Parallel Programming Framework for Large Batch Transaction Processing on Scale-out Systems

Kazuaki Ishizaki, Toshio Suganuma, Akira Koseki,
Yohei Ueda, Ken Mizuno, Daniel Silva,
Hideaki Komatsu, Toshio Nakatani

IBM Research – Tokyo

Summary of Talk

Our Goal

Build Transactional Applications ***with Smaller Number of Lines*** *on a Scale-out System*

- Design Java programming framework for distributed transactions
  - Data partitioning API
  - Transaction scope API
  - Our framework reduced the lines of code by 66% for writing an application compared to naïve programming using MPI and JDBC

Outline

- Summary
- Motivation of This Research
- Our Programming Framework
- Evaluation of Ease of Programming
- Conclusion

Large Data Processing on Scale-out Becomes Reality

- Computer is becoming cheaper

- Good programming frameworks make programming easy
  - MapReduce
  - Hadoop
  - …

Large Data Processing on Scale-out Becomes Reality

- Computer is becoming cheaper
- Good programming frameworks make programming easy
  - MapReduce
  - Hadoop
  - …

Only if transactions are not required

What is Transaction ?

- Sequence of operations must be atomic
  - Ex. Bank account and travel reservation

```
Transaction A
   balance = X.get()
   X.withdraw(balance/2)
   Y.deposit(balance/2)
```

```
Transaction B
   balance = X.get()
   X.withdraw(balance/2)
   Y.deposit(balance/2)
```

X = $400, Y = $100

		balance = X.get()	$400
balance = X.get()	$400		
		X.withdraw(balance/2)	$200
		Y.deposit(balance/2)	$300
X.withdraw(balance/2)	$0		
Y.deposit(balance/2)	$500		

If a transaction is not executed atomically, results are incorrect

What is Transaction ?

- Sequence of operation must be atomic
  - Ex. Bank account and travel reservation

```
Transaction B
  balance = X.get()
  X.withdraw(balance/2)
```

```
Transaction B
  balance = X.get()
  X.withdraw(balance/2)
```

The explosion of online services such as internet banking, internet shopping, and …
increases the number of transactions rapidly

Many transactions should be processed
using scale-out approach as non-transactions do

Difficulty for Processing Transactions on Scale-out

- Data partition on multiple nodes
- Data transfer among multiple nodes

} Common issues on scale-out

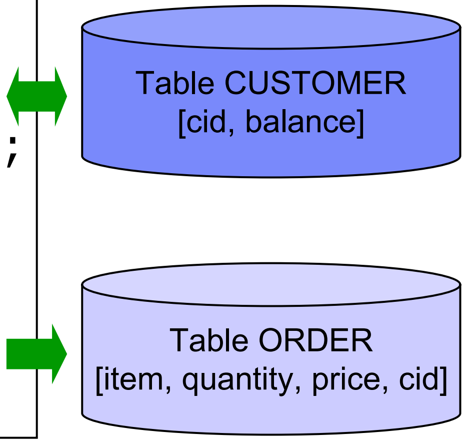
- Distributed transactions among multiple nodes

Transaction Scenario

1. Input { customer ID (cid), item, quantity, price }
2. Withdraw price from customer’s balance in table CUSTOMER
3. Insert (item, quantity, price, customer ID) into table ORDER

On single node

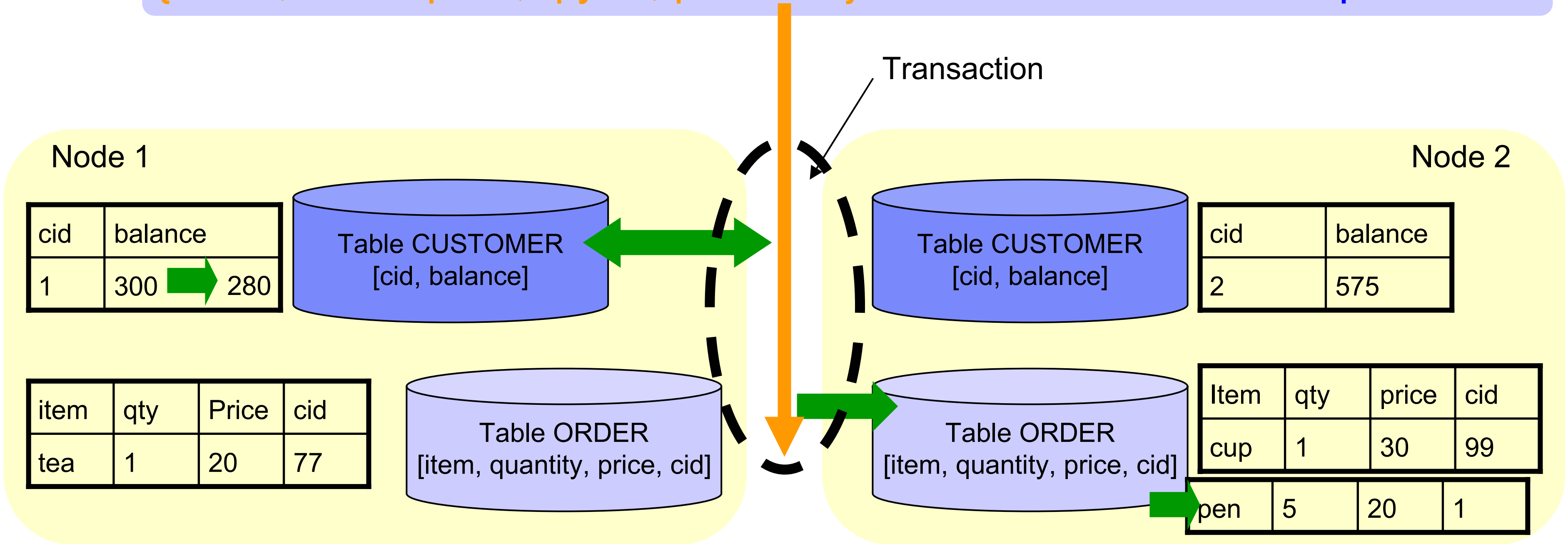
```
// In is  4-tuple {cid, item, qty, price};
transaction(Input in) {       // 1.
  // 2.
  SELECT balance from CUSTOMER where id = in.cid;
  newb = balance - in.price;
  UPDATE CUSTOMER set balance = newb where id = in.cid;

  // 3.
  INSERT into ORDER values
    (in.item, in.qty, in.price, in.cid);
}
```



Tables on Multiple Nodes are Accessed in a Transaction

- Table CUSTOMER is partitioned along **cid**
  - If cid = 1, store into node 1. If cid = 2, store into node 2
- Table ORDER is partitioned along **item**
  - If item is “tea”, store into node 1. Otherwise, store into node 2.

{cid=1, item=“pen”, qty=5, price=20} involves access to multiple nodes



Node 1:

cid	balance
1	300 → 280

item	qty	Price	cid
tea	1	20	77

Node 2:

cid	balance
2	575

Item	qty	price	cid
cup	1	30	99
pen	5	20	1

Naïve Programming Needs Lots of Lines

- Two complicated and buggy parts
  - Data transfer (orange) using Java-MPI
  - Distributed transaction management (green) – using JDBC

```
// skipped the rare paths such as rollback and exception handling
class Input { int cid; String item; int qty; double price;}
class Cust { int cid; double balance; }
{
  int doWorker() {
    List<Input> inp = new ArrayList<Input>(); // create Input data
    inp.add(new Input(2, "cup", 3, 90));
    inp.add(new Input(1, "pen", 5, 20));
    inp.add(new Input(2, "tea", 4, 80));

    List<Integer> destNode1 = new ArrayList<Integer>();
    for (Input in : inp) destNodes1.add(Partition.getNode1(in));
    CommLib.sendListBulk(inp, destNodes1);

    final Tx tx = TransactionLib.openTx();
    Thread t1 = new Thread() {
      public void run() {
       while (true) {
        List<Input> inp1 = new ArrayList<Input>();
          Int sendNode = CommLib.recvListBulk(inp1);
          If (sendNode == -1) break;    // no more data
          Connection co1 =
 DriverManager.getConnection("jdbc:db2://localhost:999/CUST", "user", "pw");
          STx stx1 = TransactionLib.openSTx(co1);
          doSubTxn1(co1, inp1);

        List<Integer> destNodes2 = new ArrayList<Integer>();
          for (Input in : inp) destNodes2.add(Partition.getNode2(in));
          CommLib.sendListBulk(inp, destNodes2);

          CommLib.BwaitData(destNodes2, PHASE1_Complete);

          CommLib.BsendData(destNodes2, PHASE2_Start);
          CommLib.BwaitData(destNodes2, PHASE2_Complete),
          stx1.commit();
          stx1.close();
        }
      }
    }.start()       // Execute subtransaction
    Thread t2 = new Thread() {
      public void run() {
        while (true) {
        List<Input> inp2 = new ArrayList<Input>();
          int senderNode = CommLib.recvListBulk(inp2);
          If (senderNode == -1) break;  // no more data
          Connection co2 =
 DriverManager.getConnection("jdbc:db2://localhost:999/ORDER", "user", "pw");
          STx stx2 = TransactionLib.openSTx(co2);
          doSubTxn2(co2, inp2);
```

```
          CommLib.sendData(senderNode, PHASE1_Complete);
          CommLib.waitData(senderNode, PHASE2_Start);
          stx2.commit();
          stx2.close();
          CommLib.sendData(senderNode, PHASE2_Complete);
        }
      }
    }.start();      // Execute subtransaction

    { t2.join(); t1.join(); }
    tx.commit();
    tx.close();
  }
  void doSubTx1(Connection con, List<Input> inp) {
    PreparedStatement ps1, ps2;
    ps1 = con.prepareStatement("SELECT balance from CUST where ? = in.cid");
    ps2 = con.prepareStatement("UPDATE CUST set balance = ? where id = in.cid");
    for (Input in : inp) {
      ps1.setInt(1, in.cid);
      ResultSet rs = ps1.executeQuery(); rs.getNext();
      double newval = (double)(rs.getDouble(1) - in.price);
      ps2.setDouble(1, newval);
      ps2.executeStatement();
      rs.close();
    }
    ps2.close(); ps1.close();
  }
  void doSubTx2(Connection con, List<Input> inp) { // insert a record into ORDER
    PreparedStatement ps1;
    ps1 = con.prepareStatement("INSERT into ORDER value (? ? ? ?)");
    for (Input in: inp) {
      ps1.setString(1, in.item); ps1.setInt(2, in.qty);
      ps1.setDouble(3, in.price); ps1.setInt(4, in.cid);
      ps1.executeStatement();
    }
    ps1.close();
  }
}
static class Partition {
  int getNode1(Input in) { return in.cid; }
  int getNode2(Input in) { “pen”.equals(in.item) ? 2 : “cup”.equals(in.item) ? 2 : 1; }
}
```

+ Communication and distributed transaction libraries (2000 lines)

Our Framework Simplify a Complicated Code

- Two complicated parts can be simplified
  - Data transfer (orange)
  - Distributed transaction management (green)

```
class Input { int cid String item; int qty; double price; }
class Cust { int id; double balance; }

int doWorker(final Job job, final PD CUSTTbl, final PD ORDERTbl,
 final PS<Input> inp,         final PS<Input> insum,
 final PS<Cust> c1, final PS<Cust> c2) {
  inp.add(new Input(2, "cup", 3, 90));
  inp.add(new Input(1, "pen", 5, 20));
  inp.add(new Input(2, "tea", 4, 80));

  new Tx(job) {
    public void scope(final Tx tx) {
      STx stx1 = new STx(inp, PA1.class, inp, CUSTTbl, inp, insum, c1, c2) {
        public void scope(final STx stx) throws Exception {
          // read c1.balance from CUST
          ReadCustTbl.exec(stx, inp, CUSTTbl, c1);
          // insum.price += inp.qty * inp.price
          ReduceInput.exec(stx, inp, insum);
          // c2.balance = insum.price * c1.balance
          JoinCust.exec(stx, c1, insum, c2);
          // update c2.balance
          WriteCUST.exec(stx, c2, CUSTTbl);
        }
      };
      STx stx2 = new STx(inp, PA2.class, null, ORDERtbl, inp) {
        public void scope(final STx stx) {
          // insert a record into ORDER
          WriteORDER.exec(stx, inp, ORDERtbl);
        }
      };
      start(stx1, stx2);
    }
  }.start();
}
```

Database access + application logic

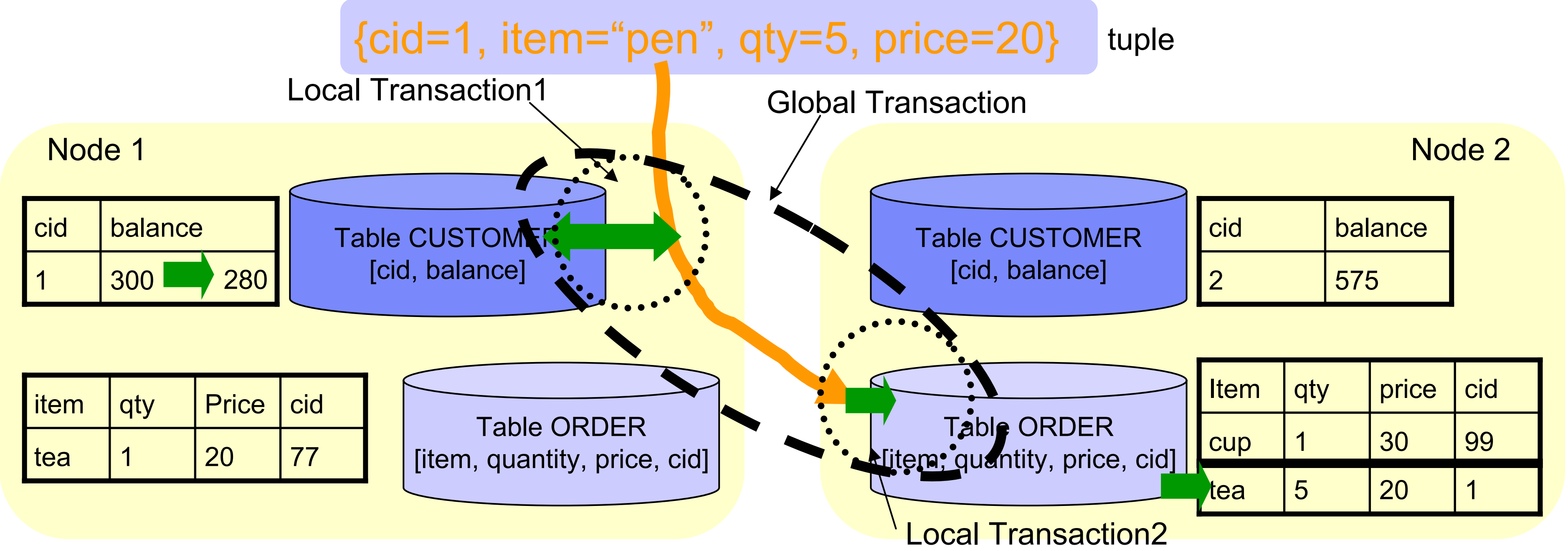
```
static class PDforCUST extends PD {
String[] properties = {“CUST”, “admin”, “pw”};
  String getReadSQL() {
    return “SELECT id, balance from ”+dbName()+” where cid = ?”;
  }
  void setReadParam(PreparedStatement s, List p) {
    s.setInt(s, 1, p.get(0));
  }
  String getWriteSQL() {
    return “UPDATE ”+dbName()+” set balance = ? where cid = ?”;
  }
  String setWriteParam(PreparedStatement s, List p) {
    s.setDouble(s, 1, p.get(0)); s.setInt(s, 2, p.get(1));
  }
}
static class PDforORDER extends PD {
  String[] properties = {“ORDER”, “admin”, “pw”};
  String getWriteSQL() { return “INSERT into ”+dbName()+” values (?, ?, ?, ?)”; }
  String setWriteParam(PreparedStatement s, List p) {
    s.setString(s, 1, p.get(0)); s.setInt(s, 2, p.get(1));
    s.setDouble(s, 3, p.get(2)); s.setInt(s, 4, p.get(3));
  }
}
static class ReadCustTbl extends Read<Input, Cust> {
  void select(Input in) {
    List p = new ArrayList(); p.add(in.cid); emitParam(p);
  }
  void read(ResultSet rs) {
    Cust out = new Cust(rs.getInt("id"), rs.getDouble("balance")); emit(out);
  }
}
static class WriteCUST extends Write<Input> {
  public void write(Input in) {
    List p = new ArrayList();
    p.add(in.balance); p.add(in.cid); emitParam(p);
  }
}
static class WriteORDER extends Write<Input> {
  void write(Input in) {
    List p = new ArrayList(); p.add(in.item); p.add(in.qty);
    p.add(in.price); p.add(in.cid); emitParam(p);
  }
}
static class ReduceInput extends Reduce<Input> {
  void reduce(Iterator<Input> itr) {
    double sum = 0;
    Input in;
    while (itr.hasNext()) { in = itr.next(); sum += in.qty * in.price; }
    Input out = new Input(in.cid, "", 0, sum); emit(out);
  }
}
static class JoinCust extends Join<Cust, Input, Cust> {
  boolean predicate(Cust c, Input in) { return (c.id == in.cid); }
  void join(Cust c, Input in) {
    Cust out = new Cust(c.id, c.balance - in.price);
    emit(out);
  }
}
```

Outline

- Summary
- Motivation of This Research
- Our Programming Framework
- Evaluation of Ease of Programming
- Conclusion

How Program is Executed

- Global transaction is initiated
  - Tuple is transferred to node 1
  - Local transaction 1 is initiated and committed
  - Tuple is transferred to node 2
  - Local transaction 2 is initiated and committed
- Global transaction is committed
  - Here, all of changes are visible to others



cid	balance
1	300 → 280

item	qty	Price	cid
tea	1	20	77

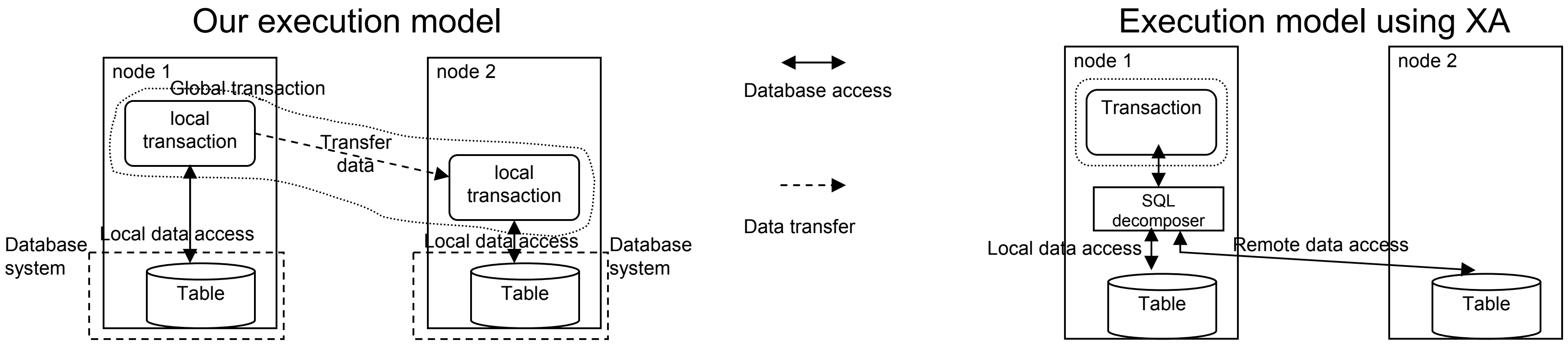
cid	balance
2	575

Item	qty	price	cid
cup	1	30	99
tea	5	20	1

Nested Transaction – Global and Local

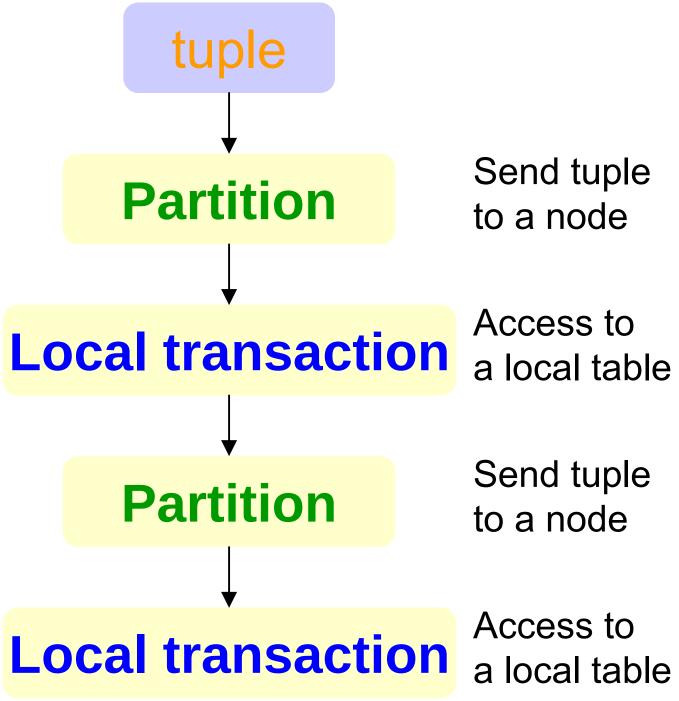
- Nested transaction consists of
  - global transaction includes **data transfer** among nodes
  - local transaction includes **access to only local database** within a node

- Tuples are transferred to other node along data partitioning API among local transactions if tuples require to access data on another node
  - Index data is smaller than extracted data



Two APIs that Programmers Use

- Data partition API (in **green**)
  - Map a tuple to a node (detail in next page)
- Transaction scope API (in **blue**)
  - Specify transaction scope



tuple

Partition

Send tuple to a node

Local transaction

Access to a local table

Partition

Send tuple to a node

Local transaction

Access to a local table

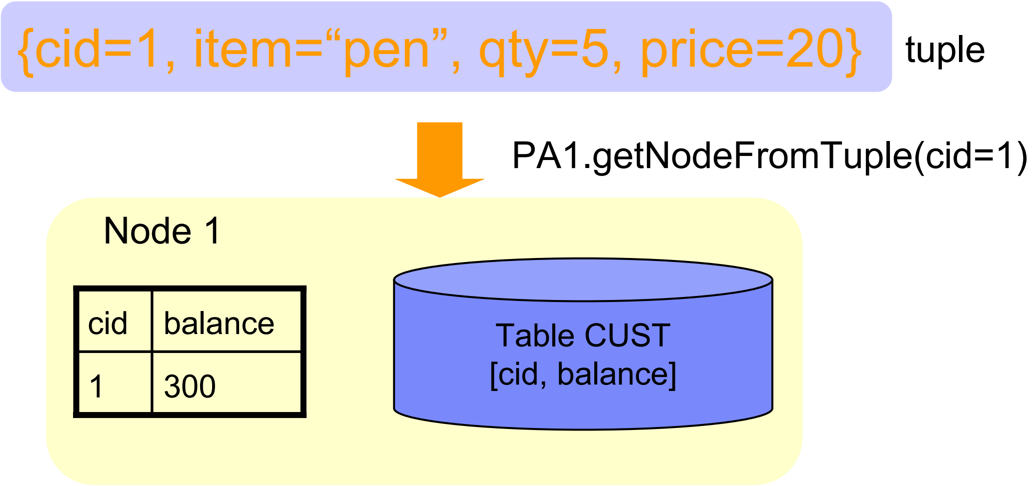
```
new GlobalTx(job) { // global transaction
  txScope(tx) {
   // Transfer tuples using PA1
   stx1 = new LocalTx(..., PA1.class, ...) {
    void txScope(stx) { // local transaction
      // Perform computation to access local table
    }
   };
   // Transfer tuples using PA2
   stx2 = new LocalTx(..., PA2.class, ...) {
    void txScope(stx) { // local transaction
      // Perform computation to access local table
    }
   };
   start(stx1, stx2);
  }
 }.start();
}
```

Data Partition API Maps a Tuple to a Node

- Programmer specifies a mapping of a tuple to a node using the method getNodeFromTuple()
  - Mapping can be generated from deployed database

Example usage of data partition API

```
class PA1 extends Partitioner<Input> {
  int getNodeFromTuple(Input inp) {
    return inp.cid % numberOfProcessors;
  }
}
```



{cid=1, item="pen", qty=5, price=20} tuple

PA1.getNodeFromTuple(cid=1)

Node 1

cid	balance
1	300

Table CUST
[cid, balance]

Outline

- Summary
- Motivation of This Research
- Our Programming Framework
- Evaluation of Ease of Programming
- Conclusion

Reduced Lines of Code by 66% (2,964 to 1,001 lines)

- Target program
  - Batch transaction processing based on TPC-C
    - 2 global transactions, 4 local transactions

Comparison of Lines of Code for the same program

Components	Ours	Naïve (MPI and JDBC)
Application Logic	220	270
Data access to database table	740	630
Partitioner	24	15
Transaction management	17	1450
Data transfer	0	599
Total	**1,001**	**2,964**

Outline

- Summary
- Motivation of This Research
- Our Programming Framework
- Evalution of Ease of Programming
- Conclusion

Our Accomplishment

Succeeded in building Transactional Applications with smaller number of lines on a Scale-out System

- Designed Java programming framework for distributed transactions
    - Data partition API
    - Transaction scope API
    - Our framework reduced the lines of code by 66%