**Motivation**

- Optimistic MVCC require validation phase before commit
- Aborts and restarts conflicting transactions
- Suboptimal for the use-cases with
  - High contention data objects
  - Long running transactions

**Use Cases**

```
begin
  B
  A
commit
```

```
begin
  B
  A
commit
```

```
begin
  Data query
```

**Requirements**

1. Low overhead on normal execution of transactions
2. Fast mechanism to narrow down conflicting portions of transactions and repair them

**Components**

- **Versions** store the value of a data object written by a transaction, as well as its timestamp. Versions are grouped together by data object in *version* chains, and by transaction in *undo buffer*.

- **Predicates** encapsulate data selection criterion and the operations that depend on the result (closure). Stores references to versions created by the closure. Stores references to parent and child predicates forming *predicate graph*. Implements *match* operation that identifies whether a given version should be part of the result-set.

- **Closures** represent deterministic function that encloses all operations in a transaction that depends on the result of a data selection criteria (predicate)

**Banking Example – An Execution Snapshot**

```
/*  in = from, acc = account and bal = balance */
TransactionMoney(int in, te, to_acc, amount) { 
 START: 
 IF (amount < 1000) fee = 3.5;
 ELSE fee = amount * 0.9;
 P, AccountMoney id = from->to acc entry
 if (acc->entry) bal = amount - fee
 TransactionMoney from, te, to_acc, amount
 P, AccountMoney id = fee acc entry
 fee acc entry bal = fee
 fee acc entry persist;
 return; 
 } 
```

**Transaction Lifecycle**

```
Validation 
Repair
```

**Execution**

1. The first root predicate gets instatiated and evaluated
2. The closure of the predicate is invoked with the result-set as argument
   i. This instantiates predicates defined in the closure
   ii. They are evaluated and their closures are invoked
3. The next root predicate is instantiated and the process repeated

**Validation**

1. The predicate graph is traversed in topological sort order, first traversing the higher-level predicates.
2. Predicates of transaction are matched against the committed versions of other transactions that committed during its execution
3. If none of the predicates match against any version, then the validation is successful, and the transaction commits.
4. Otherwise, the matched predicates, along with all their descendant predicates, are marked as invalid.

**Repair**

1. Obtain new start timestamp for transaction
2. For all predicates marked invalid by validation
   a) Delete versions created by predicate
   b) Delete all child predicates, as well as their versions
3. For all remaining invalid predicates
   a) Fix result set of predicate
   b) Invoke closure with new result-set

**Banking Example**

![Banking Example](image)

**Experimental Results**

- Simulates a simplified trading system with encryption
- **Tables:**
  - Security (s_id, symbol, s_price)
  - Customer (c_id, cipher_key)
  - Trade (t_id, c_id, t_encrypted_data)
  - TradeLine (t_id, t_id, t_id, t_encrypted_data)
- **Transactions:**
  - `TradeOrder`: to trade a given list of securities for a given customer
  - `PriceUpdate`: to update the price of a given security

![Trading - concurrency](image)

![Trading - conflicts](image)