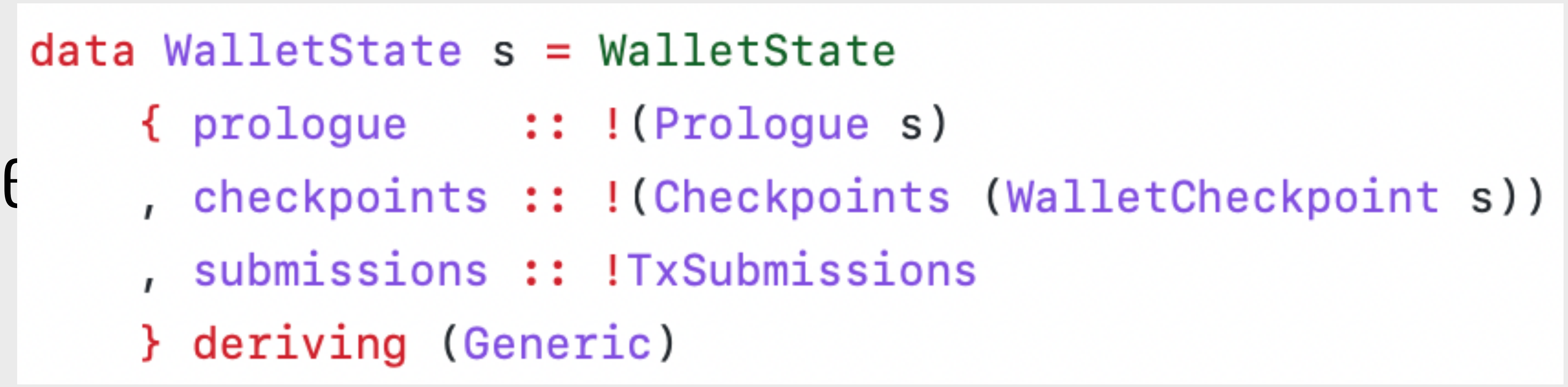
- submitted transactions
- funds
- addresses
- transaction history
- …

Store, wallet

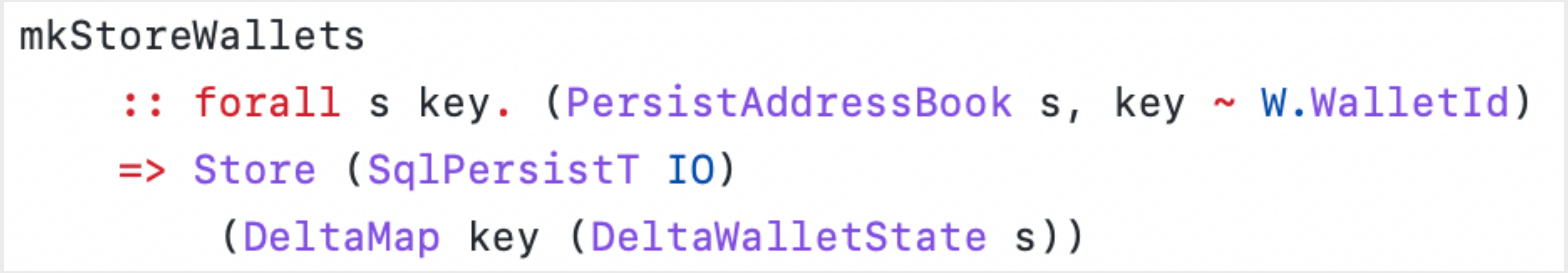
```
data Store m delta
```

for:

- submitte
- funds
- addresses



```haskell
data WalletState s = WalletState
    { prologue    :: !(Prologue s)
    , checkpoints :: !(Checkpoints (WalletCheckpoint s))
    , submissions :: !TxSubmissions
    } deriving (Generic)
```



```haskell
mkStoreWallets
    :: forall s key. (PersistAddressBook s, key ~ W.WalletId)
    => Store (SqlPersistT IO)
        (DeltaMap key (DeltaWalletState s))
```

Store, wallet

```
data Store m delta
```

for:

- transaction history

list all transactions		
in-memory:	~0.5 min,	~7 GB RAM
on-disk:	~3.5 min,	~6 GB RAM

Conclusion

Separated concerns

business logic

- efficient updates
- class Delta
- one type, many Deltas

database operations

- Store: type for stored value

Image by @rahjada2.0