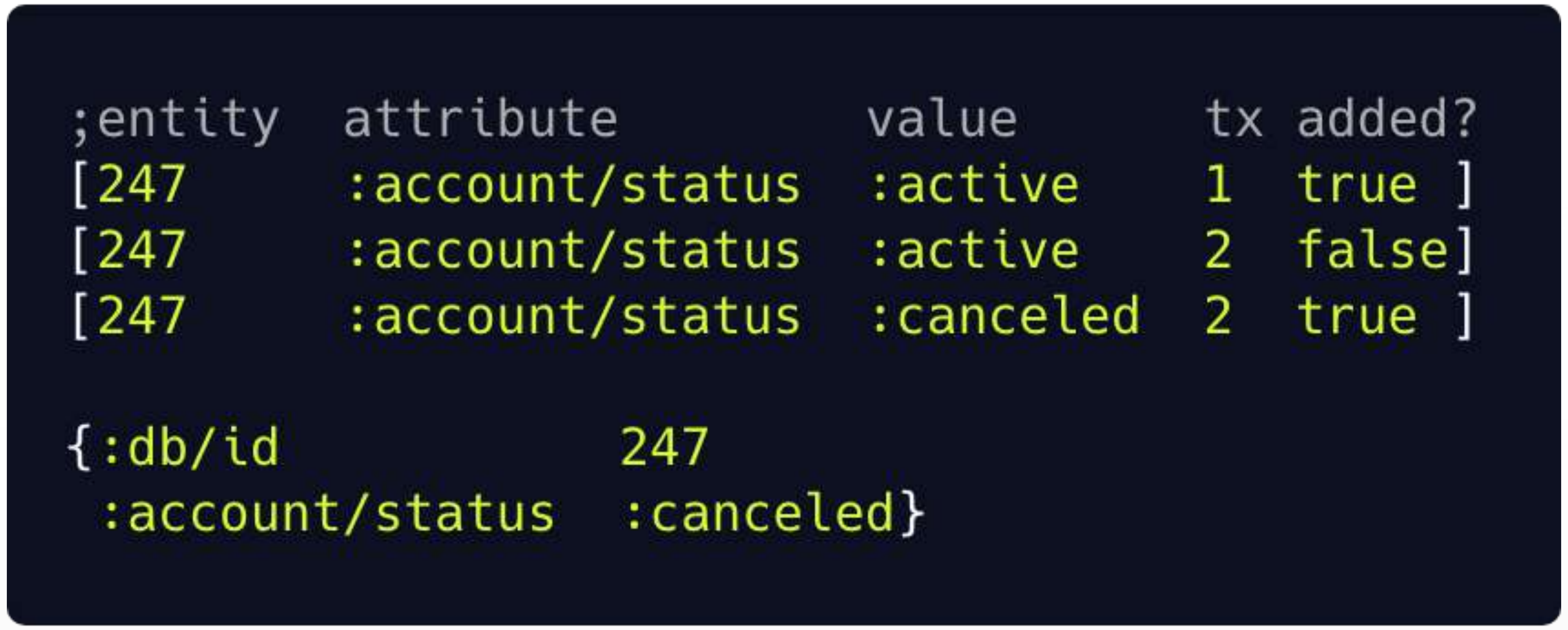
nubank

Consistent updates in Datomic

Gustavo Bicalho

Datomic in a nutshell

- Accumulate-only database - all changes add new datoms



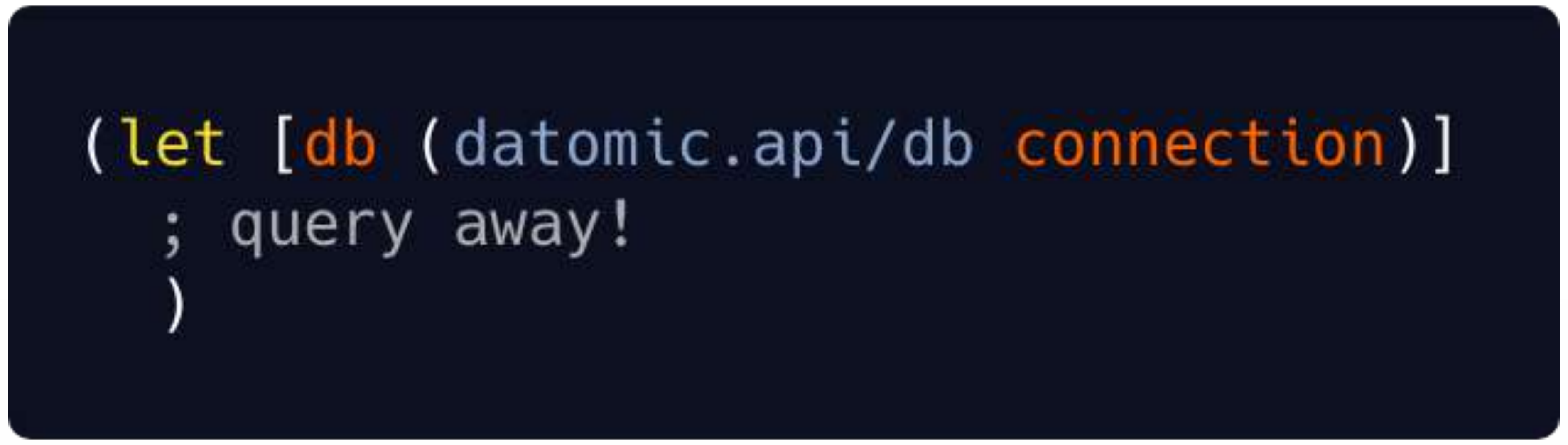
```
;entity  attribute         value      tx added?
[247     :account/status   :active    1  true ]
[247     :account/status   :active    2  false]
[247     :account/status   :canceled  2  true ]

{:db/id            247
 :account/status   :canceled}
```

- Good metaphor for this is Git
- Delegates storage to separate service, usually DynamoDB

Querying Datomic

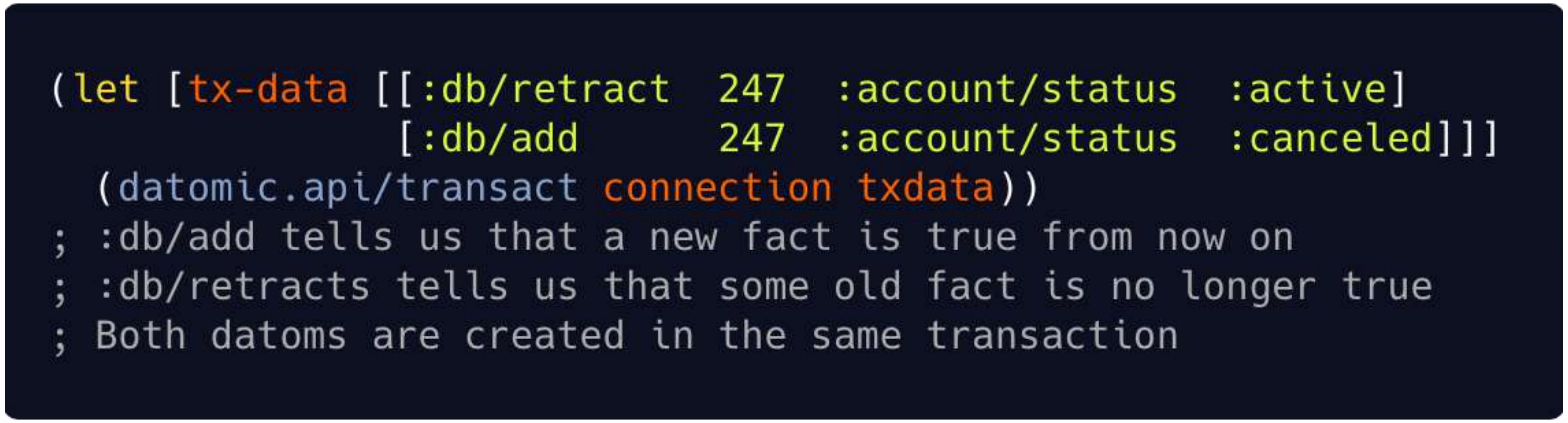
- `datomic.api/db` returns a `Db` object



```
(let [db (datomic.api/db connection)]
  ; query away!
  )
```

- `Db` gives us an immutable view of the database in a point in time
- Query with Datalog, the Entity API or access the datoms directly
- Several threads or machines can get `Db` and query independently

Writing to Datomic

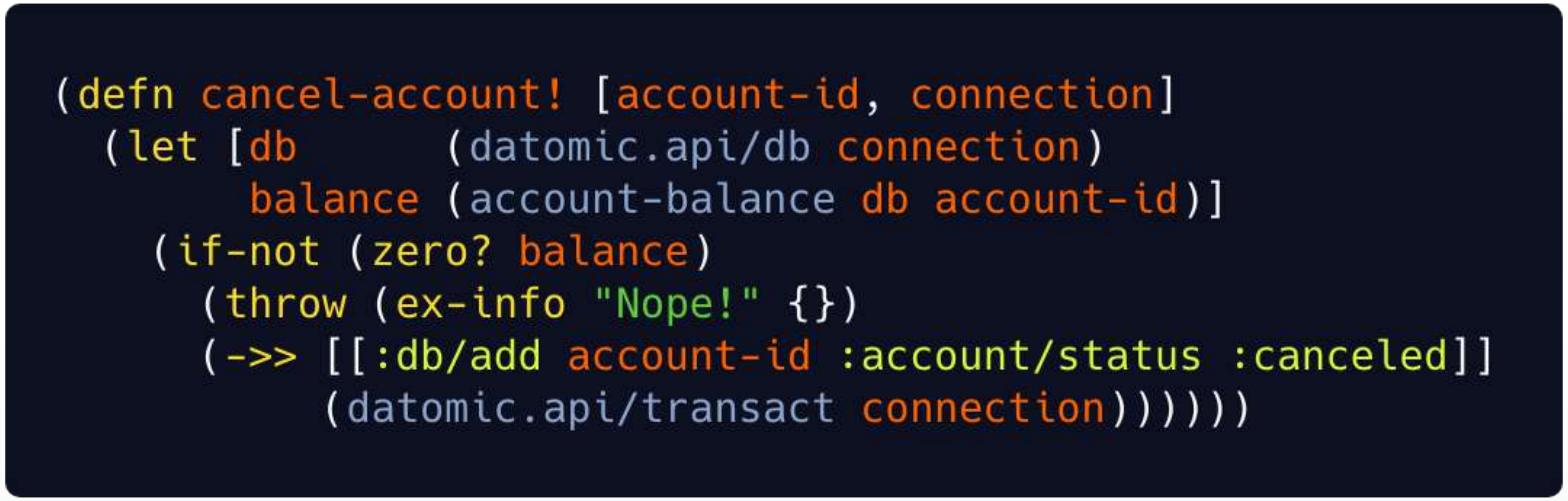


```
(let [tx-data [[:db/retract  247  :account/status  :active]
               [:db/add      247  :account/status  :canceled]]]
  (datomic.api/transact connection txdata))
; :db/add tells us that a new fact is true from now on
; :db/retracts tells us that some old fact is no longer true
; Both datoms are created in the same transaction
```

- `datomic.api/transact` sends a request to the **transactor**
- Transactor turns it into datoms and add them to the log
- No interactive transactions, all statements are sent at once
- Transactions run serially, must complete quickly

Writing to Datomic - Naive

- Get a `Db`, run a query, build statements, `transact`

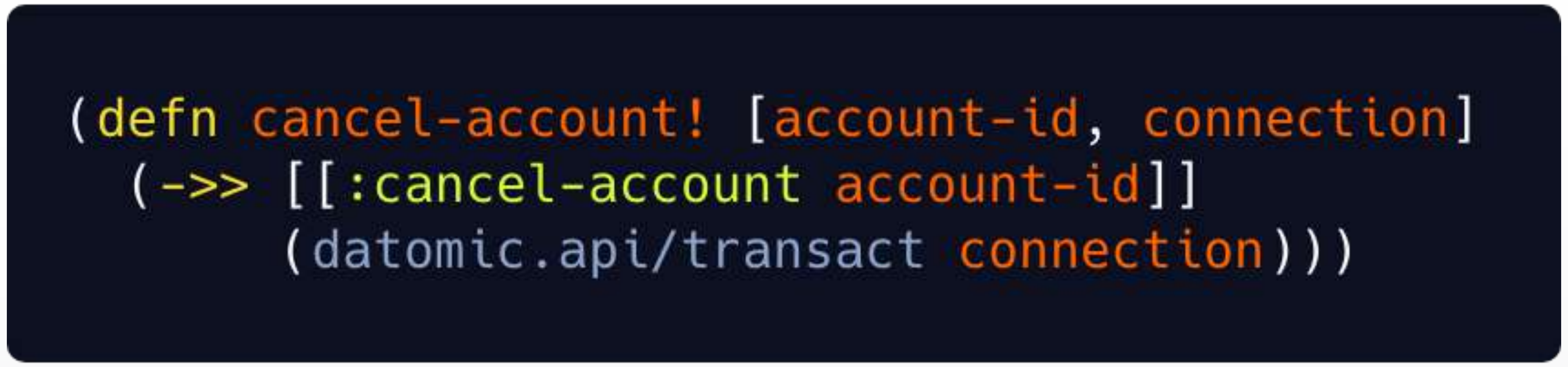


```
(defn cancel-account! [account-id, connection]
  (let [db      (datomic.api/db connection)
        balance (account-balance db account-id)]
    (if-not (zero? balance)
      (throw (ex-info "Nope!" {})
      (->> [[:db/add account-id :account/status :canceled]]
           (datomic.api/transact connection))))))
```

- Race conditions!

Writing to Datomic - Tx Functions

- Transaction functions are installed in the transactor
- Access the up-to-date snapshot of the database
- Can write stuff to the log or abort by throwing an exception



```
(defn cancel-account! [account-id, connection]
  (->> [[:cancel-account account-id]]
       (datomic.api/transact connection)))
```

- Slow transaction functions will clog all writes in the system

Isolation levels

- **A**tomicity
- **C**onsistency
- **Isolation**
- **D**urability

What happens when concurrent transactions touch the same data?

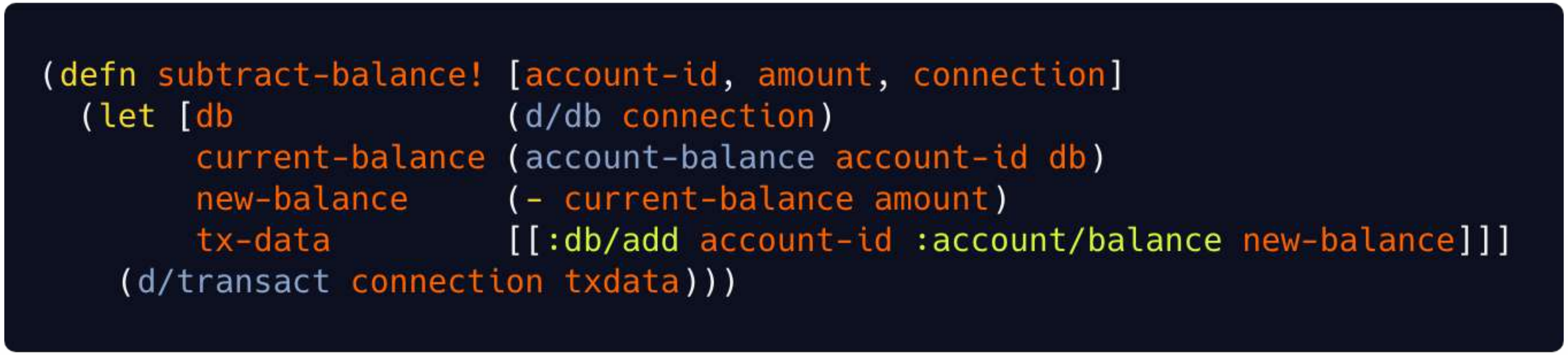
Serializable Isolation

- Transactions behave *as if* they were run serially
- Prevents all concurrency issues in a single database
- SQL databases use complex locking to achieve this. Concurrent transactions that touch the same data either get blocked or abort
- Datomic actually runs every transaction serially!

Snapshot Isolation

- All reads in a transaction see the same version of the database
- Concurrent transactions do not see each other
- Looks a lot like what we get from Datomic's `Db`!
- `d/transact` is serializable, but querying `Db` is snapshot-isolated
- Allows some concurrency anomalies to happen

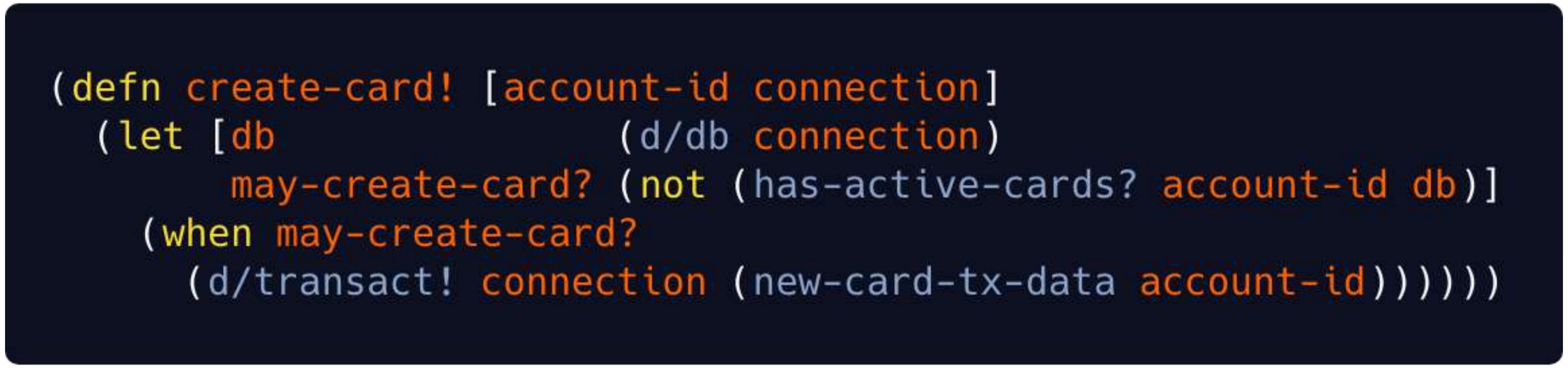
Snapshot Isolation - write-write conflicts



```
(defn subtract-balance! [account-id, amount, connection]
  (let [db              (d/db connection)
        current-balance (account-balance account-id db)
        new-balance     (- current-balance amount)
        tx-data         [[:db/add account-id :account/balance new-balance]]]
    (d/transact connection txdata)))
```

- Account starts with $100
- T1 calls `subtract-balance!` to take $90, writes $10
- T2 calls `subtract-balance!` to take $75, writes $25
- One of them will be overwritten

Snapshot Isolation - write skew



```
(defn create-card! [account-id connection]
  (let [db               (d/db connection)
        may-create-card? (not (has-active-cards? account-id db))]
    (when may-create-card?
      (d/transact! connection (new-card-tx-data account-id)))))
```

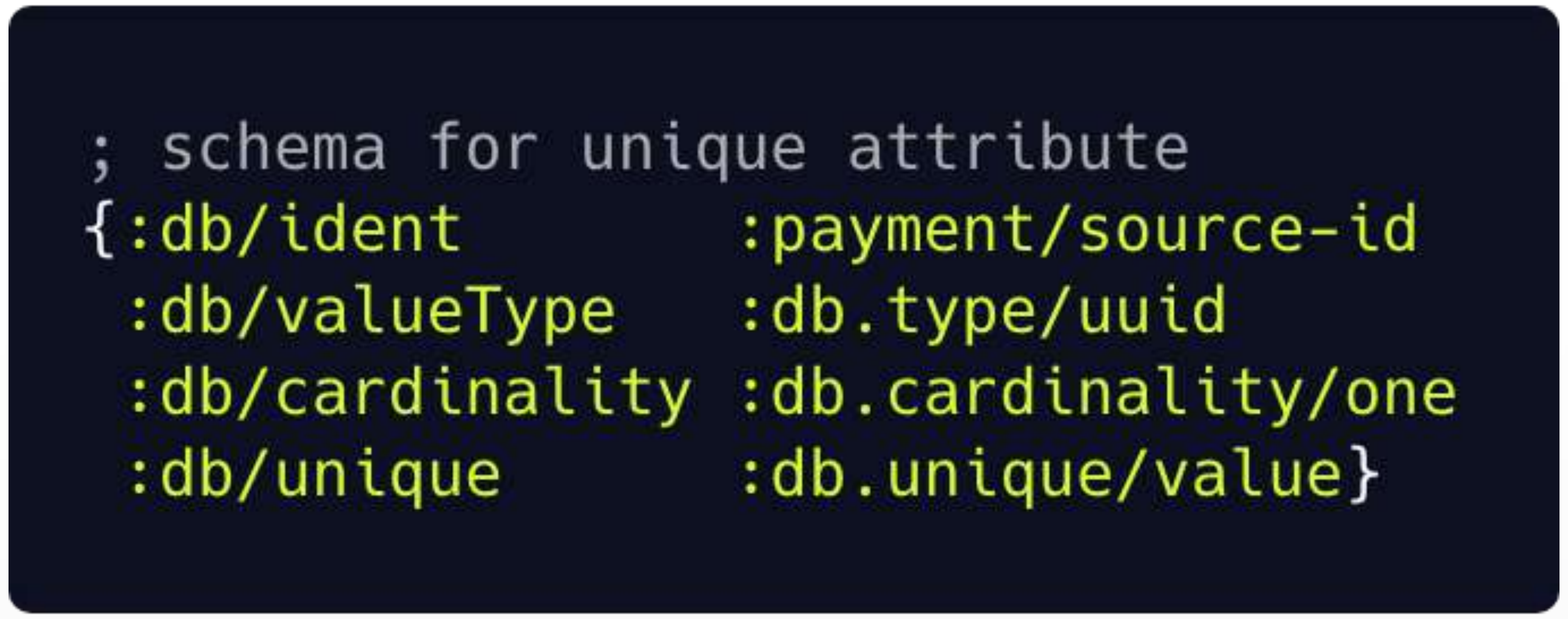
- Account has no active cards, should have at most 1
- T1 calls `create-card!`, sees snapshot with no cards, transacts
- T2 does the exact same thing
- We end up with 2 cards

Consistency in Datomic

- We want to do most work using `Db` queries
- Compute decisions with a snapshot
- Use a tx function to check if nothing changed
- Good data models can simplify this
- The less you read, the less you have to check
- The less you change, the less you have to check

Take advantage of uniqueness checks

- If we register an attribute as `unique`, Datomic will guarantee that for us automatically

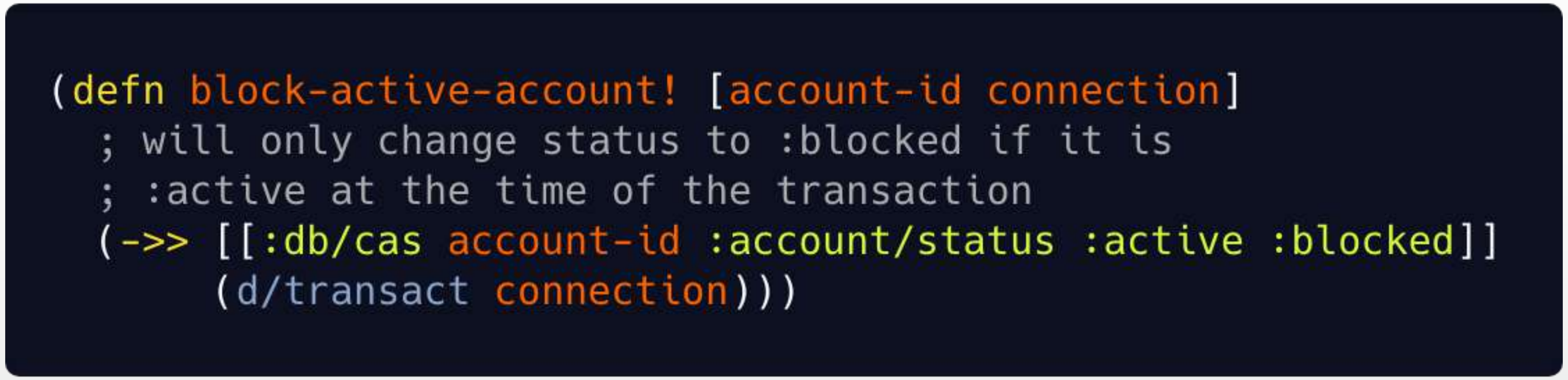


```
; schema for unique attribute
{:db/ident       :payment/source-id
 :db/valueType   :db.type/uuid
 :db/cardinality :db.cardinality/one
 :db/unique      :db.unique/value}
```

- Consume an input -> save a new payment, with the unique `:payment/source-id` attribute
- If there’s no unique id field in the input, we can build a unique hash

Use `:db/cas`

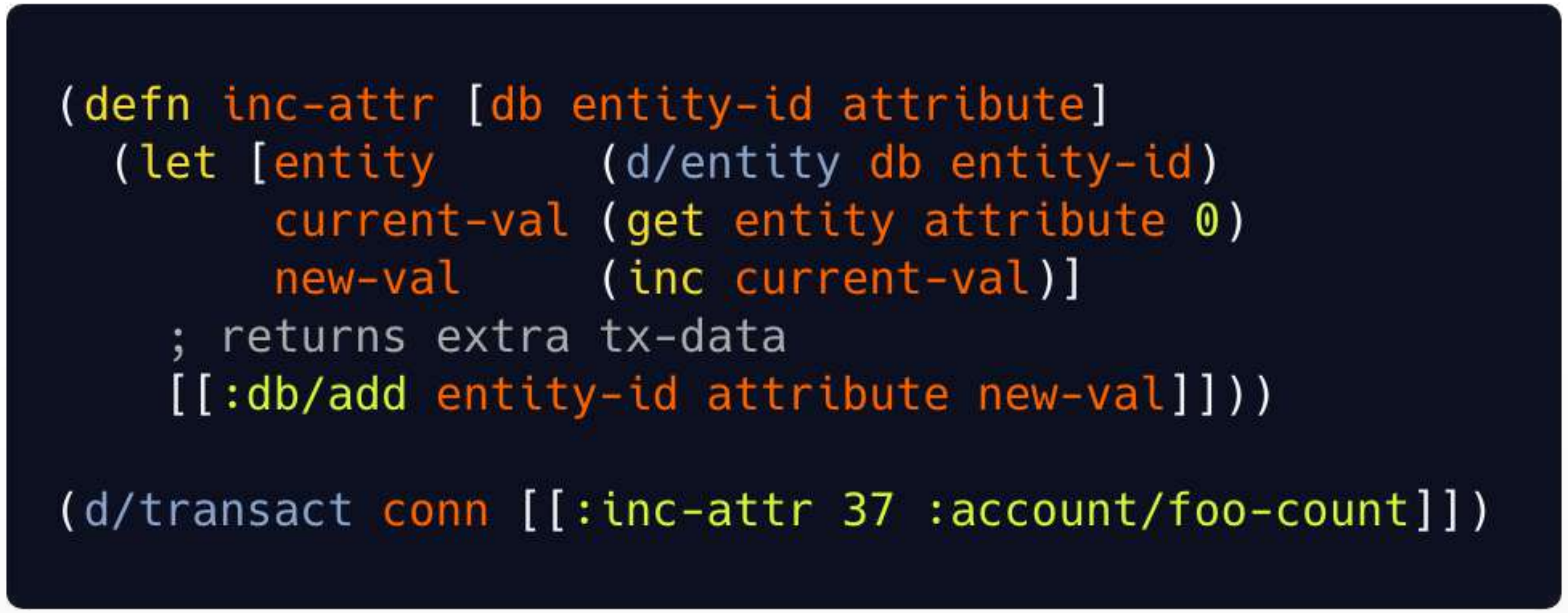
- `:db/cas` is a built-in transaction fn that does **c**ompare-**a**nd-**s**wap
- Solves write-write conflicts by checking that no concurrent transaction changed the attribute we're changing
- Use it to implement state-machines, where we want to ensure valid transitions



```
(defn block-active-account! [account-id connection]
  ; will only change status to :blocked if it is
  ; :active at the time of the transaction
  (->> [[:db/cas account-id :account/status :active :blocked]]
       (d/transact connection)))
```

Custom transaction functions

- Installed via transaction or via classpath



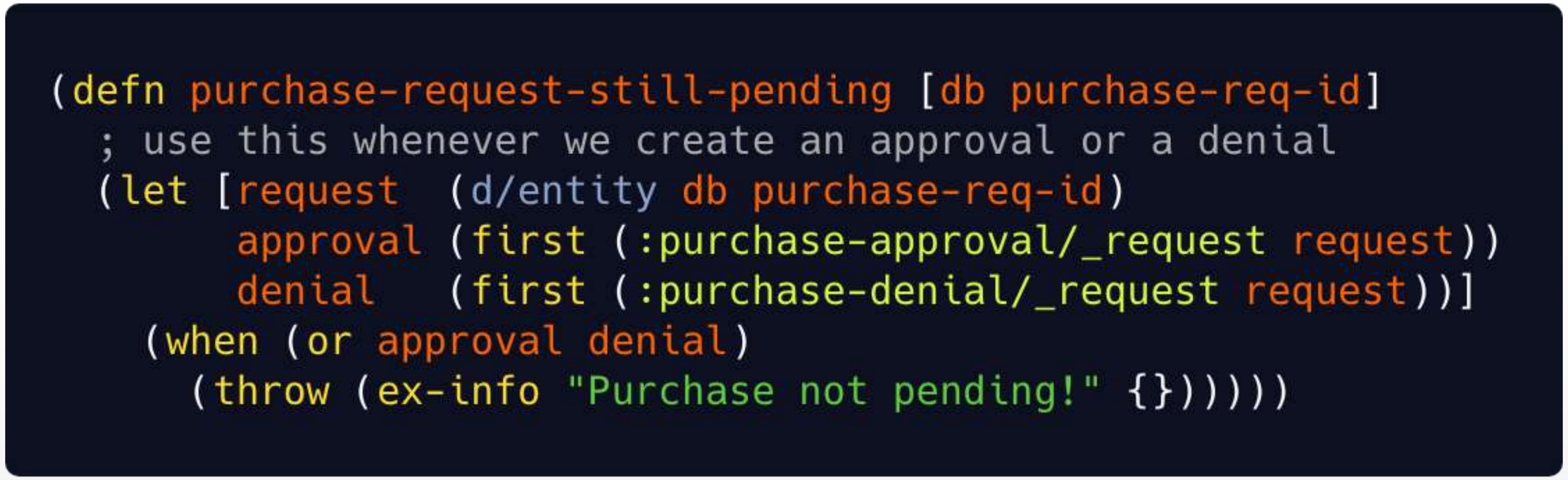
```
(defn inc-attr [db entity-id attribute]
  (let [entity      (d/entity db entity-id)
        current-val (get entity attribute 0)
        new-val     (inc current-val)]
    ; returns extra tx-data
    [[:db/add entity-id attribute new-val]]))

(d/transact conn [[:inc-attr 37 :account/foo-count]])
```

- Can query the up-to-date version of the database
- Pure functions that return some tx-data or throw an exception

Tx fns - checking for child entities

- Example: `purchase-request` must have at most one of two child entities, `purchase-approval` and `purchase-denial`
- Write skew could violate this condition

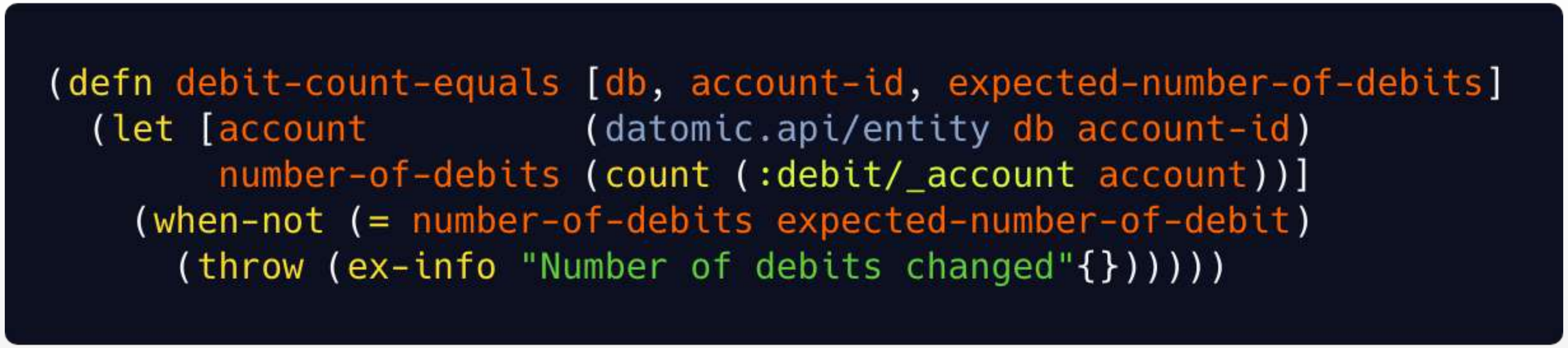


```
(defn purchase-request-still-pending [db purchase-req-id]
  ; use this whenever we create an approval or a denial
  (let [request  (d/entity db purchase-req-id)
        approval (first (:purchase-approval/_request request))
        denial   (first (:purchase-denial/_request request))]
    (when (or approval denial)
      (throw (ex-info "Purchase not pending!" {})))))
```

- Check for existence at the serializable level, before inserting

Tx fns - lists of entities

- Example:
  - compute the balance of a bank-account from the `debit` and `credit` entities the belong to it
  - Never create a `debit` if the balance would become negative

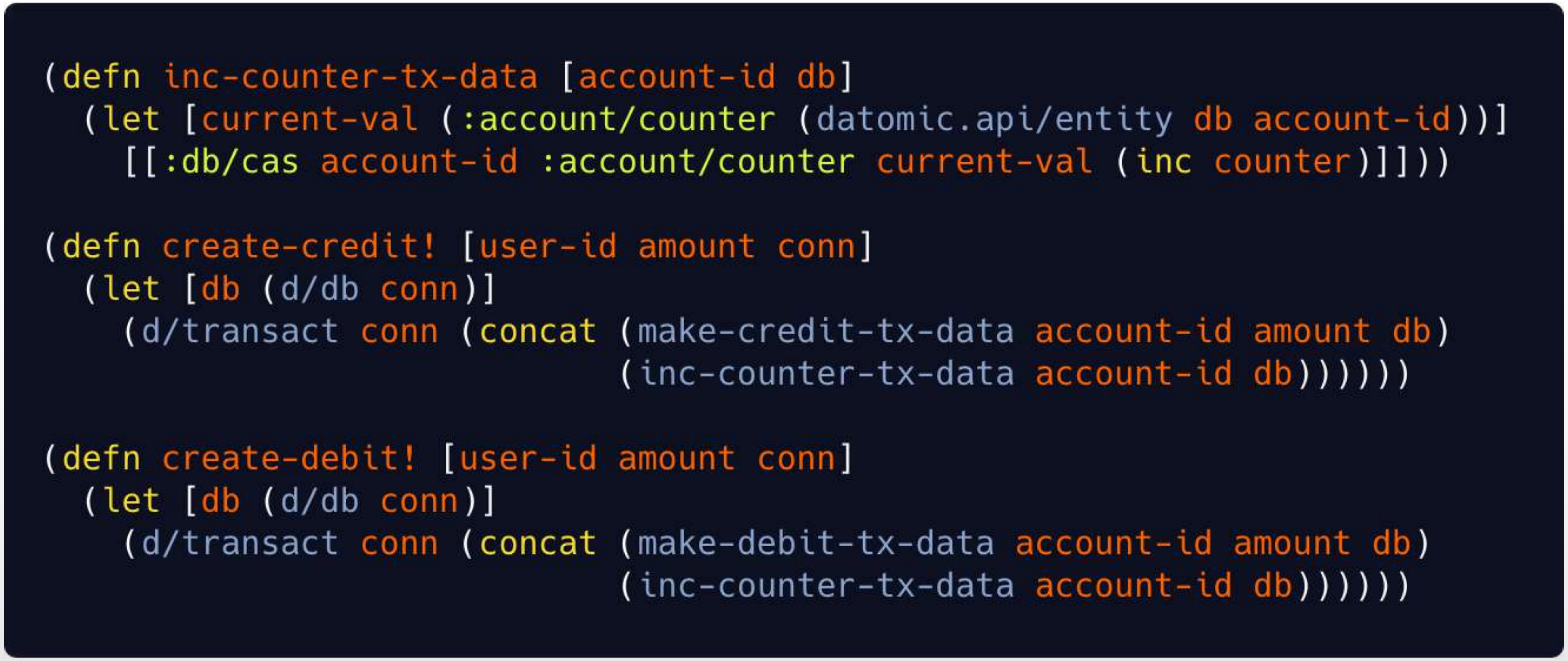


```
(defn debit-count-equals [db, account-id, expected-number-of-debits]
  (let [account           (datomic.api/entity db account-id)
        number-of-debits (count (:debit/_account account))]
    (when-not (= number-of-debits expected-number-of-debit)
      (throw (ex-info "Number of debits changed"{})))))
```

- We can do this because our entities are immutable
- Smarter checks relying on domain knowledge

Account lock

- To ensure all updates that relate to a single `account` are serialized, use an attribute as a sequential counter
- Update the counter with `:db/cas`



```clojure
(defn inc-counter-tx-data [account-id db]
  (let [current-val (:account/counter (datomic.api/entity db account-id))]
    [[:db/cas account-id :account/counter current-val (inc counter)]]))

(defn create-credit! [user-id amount conn]
  (let [db (d/db conn)]
    (d/transact conn (concat (make-credit-tx-data account-id amount db)
                             (inc-counter-tx-data account-id db)))))

(defn create-debit! [user-id amount conn]
  (let [db (d/db conn)]
    (d/transact conn (concat (make-debit-tx-data account-id amount db)
                             (inc-counter-tx-data account-id db)))))
```

Wrapping up

- Database as a value is awesome, but not a silver bullet
- Datomic has two different APIs, two distinct isolation levels
- If we think carefully, we can move work between isolation levels, and get the best of each
- Think about concurrency when writing transactions
- Think about concurrency when designing data models