Pyrrho and the LDBC Financial Benchmark

Malcolm Crowe, Draft 12 September 2024

Metadata and truncating in Pyrrho

Apart from having a different set of standard types, Pyrrho has some additional syntax for metadata, edge type end points and graph matching truncation. Full details are in the Pyrrho manual: the specific additions to ISO 39075 syntax described below match specifications in the LDBC Financial Benchmark specification. In the syntax notation in the Pyrrho manual, we have for example

```
GraphTypeDef = '{' ElementList '}' .
ElementList = (NodeTypeDetails|EdgeTypeDetails) {Metadata} {',' ElementList} .
EdgeTypeDetails =    [Direction Edge [TYPE] id] EdgePattern
                  |  Direction Edge [TYPE] (id|Filler) EndPoints.
EndPoints = CONNECTING '('NodeTypeRef ('<-'|'->'|TO|'~') NodeTypeRef ')' .
NodeTypeRef = NodeType_id {Metadata} {'|'NodeType_id {Metadata}}.
```

Metadata can be specified at the end of many DDL statements, and includes the syntax ((CARDINALITY|MULTIPLICITY) '(' int [TO (int|'*')] ')') . Cardinality, default 0 to *, applies to the edge type, while multiplicity, default 1 to *, applies to a node type reference. The syntax for EdgePattern also allows multiplicity to be specified inline.

In MatchStatement, we allow truncation specification, for all edge types, or for named edge types and their supertypes, according to an optionally specified ordering of such edges.

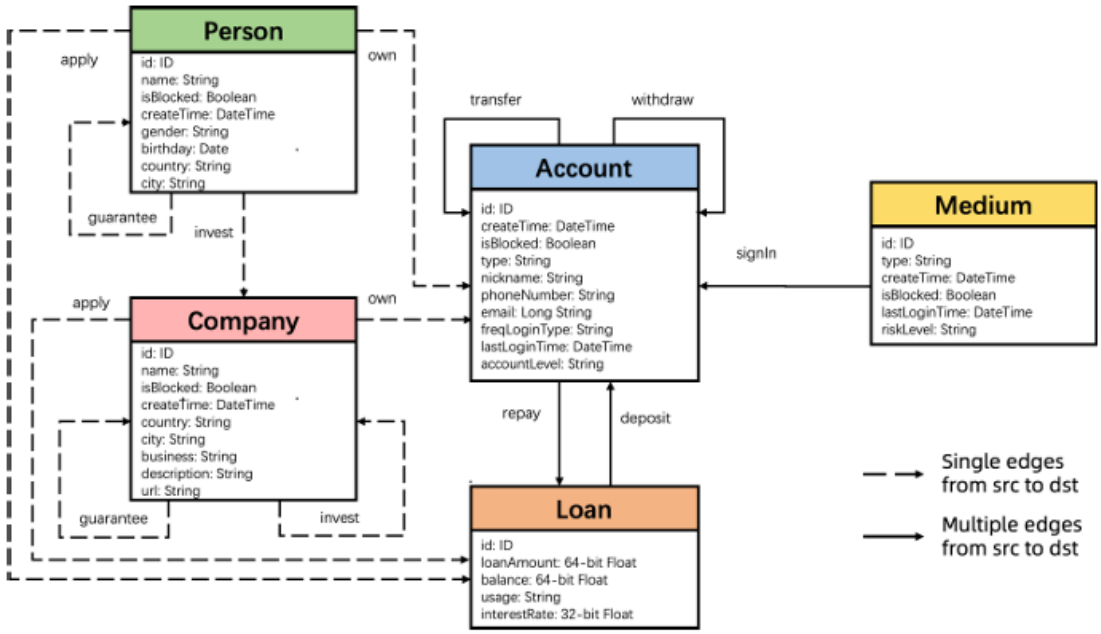
```
MatchStatement = MATCH [Truncation] Match {',' Match} [WhereClause] [Statement]
                 [THEN Statements END ] .
Truncation = TRUNCATING '('TruncationSpec{',' TruncationSpec}')' .

TruncationSpec = [EdgeType_id] ['('{OrderSpec {',' OrderSpec}')'] '=' int .
```

The truncation clause defines an upper bound for the number of edges to be traversed from a node in a step of the match process.

The LDBC Financial Benchmark

A full solution to the LDBC benchmark would be a database equipped with triggers so that the workloads run on every commit, to lock accounts and media for which apparently fraudulent transactions have been attempted. Applications (bank tellers etc) should report on such results, and the role-based security model should be used to ensure that only appropriately authorized persons or software can unlock accounts or media or suspend use of the triggers.

Pyrrho's client application needs multi-line statements to be enclosed in square brackets, and these have been added to the code soamples in these notes.

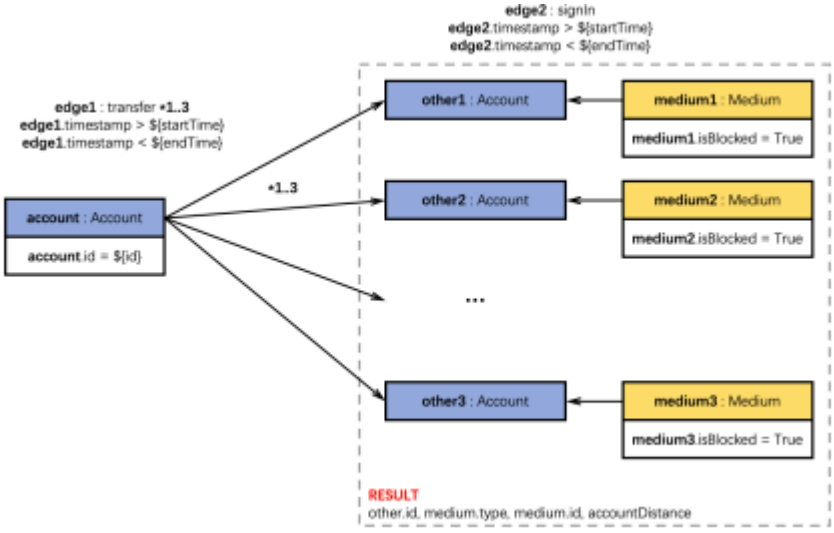



```
create schema /ldbc
[create graph type /ldbc/finBenchMark  {
   node Person {id::int,name::string,isBlocked::boolean,
      createTime::timestamp,gender::string,birthday::date,country::string,
      city::string},
   node Account {id::int,createTime::timestamp,isBlocked::boolean,
      type::string,nickname::string,phoneNumber::string,email::string,
      freqLoginType::string,lastLoginTime::timestamp,accountLevel::string},
   node Medium {id::int,type::string,isBlocked::boolean,
      createTime::timestamp,lastLoginTime::timestamp,riskLevel::string},
   node Company{id::int,name::string,isBlocked::boolean,
      createTime::timestamp,country::string,city::string,
      business::string,description::string, url::string},
   node Loan {id::int,loanAmount::decimal,balance::decimal,
      createTime::timestamp,usage::string,interestRate::decimal},
   directed edge Transfer {amount::decimal,createTime::timestamp,
      ordernumber::string,comment::string,payType::string,
      goodsType::string} connecting (Account to Account),
   directed edge Withdraw {createTime::timestamp,amount::decimal}
      connecting (Account to Account),
   directed edge Repay {createTime::timestamp,amount::decimal} connecting
      (Account to Loan),
   directed edge Deposit {createTime::timestamp,amount::decimal} connecting (Loan
      to Account),
   directed edge SignIn {createTime::timestamp,location::string}
      connecting (Medium to Account),
   directed edge Invest {createTime::timestamp,ratio::decimal}
      connecting (Person|Company to Company) cardinality (1),
   directed edge Apply {createTime::timestamp,organization::string}
      connecting (Person|Company to Loan) cardinality (1),
   directed edge Guarantee {createTime::timestamp,relationship::string}
      connecting (Person|Company to Person|Company) cardinality (1),
   directed edge Own {createTime::timestamp}
      connecting (Person|Company to Account) cardinality(1)}]
```

If p is a path reference and x a transfer edge in the path, the condition for ensuring chains of transfers are in sequence can be written as a where condition:

```
where let c=cardinality(p.x)
   case c when 0 then true else p.x[c-1].createTime<createtime end
```

TCR1



Given an Account and a specified time window between startTime and endTime, find all the Account that is signed in by a blocked Medium and has fund transferred via edge1 by at most 3 steps. Note that all timestamps in the transfer trace must be in ascending order(only greater than). Return the id of the account, the distance from the account to given one, the id and type of the related medium.

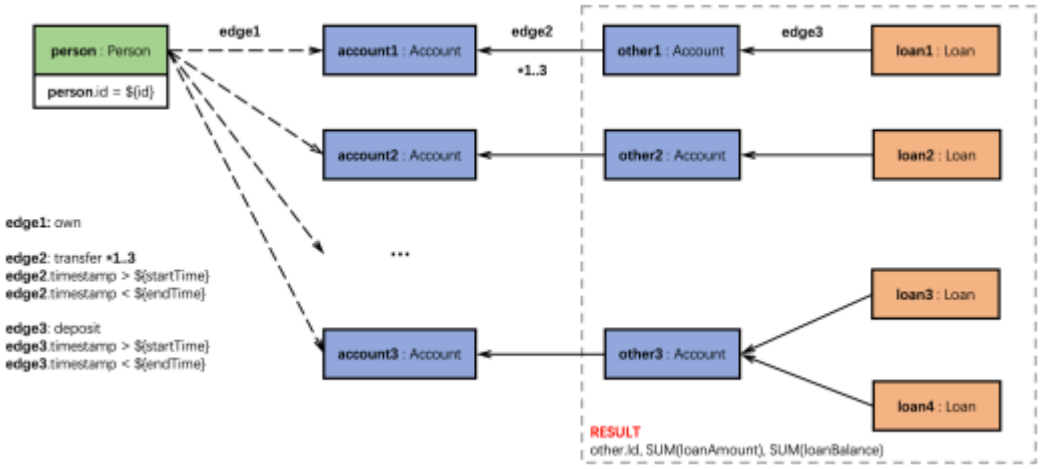
Params id1,startTime,endTime,truncationLimit,truncationOrder

Result otherId,accountDistance,mediumId,mediumType

Sort accountDistance, otherId, mediumId

```
[CREATE PROCEDURE ComplexRead1(id1 int,startTime timestamp,endTime
timestamp,truncationLimit int,truncationOrder string)
MATCH
   truncating (transfer (truncationOrder)=truncationLimit)
   trail p=(m:Medium{isBlocked:true})
      -[:signIn where createTime>startTime and createTime<endTime]->
         (:Account{id:otherId}) [()
      -[x:transfer where createTime>startTime and createTime<endTime
         and (cardinality(p) =0 or
         p.x[cardinality(p)-1].createTime<createtime)
       ]->()]{1,3} (:Account{id:id1})
   RETURN
      otherId,
      (cardinality(p)-3)/2 as accountDistance,
      m.id as mediumId,
      m.type as mediumType
      order by (accountDistance,otherId,mediumId)]
```

TCR2



Given a Person and a specified time window between startTime and endTime, find an Account owned by the Person which has fund transferred from other Accounts by at most 3 steps (edge2) which has fund deposited from a loan. The timestamps of in transfer trace (edge2) must be in ascending order(only greater than) from the upstream to downstream. Return the sum of distinct loan amount, the sum of distinct loan balance and the count of distinct loans.

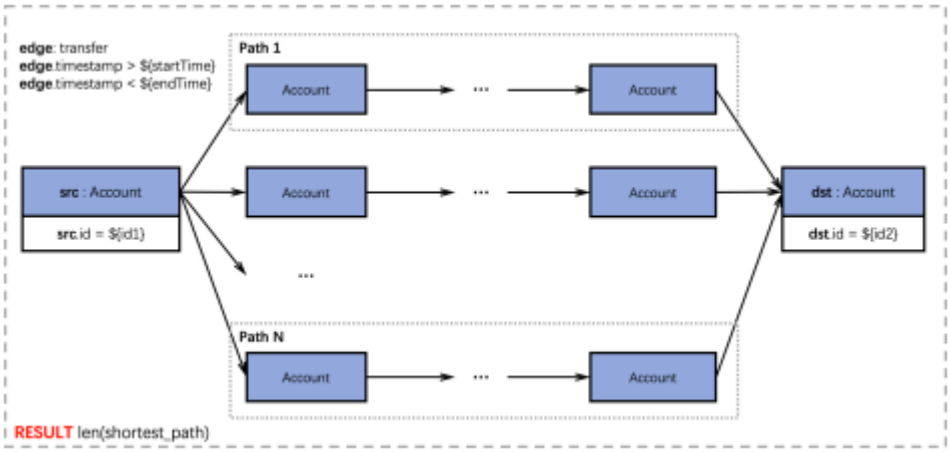
Params: id1, startTime, endTime, truncationLimit, truncationOrder

Result: otherId, sumLoanAmount, sumLoanBalance

Sort: sumLoanAmount, ??sumLoanBalance, otherId

```
[CREATE PROCEDURE ComplexRead2(id1 int, startTime timestamp, endTime
timestamp, truncationLimit int, truncationOrder string)
MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
      trail p=(:Person{id:id1})-[:own]->(:Account)
            [()<-[x:transfer
                  where createTime >startTime and createTime <endTime
                  and (cardinality(p) =0
                    or p.x[cardinality(p)-1].createTime<createtime)
                  ]-()]{1,3} (:Account{id:otherId})<-[:deposit]-(a:Loan)
      return
            otherId,
            sum (a.loanAmount) as sumLoanAmount,
            sum (a.balance) as sumLoanBalance,
            count(a)
            group by otherId
            order by (sumLoanAmount desc,otherId)]
```

TCR3



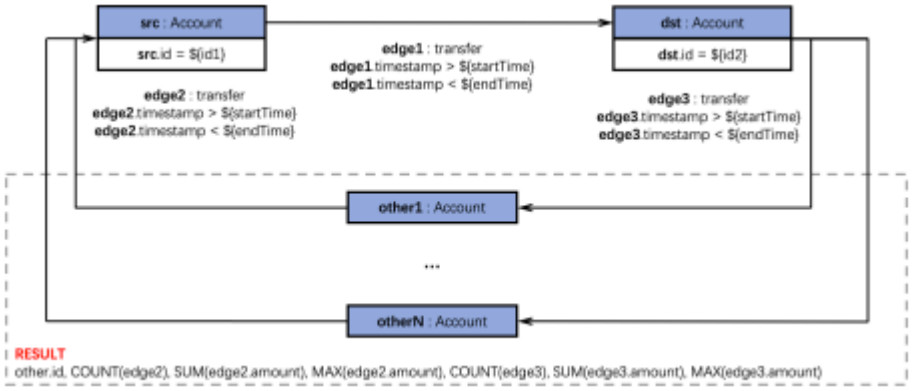
Given two accounts and a specified time window between startTime and endTime, find the length of shortest path between these two accounts by the transfer relationships. Note that all the edges in the path should be in the time window and of type transfer. Return 1 if src and dst are directly connected. Return -1 if there is no path found.

Params: id1, id2, startTime, endTime

Result: shortestPathLength

```
[CREATE PROCEDURE ComplexRead3(id1 int, id2 int, startTime timestamp,
endTime timestamp)
MATCH
      shortest p=(:Account{id:id1})
            [()-[x:transfer
                  where createTime >startTime and createTime <endTime
                  and (cardinality(p)=0 or
                  p.x[cardinality(p)-1].createTime<createtime)]->()]+
            (:Account{id:id2})
RETURN min(cardinality(p)-4) as shortestPathLength]
```

TCR4



Given two accounts src and dst, and a specified time window between startTime and endTime, (1) check whether src transferred money to dst in the given time window (edge1). If edge1 does not exist, return with empty results (the result size is 0). (2) find all other accounts (other1, . . . , otherN) which received money from dst (edge2) and transferred money to src (edge3) in a specific time. For each of these other accounts, return the id of the account, the sum and max of the transfer amount (edge2 and edge3)

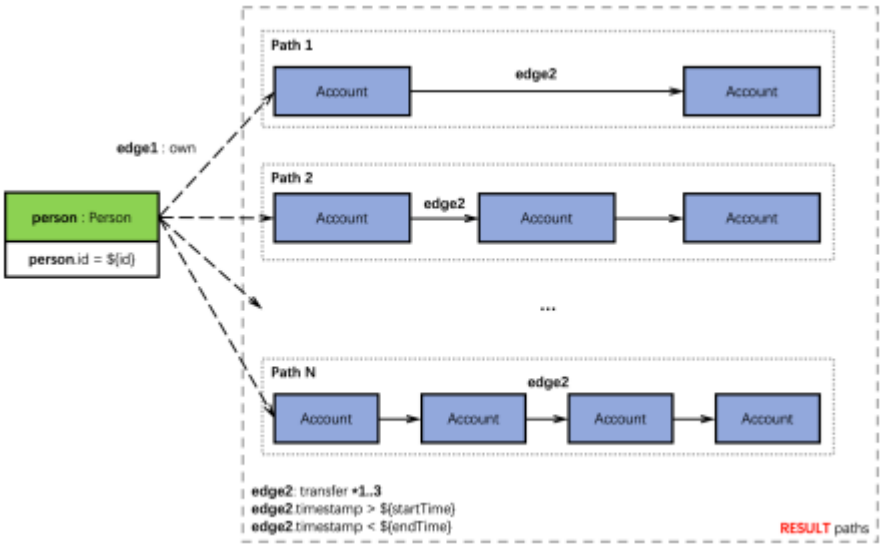
Params: id1, id2, startTime, endTime

Result: otherId, numEdge2, sumEdge2Amount, maxEdge2Amount, numEdge3, sumEdge3Amount, maxEdge3Amount

Sort: sumEdge2Amount desc,sumEdge3Amount desc, otherId

```
[CREATE PROCEDURE ComplexRead4 (id1 int, id2 int, startTime timestamp,
endTime timestamp)
MATCH
      (src:Account{id:id1})
      -[:transfer
            where createTime>startTime and createTime<endTime]->
      (dst:Account{id:id2}),
      (:Account{id:otherId})
      -[:transfer {amount:amt2}
            where createTime>startTime and createTime<endTime]->
      (src:Account{id:id2}),
      (dst:Account{id:id1})
      -[:transfer {amount:amt3}
            where createTime>startTime and createTime<endTime]->
      (:Account{id:otherId})
return
      count(amt2) as numEdge2,
      sum(amt2) as sumEdge2Amount,
      max(amt2) as maxEdge2Amount,
      count(amt3) as numEdge3,
      sum(amt3) as sumEdge3Amount,
      max(amt3) as maxEdge3Amount,
      otherId
      group by otherId
      order by (sumEdge2Amount desc, sumEdge3Amount desc, otherId)]
```

TRC5



Given a Person and a specified time window between startTime and endTime, find the transfer trace from the account (src) owned by the Person to another account (dst) by at most 3 steps. Note that the trace (edge2) must be ascending order(only greater than) of their timestamps. Return all the transfer traces. Note: Multiple edges of from the same src to the same dst should be seen as identical path. And the resulting paths shall not include recurring accounts (cycles in the trace are not allowed). The results may not be in a deterministic order since they are only sorted by the length of the path. Driver will validate the results after sorting.

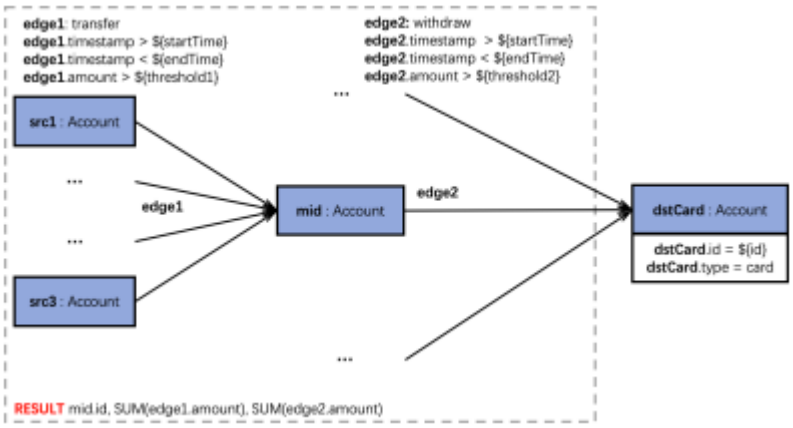
Params: id1, startTime, endTime, truncationLimit, truncationOrder

Result: Path

Sort: pathLength

```
[CREATE PROCEDURE ComplexRead5(id1 int, startTime timestamp,endTime
timestamp,truncationLimit int, truncationOrder string)
MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
      trail p=(:Account{id:id1})
            [()-[x:transfer
                  where createTime>startTime and createTime<endTime and
                  (cardinality(p)=0 or
                  p.x[cardinality(p)-1].createTime<createtime)]->()]{1,3}
            (:Account{id:id2})
      order by cardinality(p) desc]
```

TRC6



Given an account of type card and a specified time window between startTime and endTime, find all the connected accounts (mid) via withdrawal (edge2) satisfying, (1) More than 3 transfer-ins (edge1) from other accounts (src) whose amount exceeds threshold1. (2) The amount of withdrawal (edge2) from mid to dstCard whose exceeds threshold2. Return the sum of transfer amount from src to mid, the amount from mid to dstCard grouped by mid.

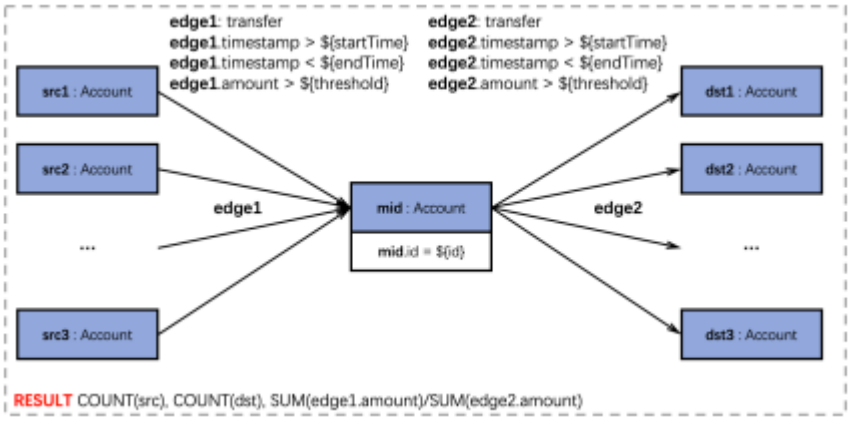
Params: id1, threshold1, threshold2, startTime, endTime, truncationLimit, truncationOrder

Result: midId, sumEdge1Amount, sumEdge2Amount

Sort: sumEdge2Amount desc, midId

```
[CREATE PROCEDURE ComplexRead6(id1 int, threshold1 decimal, threshold2
decimal, startTime timestamp, endTime timestamp, truncationLimit int,
truncationOrder string)
MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
      (:Account{id:id1,type:'card'})
      <-[:withdrawal {amount:amt2}
            where createTime>startTime and createTime<endTime
            and amt2>threshold2]-(mid:Account)
      <-[:transfer {amount:amt1}
            where createTime>startTime and createTime<endTime
            and amt1>threshold1
            and count(amt1)>3]-(:Account)
      return
            sum(amt1) as sumEdge1Ammount,
            sum(amt2) as sumEdge2Amount,
            mid.id as midId group by midId
            order by (sumEdge2Amount desc, midId)]
```

TRC7



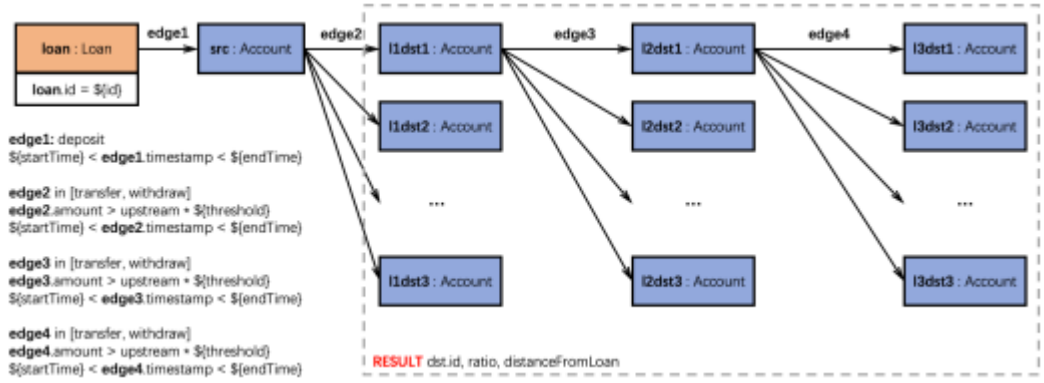
Given an Account and a specified time window between startTime and endTime, find all the transfer-in (edge1) and transfer-out (edge2) whose amount exceeds threshold. Return the count of src and dst accounts and the ratio of transfer-in amount over transfer-out amount. The fast-in and fast-out means a tight window between startTime and endTime. Return the ratio as -1 if there is no edge2.

Params: id1, threshold, startTime, endTime, truncationLimit, truncationOrder

Result: numSrc, numDst, inOutRatio

```
[CREATE PROCEDURE ComplexRead7(id1 int, threshold decimal, startTime
timestamp, endTime timestamp, truncationLimit int, truncationOrder string)
MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
      (src:Account)-[:transfer{amount:amt1}
                  where timestamp>startTime and timestamp<endTime
                  and amt1>threshold]->(:Account{id:id1})
      -[:transfer {amount:amt2}
            where createTime>startTime and createTime<endTime
            and amt2>threshold]->(dst:Account)
      return
            count(src) as numSrc,
            count(dst) as numDst,
            case count(amt2) when 0 then -1 else sum(amt1)/sum(amt2) end
                  as inOutRatio]
```

TRC8



Given a Loan and a specified time window between startTime and endTime, trace the fund transfer or withdraw by at most 3 steps from the account the Loan deposits. Note that the transfer paths of edge1, edge2, edge3 and edge4 are in a specific time range between startTime and endTime. Amount of each transfers or withdrawals between the account and the upstream account should exceed a specified threshold of the upstream transfer. Return all the accounts' id in the downstream of loan with the final ratio and distanceFromLoan. Note: Upstream of an edge refers to the aggregated total amounts of all transfer-in edges of its source Account.

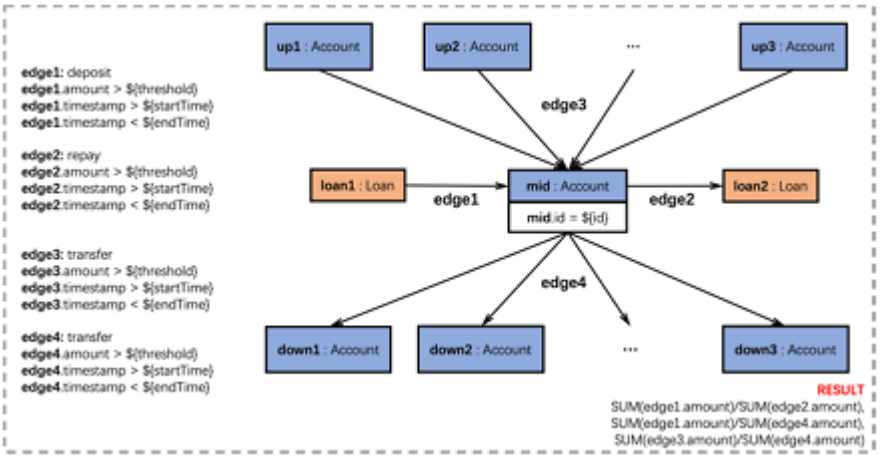
Params: id1, threshold, startTime, endTime, truncationLimit, truncationOrder

Result: dstId, ratio, minDistanceFromLoan

Sort: distanceFromLoan, ratio, dstId

```
[CREATE PROCEDURE ComplexRead8 (id1 int, threshold decimal, startTime
timestamp, endTime timestamp, truncationLimit int, truncationOrder string)
MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
      trail longest p=
        (:Loan{id:id1})-[:deposit{amount:depAmt}]->(:Account)
            [()-[x:transfer|withdraw {amount:amt}
                  where createTime>startTime and createTime<endTime
                  and (cardinality(p)=0 or amount>threshold*
            p.x[cardinality(p)-1].amount)]->(dst)]{1,3} (:Account)
      return
            cardinality(p)-2 as distanceFromLoan,
            dst.id as dstId,
            case count(amt) when 0 then -1
              else depAmt/sum(amt) end as ratio
            order by (distanceFromLoan, ratio, dstId)]
```

TCR9



Given an account, a bound of transfer amount and a specified time window between startTime and endTime, find the deposit and repay edge between the account and a loan, the transfers-in and transfers-out. Return ratioRepay (sum of all the edge1 over sum of all the edge2), ratioDeposit (sum of edge1 over sum of edge4), ratioTransfer (sum of edge3 over sum of edge4). Return -1 for ratioRepay if there is no edge2 found. Return -1 for ratioDeposit and ratioTransfer if there is no edge4 found.

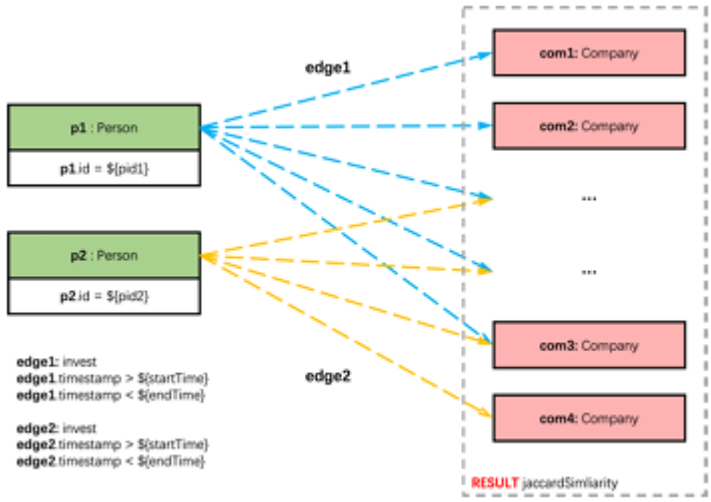
Note: There may be multiple loans that the given account is related to.

Parameters: `id1`, `threshold`, `startTime`, `endTime`, `truncationlimit`, `truncationOrder`

Result: `ratioRepay`, `ratioDeposit`, `ratioTransfer`

```
[CREATE PROCEDURE ComplexRead9(id1 int, threshold decimal, startTime
timestamp, endTime timestamp, truncationlimit int, truncationOrder string)
MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
        (:Account)-[e3:transfer
            where createTime>startTime and createTime<endTime]->
      (a:Account{id:id1})-[e4:transfer
            where createTime>startTime and createTime<endTime]->
      (:Account),
      (:Loan)-[e1:deposit
            where createTime>startTime and createTime<endTime]->
      (a)-[e2:repay
            where createTime>startTime and createTime<endTime]-> (:Loan)
return
      sum(e1.amt)/sum(e2.amt) as ratioRepay,
      case count(e4) when 0 then -1 else sum(e1.amt)/sum(e4.amt) end
            as ratioDeposit,
      case count(e4) when 0 then -1 else sum(e3.amt)/sum(e4.amt) end
            as ratioTransfer]
```

TCR10



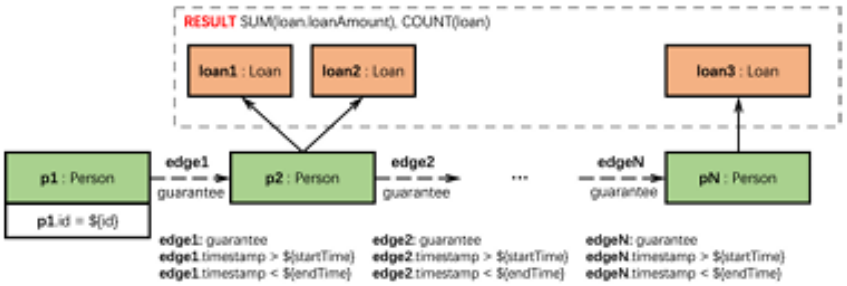
Given two Persons and a specified time window between startTime and endTime, find all the Companies the two Persons invest in. Return the Jaccard similarity between the two companies set. Return 0 if there is no edges foundc onnecting to any of these two persons.

Parameters: `pid1`, `pid2`, `startTime`, `endTime`

Result: `jaccardSimilarity`

```
[CREATE PROCEDURE ComplexRead10(pid1 int, pid2 int, startTime timestamp,
endTime timestamp)
MATCH
      (:Person{id:pid1})-[:invest
            where createTime>startTime and createTime<endTime]->
      (a:Company),
      (:Person{id:pid2})-[:invest
            where createTime>startTime and createTime<endTime]->
      (b:Company)
return
      cast(
         cardinality(collect(a) multiset intersect collect(b))
         /cardinality(collect(a) multiset union collect(b))
         as decimal(5,3))
      as jaccardSimilarity]
```

TRC11



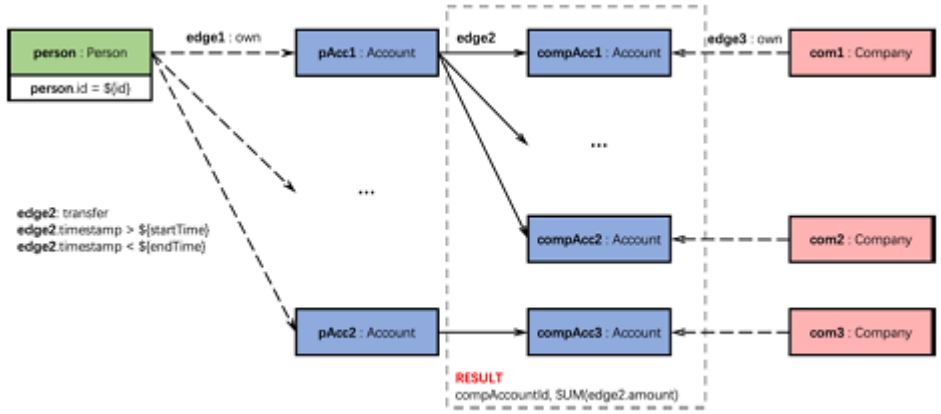
Given a Person and a specified time window between startTime and endTime, find all the persons in the guarantee chain until end and their loans applied. Return the sum of loan amount and the count of distinct loans.

Parameters: `id1`, `startTime`, `endTime`, `truncationlimit`, `truncationOrder`

Result: `sumLoanAmount`, `numLoans`

```
[CREATE PROCEDURE ComplexRead11(id1 int, startTime timestamp, endTime
timestamp, truncationlimit int, truncationOrder string)
      for x in MATCH
            truncating (guarantee (truncationOrder)=truncationLimit)
             (:Person{id:id1})
             [()-[:guarantee
                   where createTime>startTime and createTime<endTime]->
             (q:Person)]+ () return distinct unnest(q)
      MATCH (x)-[:apply]->(:Loan{id:lid,amount:amt})
      return
            sum(amt) as sumLoanAmount,
            count (distinct lid) as numLoans]
```

TCR12



Given a Person and a specified time window between startTime and endTime, find all the company accounts that s/he hast ransferred to. Return the ids of thecompanies' accounts and the sum of their transfer amount.

Parameters: `id1, startTime, endTime, truncationLimit, truncationOrder`

Result: `compAccountId, sumEdge2Amount`

Sort: `sumEdge2Amount desc, compAccountId`

```
[CREATE PROCEDURE ComplexRead12 (id1 int, startTime timestamp, endTime
timestamp, truncationLimit int, truncationOrder string)
MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
      (:Person{id:id1})-[:own]->(:Account)
      [()-[:transfer{amount:amt}
            where createTime>startTime and createTime<endTime]->
      (x:Account where exists
            (MATCH(x)<-[:own]-(:Company)))]+()
      return
            sum(amt) as sumEdge2Amount,
            x.id as compAccountId
            group by x.id
            order by (sumEdge2Amount desc, compAccountId)]
```

TSR1

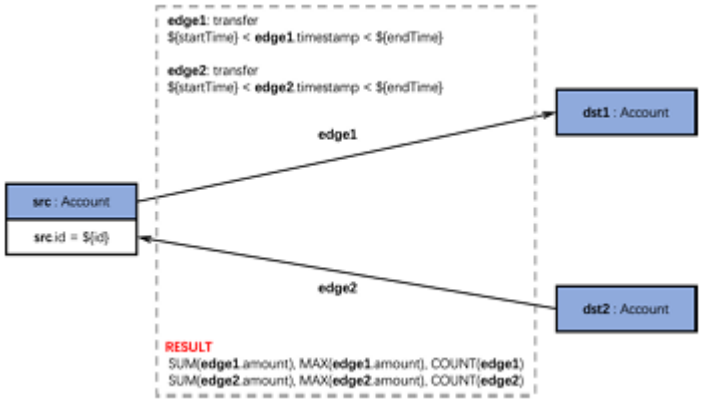
Given an id of an Account, find the properties of the specific Account

Parameters: `id1`

Result: `createTime`, `isBlocked`, `type`

```
CREATE PROCEDURE SimpleRead1 (id1 int) MATCH (x:Account{id:id1}) return
x.createTime, x.isBlocked, x.type
```

TSR2



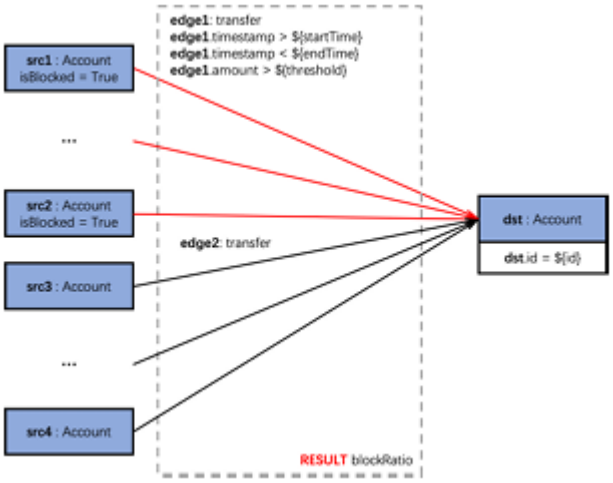
Given an account, find the sum and max of fundamount in transfer-ins and transfer-outs between them in a specific time range between startTime and endTime. Return the sum and max of amount. For edge1 and edge2, return -1 for the max (maxEdge1Amount and maxEdge2Amount) if there is no transfer.

Parameters: `id1`, `startTime`, `endTime`

Result: `sumEdge1Amount`, `maxEdge1Amount`, `numEdge1`, `sumEdge2Amount`, `maxEdge2Amount`, `numEdge2`

```
[CREATE PROCEDURE SimpleRead2 (id1 int, startTime timestamp, endTime timestamp)
MATCH
      (:Account)<-[:transfer{amount:amt1}
            where createTime>startTime and createTime<endTime]-
      (:Account{id:id1})-[:transfer{amount:amt2}
            where createTime>startTime and createTime<endTime]->
      (:Account)
return
      sum(amt1) as sumEdge1Ammount,
      max(amt1) as maxEdge1Amount,
      count(amt1) as numEdge1,
      sum(amt2) as sumEdge2Amount,
      max(amt2) as maxEdge2Amount,
      count(amt2) as numEdge2]
```

TSR3



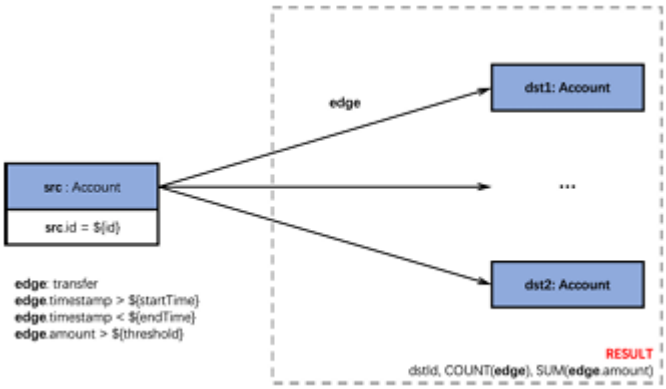
Given an Account, find the ratio of transfer-ins from blockedAccounts in all its transfer-ins in a specific time range between startTime and endTime. Return the ratio. Return -1 if there is no transfer-ins to the given account.

Parameters: `id1, threshold, startTime, endTime`

Result: `blockRatio`

```
[CREATE PROCEDURE SimpleRead3 (id1 int,threshold decimal, startTime
timestamp,endTime timestamp)
MATCH
      (:Account{id:id1})<-[:transfer {amount:amtb}
            where createTime>startTime and createTime<endTime]-
      (:Account{isBlocked:true}),
      (:Account{id:id})<-[:transfer {amount:amt}
            where createTime>startTime and createTime<endTime]-(:Account)
RETURN
      case when count(amt)=0 then -1 else sum(amtb)/sum(amt) end
            as ratioTransfer]
```

TSR4



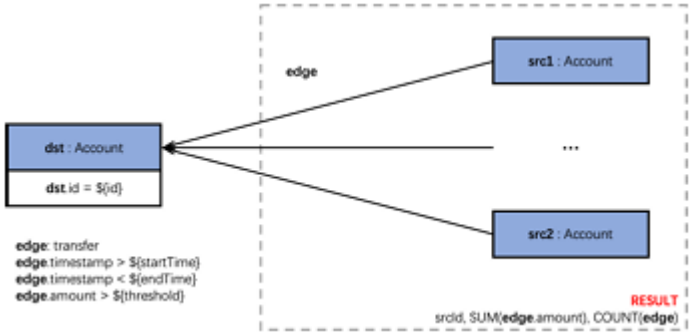
Given an account (src), find all thetransfer-outs (edge) from the src to a dst where the amount exceeds threshold in a specific time range between startTime and endTime. Return the count of transfer-outs and the amount sum.

Parameters: id1, threshold, startTime, endTime

Result: numEdges, sumAmount

```
[CREATE PROCEDURE SimpleRead4 (id1 int,threshold decimal,startTime
timestamp,endTime timestamp)
MATCH (:Account{id:id1})-[:transfer{amount:amt}
      where amt>=threshold and createTime>startTime and createTime
<endTime]->(:Account) return
      count(amt) as numEdges,
      sum(amt) as sumAmount]
```

TSR5



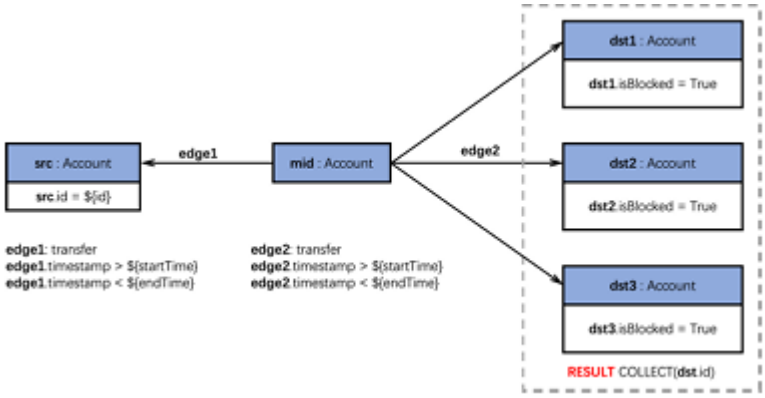
Given an account (dst), find all the transfer-ins (edge) from the src to a dst where the amount exceeds threshold in a specific time range between startTime and endTime. Return the count of transfer-ins and the amount sum.

Parameters: `id1`, `threshold`, `startTime`, `endTime`

Result: `numEdges`, `sumAmount`

```
[CREATE PROCEDURE SimpleRead5(id1 int, threshold decimal, startTime
timestamp, endTime timestamp) MATCH (:account{id:id1})
<-[:transfer{amount:amt}
      where amt>=threshold and createTime>startTime and
createTime<endTime]-(:Account)
return
      count(amt) as numEdges,
      sum(amt) as sumAmount]
```

TSR6



Given an Account (account), find all the blocked Accounts (dstAccounts) that connect to a common account (midAccount) with the given Account (account). Return all the accounts' id.

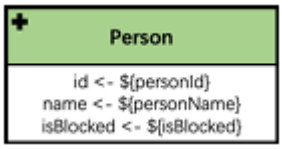
Parameters: `id1, startTime, endTime`

Result: dst, mid

Sort: `dstId`

```
[CREATE PROCEDURE SimpleRead6 (id1 int, startTime timestamp, endTime
timestamp) MATCH
      (:Account{id:id1})<-[:transfer
            where createTime>startTime and createTime<endTime]-
      (:Account{id:mid})
      <-[:transfer
            where createTime>startTime and createTime<endTime]-
      (:Account{id:dst})
RETURN mid,collect(dst) group by mid]
```

TW1

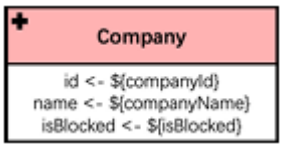


Add a Person.

Parameters: id1, name1, block1

```
[CREATE PROCEDURE Write1 (id1 int, name1 string, block1 boolean) CREATE
(:Person{id:id1,name:name1,isBlocked:block1,createTime:current_time})]
```

TW2

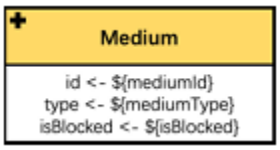


Add a Company.

Parameters: id1, name1, block1

```
[CREATE PROCEDURE Write2 (id1 int, name1 string, block1 boolean)
CREATE (:Company{id:id1,name:name1,isBlocked:block1,
createTime:current_time})]
```

TW3

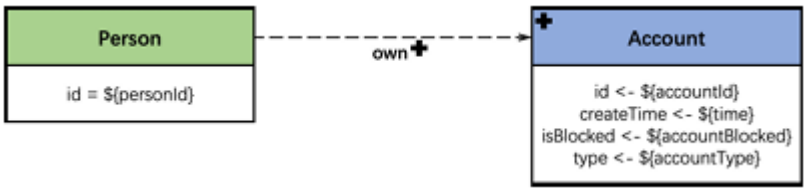


Add a Medium.

Parameters: id1, type1, block1

```
[CREATE PROCEDURE Write3 (id1 int, type1 string, block1 boolean)
CREATE (:Medium{id:id1,type:type1,isBlocked:block1,
createTime:current_time })]
```

TW4

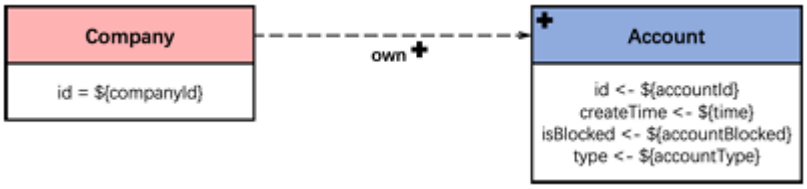


Add an Account owned by a Person.

Parameters: id1, accountId1, time1, type1, blocked

```
[CREATE PROCEDURE Write4 (id1 int, accountId1 int, time1 timestamp,
blocked boolean, type1 string)
MATCH (p:Person{id:id1})
      CREATE (p)-[:own]->
      (:Account {id:accountid1,
            createTime:time1,
            type:type1,
            isBlocked:blocked})]
```

TW5

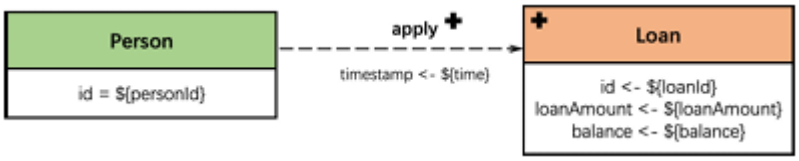


Add an Account and an own edge from the Company to the Account.

Parameters: id1, accountId1, time1, type1, blocked

```
[CREATE PROCEDURE Write5 (id1 int, accountId1 int, time1 timestamp,
blocked boolean, type1 string)
MATCH (c:Company{id:id1})
      CREATE (c)-[:own]->
      (:Account {id:accountid1,
            createTime:time1,
            type:type1,
            isBlocked:blocked})]
```

TW6

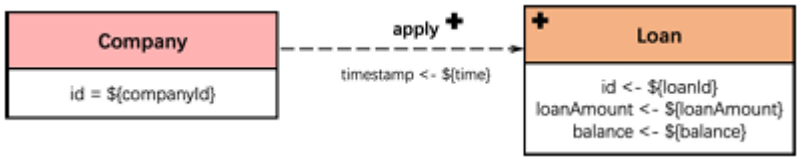


Add a Loan and add an apply edge from Person to Loan.

Parameters: id1, loanId, amt, bal, time

```
[CREATE PROCEDURE Write6(id1 int, loanId int, amt decimal, bal decimal,
time timestamp)
MATCH (p:Person{id:id1})
      CREATE (p)-[:apply {createTime:time}]->
            (:Loan{id:loanId,
                  loanAmount:amt,
                  balance:bal})]
```

TW7

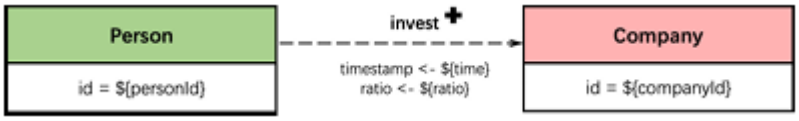


Add a Loan and add an apply edge from Company to Loan.

Parameters: id1, loanId, amt, bal, time

```
[CREATE PROCEDURE Write7 (id1 int, loanId int, amt decimal, bal decimal,
time timestamp)
MATCH (c:Company{id:id1})
      CREATE (c)-[:apply {createTime:time}]->
            (:Loan{id:loanId, loanAmount:amt, balance:bal})]
```

TW8

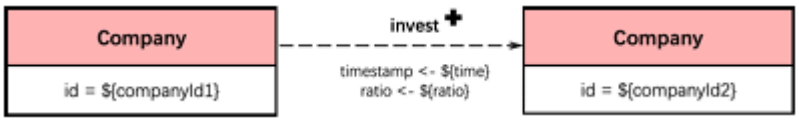


Add an invest edge from a Person to a Company.

Parameters: id1, companyId, time1, ratio1

```
[CREATE PROCEDURE Write8 (id1 int, companyId int, time1 timestamp, ratio1
decimal)
MATCH (p:Person{id:id1}),(c:Company{id:companyId})
      CREATE (p)-[:invest{createTime:time1,ratio:ratio1}]->(c)]
```

TW9

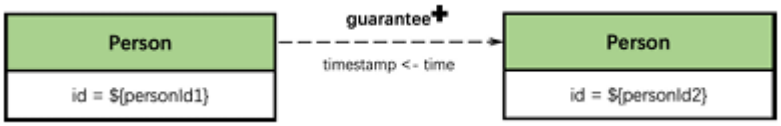


Add an invest edge from a Company to a Company.

Parameters: id1, id2, time1, ratio1

```
[CREATE PROCEDURE Write9 (id1 int, id2 int, time1 timestamp, ratio1
decimal)
MATCH (c:Company{id:id1}),(d:Company{id:id2})
      CREATE (c)-[:invest{createTime:time1, ratio:ratio1}]->(d)]
```

TW10

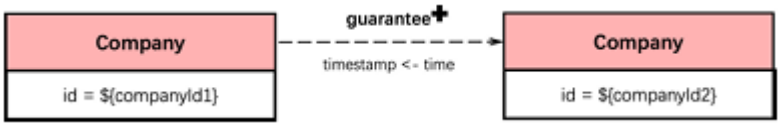


Add a guarantee edge from a Person to another Person.

Parameters: id1, id2, time1

```
[CREATE PROCEDURE Write10(id1 int, id2 int, time1 timestamp)
MATCH (p:Person{id:id1}),(q:Person{id:id2})
      CREATE (p)-[:guarantee{createTime:time1 }]->(q)]
```

TW11

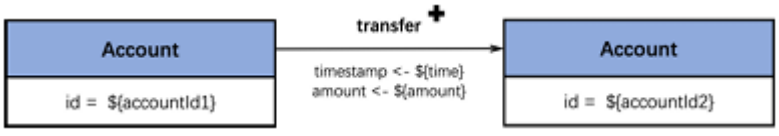


Add a guarantee edge from a Company to another Company.

Parameters: id1, id2, time1

```
[CREATE PROCEDURE Write11 (id1 int, id2 int, time1 timestamp)
MATCH (c:Company{id:id1}),(d:Company{id:id2})
      CREATE (c)-[:guarantee{createTime:time1 }]->(d)]
```

TW12



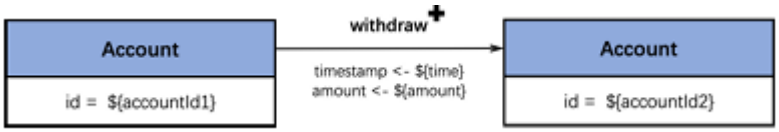
Add a transfer edge from an Account to another Account.

Parameters: id1, id2, time1, amt

```
[CREATE PROCEDURE Write12 (id1 int, id2 int, time1 timestamp, amt decimal)
MATCH (a:Account{id:id1}),(b:Account{id:id2})
```

```
      CREATE (a)-[:transfer{createTime:time1, amount:amt}]->(b)]
```

TW13



Add a withdraw edge from an Account to another Account.

Parameters: id1, id2, time1, amt

```
[CREATE PROCEDURE Write13 (id1 int, id2 int, time1 timestamp, amt decimal)
MATCH (a:Account{id:id1}),(b:Account{id:id2})
      CREATE (a)-[:withdraw{createTime:time1, amount:amt}]->(b)]
```

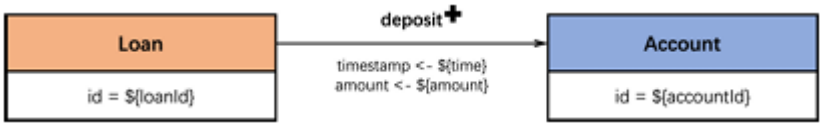
TW14

Add a repay edge from an Account to a Loan.

Parameters: id1, id2, time1, amt

```
[CREATE PROCEDURE Write14 (id1 int, id2 int, time1 timestamp, amt decimal)
MATCH (a:Account{id:id1}),(b:Loan{id:id2})
      CREATE (a)-[:repay{createTime:time1, amount:amt}]->(b)]
```

TW15

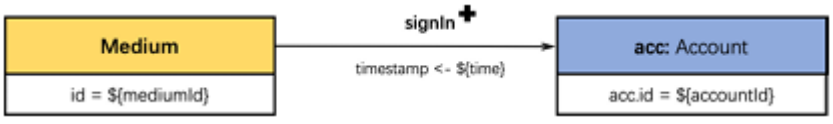


Add a deposit edge from a Loan to an Account.

Parameters: id1, id2, time1, amt

```
[CREATE PROCEDURE Write15(id1 int, id2 int, time1 timestamp, amt decimal)
MATCH (a:Loan{id:id1}),(b:Account{id:id2})
      CREATE (a)-[:deposit{createTime:time1, amount:amt}]->(b)]
```

TW16

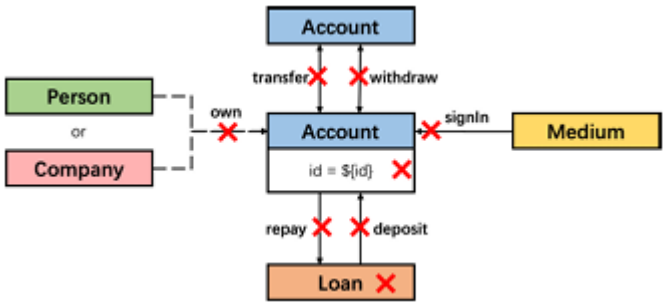


Add a signIn edge from a Medium to an Account.

Parameters: id1, id2, time1

```
[CREATE PROCEDURE Write16 (id1 int, id2 int, time1 timestamp)
MATCH (m:Medium{id:id1}),(a:Account{id:id2})
      CREATE (m)-[:signIn{createTime:time1 }]->(a)]
```

TW17

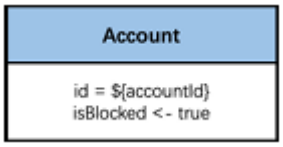


Given an id, remove the Account, and remove the related edges including own, transfer, withdraw, repay, deposit, signIn. Remove the connected Loan vertex in cascade.

Parameter: `id1`

```
CREATE PROCEDURE Write17 (id1 int) MATCH (a:Account{id:id1}) detach delete
a
```

TW18

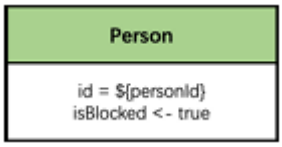


Set an Account's isBlocked to True

Parameter: `id1`

```
CREATE PROCEDURE Write18 (id1 int) MATCH(a:Account{id:id1}) set
a.isBlocked = true
```

TW19

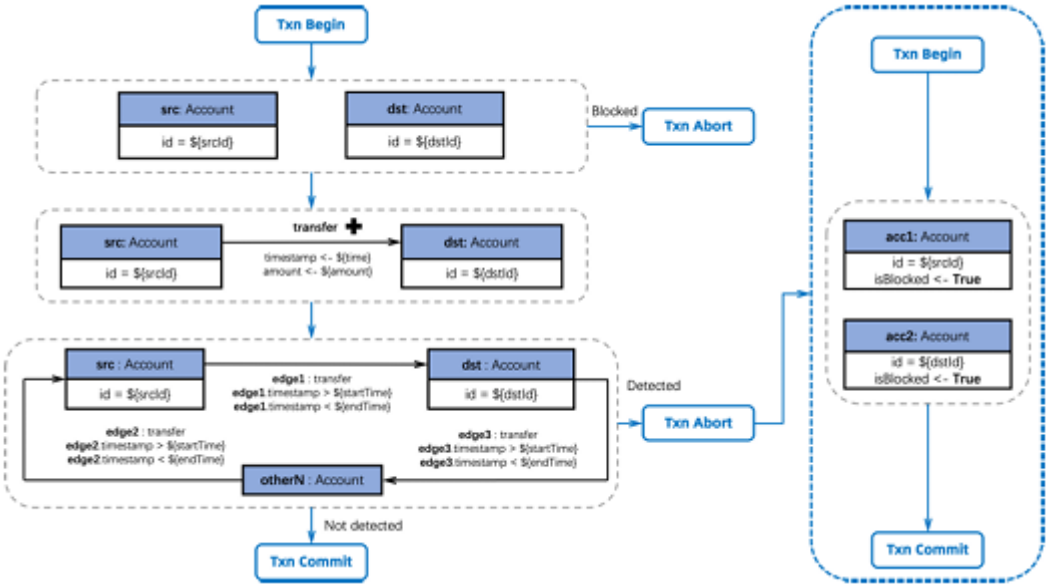


Set an Person's isBlocked to True

Parameter: `id1`

```
CREATE PROCEDURE Write19 (id1 int) MATCH(p:Person{id:id1}) set p.isBlocked
= true
```

TRW1



The workflow of this read write query contains at least one transaction. It works as:

In the very beginning, read the blocked status of related accounts with given ids of two src and dst accounts. The transaction aborts if one of them isblocked. Move to the nextstep if none is blocked.

Add a transfer edge from src to dst inside a transaction. Given a specified time window between startTime and endTime, find the other accounts which received money from dst and transferred money to src in a specifictime.

Transaction aborts if a new transfer cycle is formed, otherwise the transaction commits.

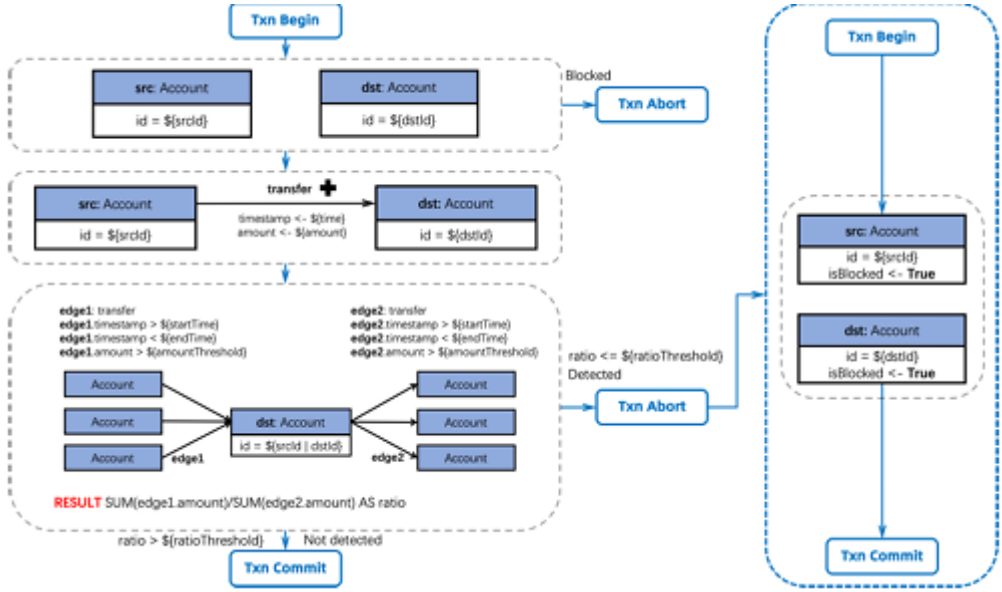
If the last transaction aborts, mark the src and dst accounts as blocked in another transaction.

(NB: *Transaction semantics in Pyrrho are desscribed differently but the statement below implements the above in a single transaction.*)

Parameters: `srcId`, `dstId`, `time`, `amount`, `startTime`, `endTime`

```
[CREATE PROCEDURE ReadWrite1(srcId int, dstId int, time timestamp,
   amt decimal, startTime timestamp, endTime timestamp)
MATCH (src:Account{id:srcId,isBlocked:false}),
      (dst:Account{id:dstId,isBlocked:false})
if exists (MATCH trail p=(dst) [()-[x:transfer
      where createTime>startTime and createTime<endTime
      and (cardinality(p)=0
            or p.x[cardinality(p)-1].createTime<createtime)]->()]+(src))
   then set src.isBlocked=true; set dst.isBlocked=true
   else CREATE(src)-[:transfer{createTime:current_time,amount:amt}]->(dst)
end if]
```

TRW2

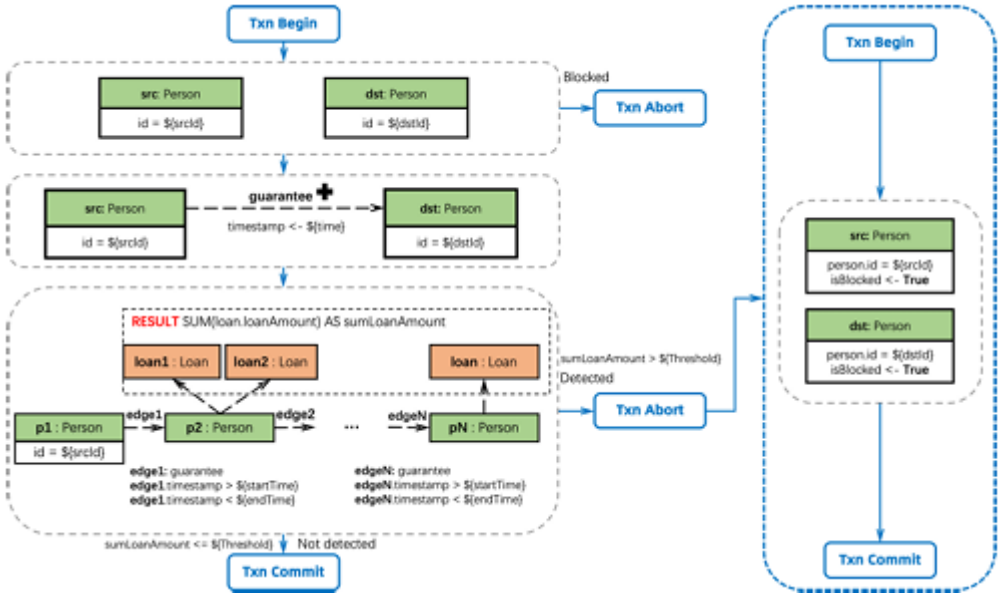


The workflow of this read write query contains at least one transaction. It works as:
In the very beginning, read the blocked status of related accounts with given ids of two src and dst accounts. The transaction aborts if one of them is blocked. Move to the next step if none isblocked.
Add a transfer edge from src to dst inside a transaction. Given a specified time window between startTime and endTime, find all the transfer-in and transfer-out whose amount exceeds amountThreshold. Transaction aborts if the ratio of transfers-in/transfers-out amount exceeds a given ratioThreshold, both for the src and dst account. Otherwise the transaction commits.
If the last transaction aborts, mark the src and dst accounts as blocked in another transaction.

Parameters: srcId, dstId, time1, amt, amountThreshold, startTime, endTime, ratioThreshold, truncationLimit, truncationOrder

```
[CREATE PROCEDURE ReadWrite2(srcId int, dstId int, time1 timestamp,
amt decimal, amountThreshold decimal, startTime timestamp, endTime
timestamp, ratioThreshold decimal, truncationLimit int, truncationOrder
string)
MATCH (src:Account{id:srcId,isBlocked:false}),
                 (dst:Account{id:dstId, isBlocked:false})
 if exists(MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
      (:Account)-[:transfer{amount:amtIn}
            where createTime>startTime and createTime<endTime]->(src)
      -[:transfer{amount:amtOut}
      where createTime>startTime and createTime<endTime]-> (:Account)
   | (:Account)-[:transfer{amount:amtIn}
            where createTime>startTime and createTime<endTime]->(dst)
      -[:transfer{amount:amtOut}
      where createTime>startTime and createTime<endTime]-> (:Account)
  return sum(amtOut)<>0 and sum(amtIn)/sum(amtOut)>ratioThreshold)
 then
       set src.isBlocked = true;
       set dst.isBlocked = true
 else
     CREATE(src)-[:transfer{createTime:time1,amount:amt}]->(dst)
 end if]
```

TRW3



The workflow of this read write query contains at least one transaction. It works as:

In the very beginning, read the blocked status of related persons with given ids of two src and dst persons. The transaction aborts if one of them is blocked. Move to the next step if none is blocked. Add a guarantee edge between the src and dst persons inside a transaction. Given a specified time window between startTime and endTime, find all the persons in the guarantee chain of until end and their loans applied. Detect if a guarantee chain pattern formed, only for the src person. Calculate if the amount sum of the related loans in the chain exceeds a given threshold. Transaction aborts if the sum exceeds the threshold. Otherwise the transaction commits. If the last transaction aborts, mark the src and dst persons as blocked in another transaction.

Parameters: srcId, dstId, time, threshold, startTime, endTime, truncationLimit, truncationOrder

```
[CREATE PROCEDURE ReadWrite3(srcId int, dstId int, time timestamp,
threshold decimal, startTime timestamp, endTime timestamp, truncationLimit
int, truncationOrder string)
MATCH(src:Person{id:srcId,isBlocked:false}),
      (dst:Person{id:dstId,isBlocked:false})
   if exists (MATCH
      truncating (transfer (truncationOrder)=truncationLimit)
        (p:Person)-[:apply]->(:Loan{amount:amt})
      where p in (MATCH (src)[()-[:guarantee where createTime>startTime
         and createTime<endTime]->(q)]+() return sum(amt)>threshold))
   then
      set src.isBlocked = true;
      set dst.isBlocked=true
   else
   CREATE(src)-[:guarantee{createTime:current_time}]->(dst)
   end if]
```