# A Prolegomenon for Predictive Modeling of Parallel OLTP Systems

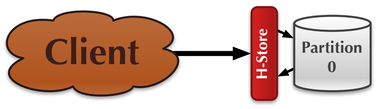

Andrew Pavlo
Saurya Velagapudi
Stanley Zdonik
**Brown University**

Evan P.C. Jones
Samuel Madden
**MIT CSAIL**


## Project Goal:

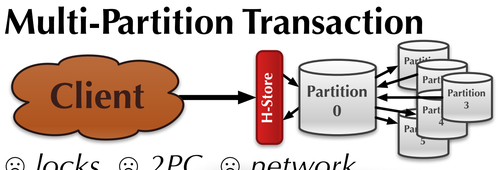
For an OLTP workload containing distributed transactions, one would like to predict when transactions are not single-partitioned. We are exploring optimizations for these types of workloads for the **H-Store** parallel, main-memory DBMS using partitioned predictive Markov models.

**Single-Partition Transaction**




*No locks, No 2PC, No network*

**Multi-Partition Transaction**




*☹ locks, ☹ 2PC, ☹ network*

## Inputs:

### ▪ Stored Procedures

H-Store requires that administrators pre-define stored procedures comprised of Java control code intermixed with parameterized SQL statements.

```
public class RecallOrderLine extends StoredProcedure {
  public final String GetOrders =
    "SELECT * FROM ORDERS" +
    " WHERE ((O_ALL_LOCAL = 1 AND O_W_ID = ?)" +
    "    OR (O_ALL_LOCAL = 0)) AND O_ENTRY_D >= ?";
  public final String GetOrderLines =
    "SELECT * FROM ORDER_LINE" +
    " WHERE OL_O_ID = ? AND OL_W_ID = ?" +
    "   AND OL_SUPPLY_W_ID = ? AND OL_I_ID = ?" +
    "   AND OL_DELIVERY_ID IS NULL";
  public final String UpdateCustomer =
    "UPDATE CUSTOMER SET C_BALANCE -= ?" +
    " WHERE C_ID = ? AND C_W_ID = ?";
  public final String UpdateStock =
    "UPDATE STOCK SET S_QUANTITY = 0 " +
    " WHERE S_W_ID = ? AND S_I_ID = ?";

  public void run(long s_w_id, long i_id, Date o_date) {
    Table orders = executeSQL(GetOrders, s_w_id, o_date);
    for (Row record : orders.getRows()) {
      long o_id = record.get("O_ID");
      long o_w_id = record.get("O_W_ID");
      long o_c_id = record.get("O_C_ID");
      Table order_lines = executeSQL(GetOrderLines,
                                     o_id, o_w_id, s_w_id, i_id);
      double refund = 0.0d;
      for (Row ol : order_lines.getRows()) {
        refund += ol.get("OL_QTY") * ol.get("OL_AMT");
      }
      executeSQL(UpdateCustomer, refund, o_c_id, o_w_id);
    }
    executeSQL(UpdateStock, s_w_id, i_id);
} }
```

### ▪ Sample Workload Trace

The administrator must also provide a sample workload trace for the target application. A trace record consists of (1) the stored procedure executed (2) the queries executed in the transaction, and (3) the input parameters.

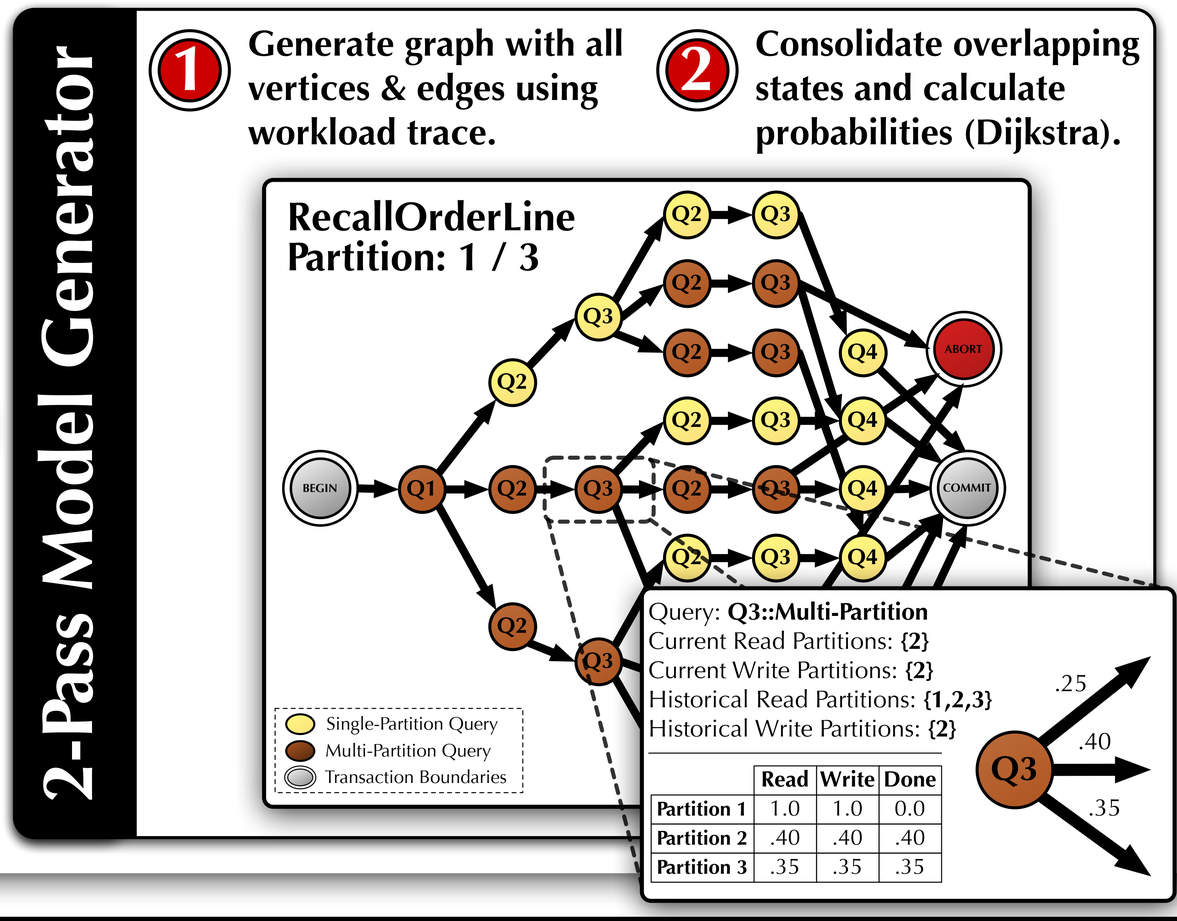
```
RecallOrderLine(1001,123,2010-01-18)
  exec GetOrders(1001,2010-01-18)
  exec GetOrderLines(555,1002,1001,123)
  exec UpdateCustomer($10,19811,1002)
  exec UpdateStock(1001,123)
  commit

RecallOrderLine(1001,123,2010-01-18)
  exec GetOrders(1001,2010-01-18)
  exec GetOrderLines(555,1002,1001,123)
  exec UpdateCustomer($10,19811,1002)
  exec GetOrderLines(555,1002,1001,123)
  exec UpdateCustomer($10,19811,1002)
  exec UpdateStock(1001,123)
  commit

RecallOrderLine(1001,123,2010-01-18)
  exec GetOrders(1001,2010-01-18)
  exec GetOrderLines(555,1002,1001,123)
  exec UpdateCustomer($10,19811,1002)
  exec UpdateStock(1001,123)
  commit
```

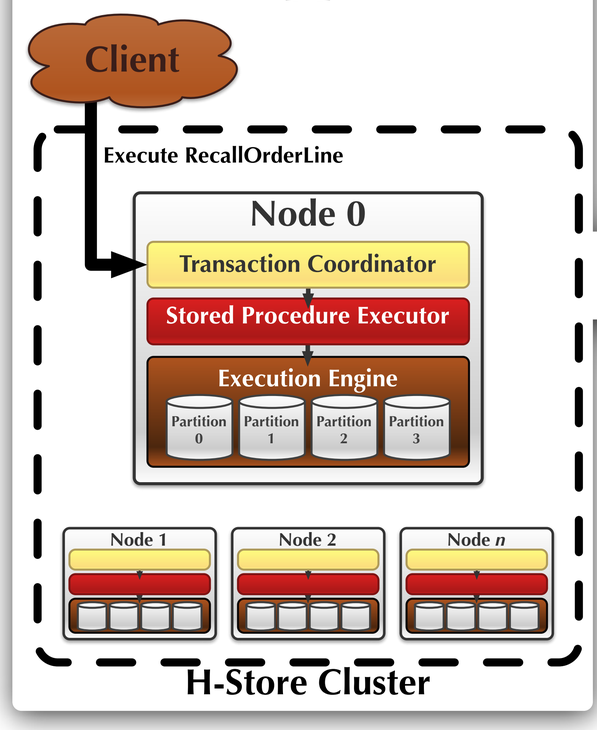
## OLTP Markov Models:

Using the workload trace, the modeler generates a separate **Markov model** for each stored procedure. Each graph is specific to the initial partition in the database that can invoke the stored procedure. The vertices represent the potential execution states of a transaction, including (1) the executed query and (2) what partitions have been read/written up to that point. The edges weights represent the probabilities of one state transitioning to another.

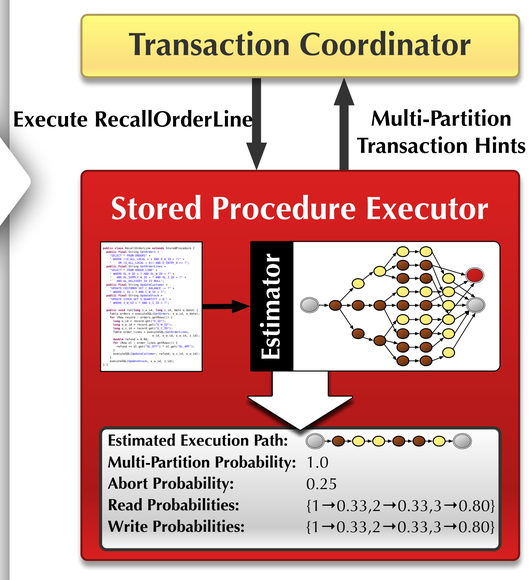



## Runtime Optimizations:

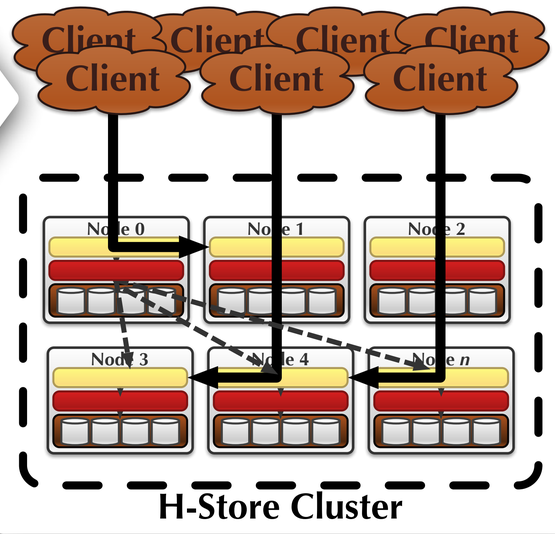
**1** An OLTP client sends a transaction request to a single H-Store execution node in the cluster.




**2** The node's transaction coordinator invokes its procedure executor and estimates the execution path of the new transaction.




**3** The estimator tracks state and sends the coordinator the relevant partition ids. The coordinator informs others on whether it is safe to execute new transactions on those partitions before the original transaction commits.




**4** When the transaction commits or aborts, the node recomputes edge weights if the transaction deviated from the original predictions.