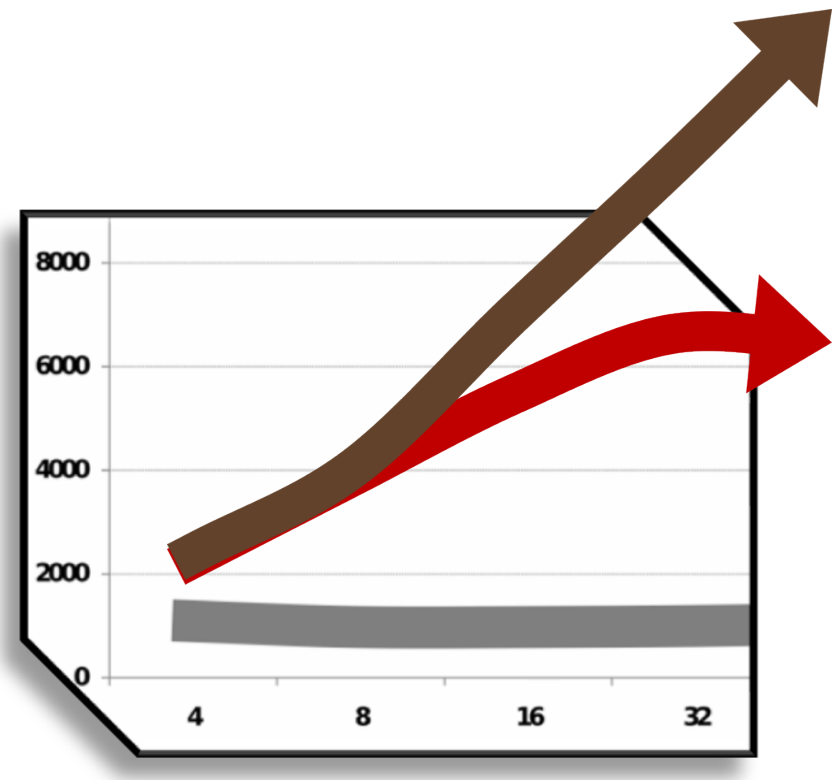
Magical Parallel OLTP Databases

Andy Pavlo

BROWN

November 10th, 2011 – MIT CSAIL



Databases?

Evan Jones?


Lebron is going to Miami!
The McRib will be back!
Michael Jackson is in trouble!

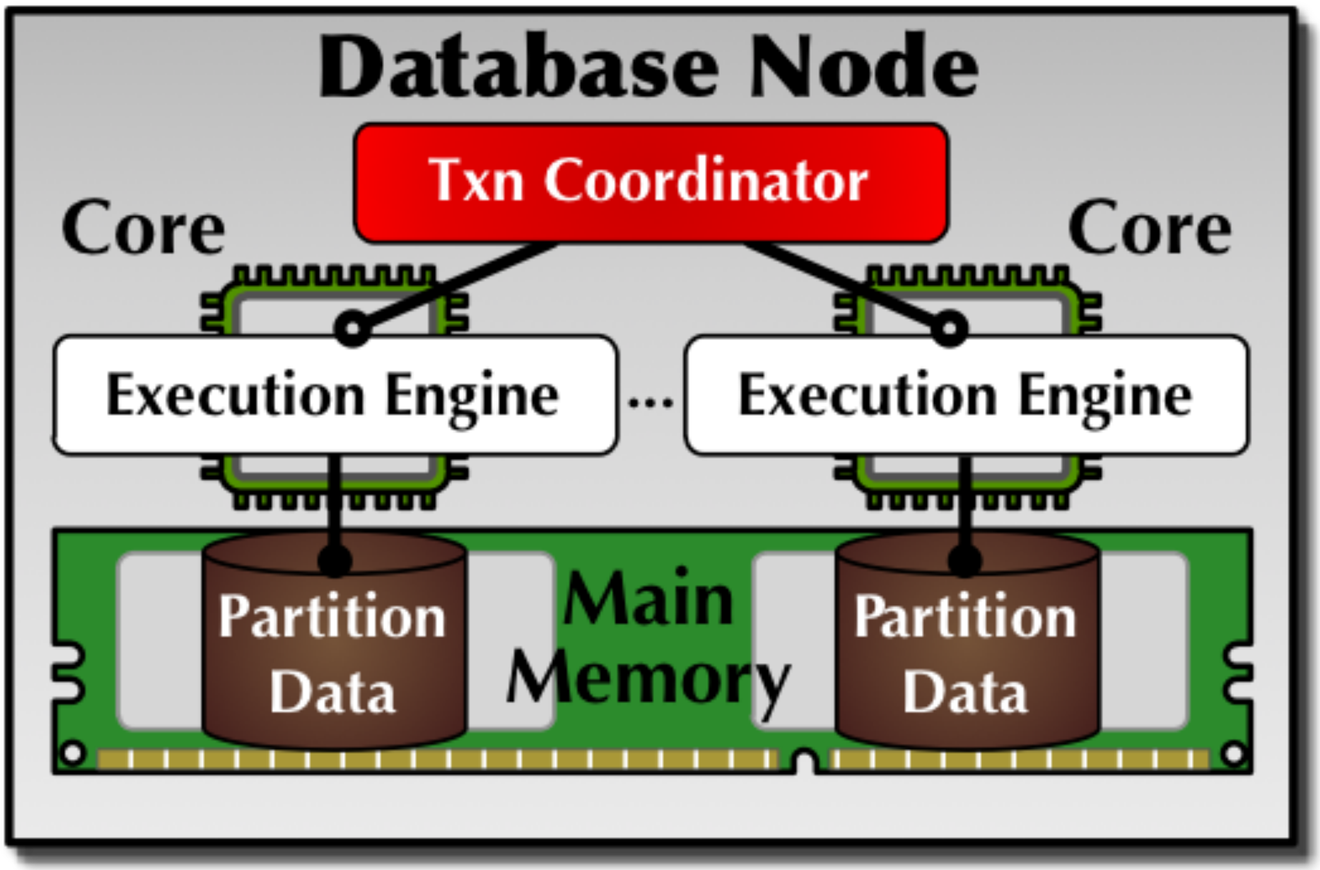
On-Line Transaction Processing

Fast

Repetitive

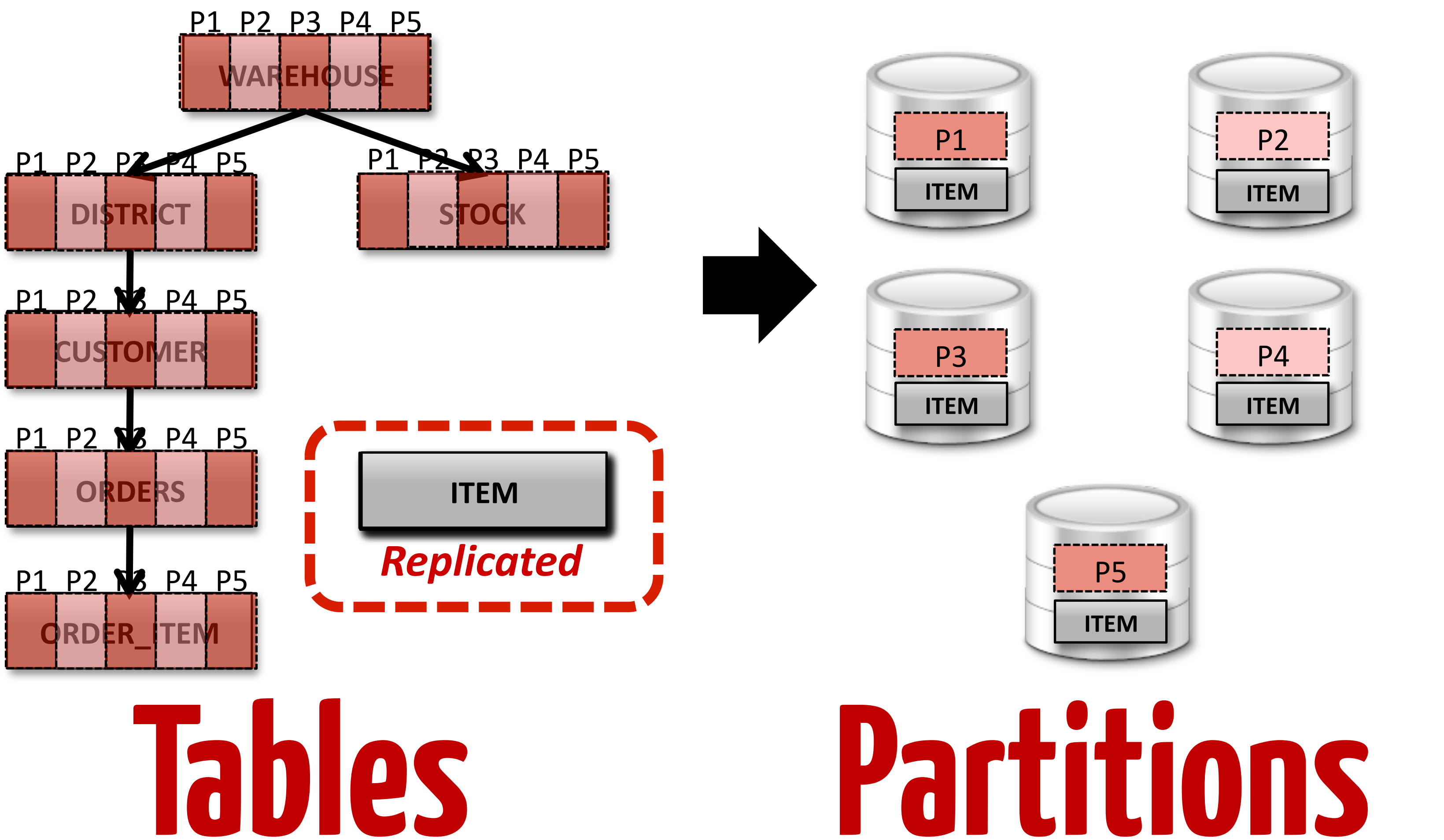
Small

H-Store



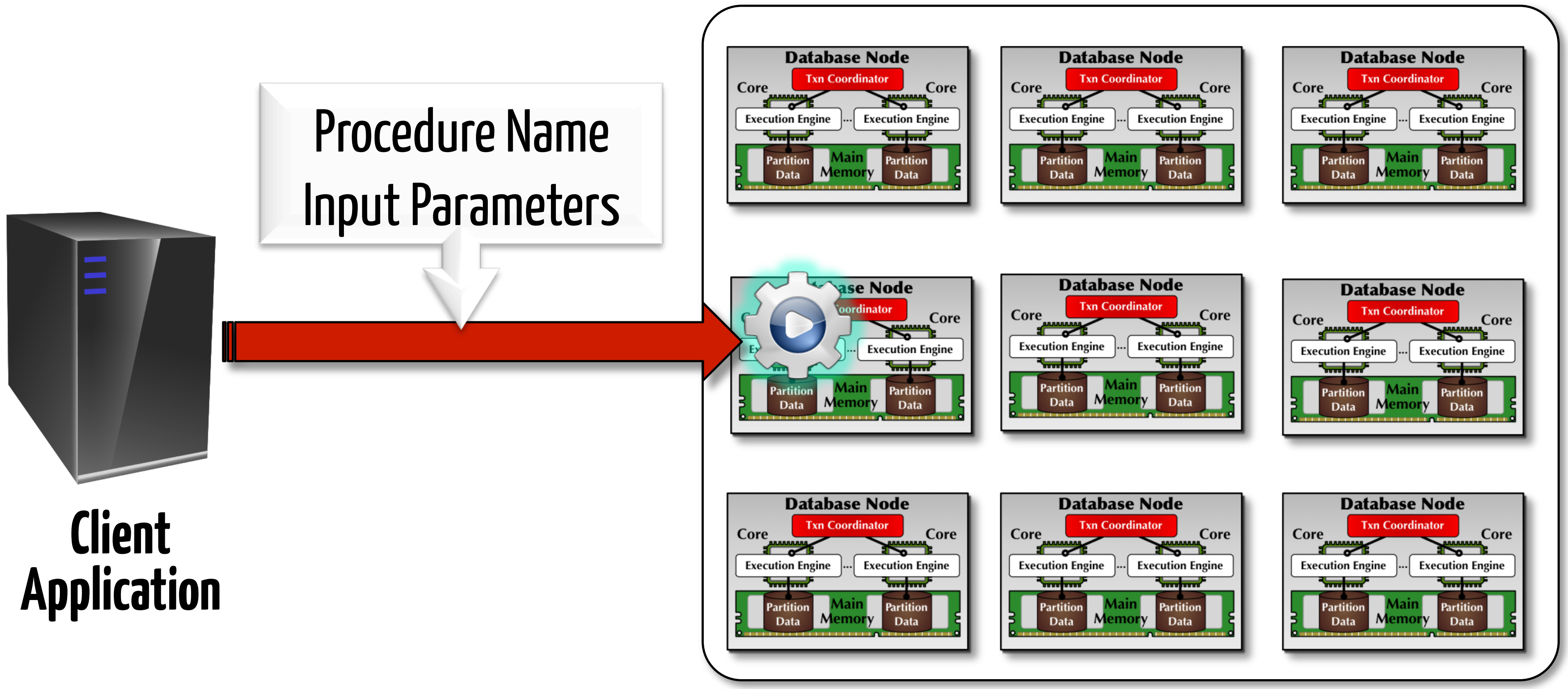
Database Node
Txn Coordinator
Core
Core
Execution Engine
...
Execution Engine
Partition Data
Main Memory
Partition Data

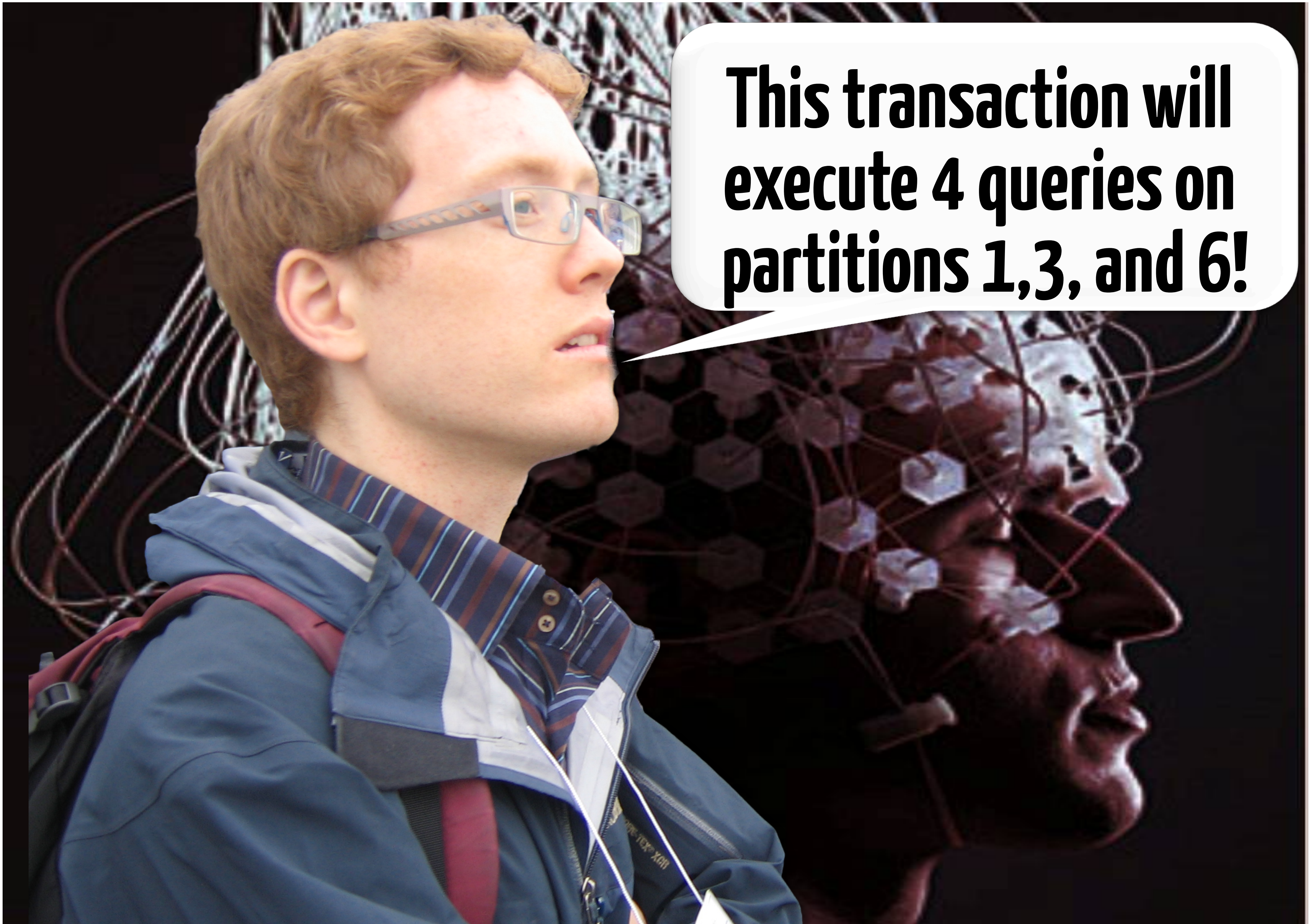
H-Store Partitioning



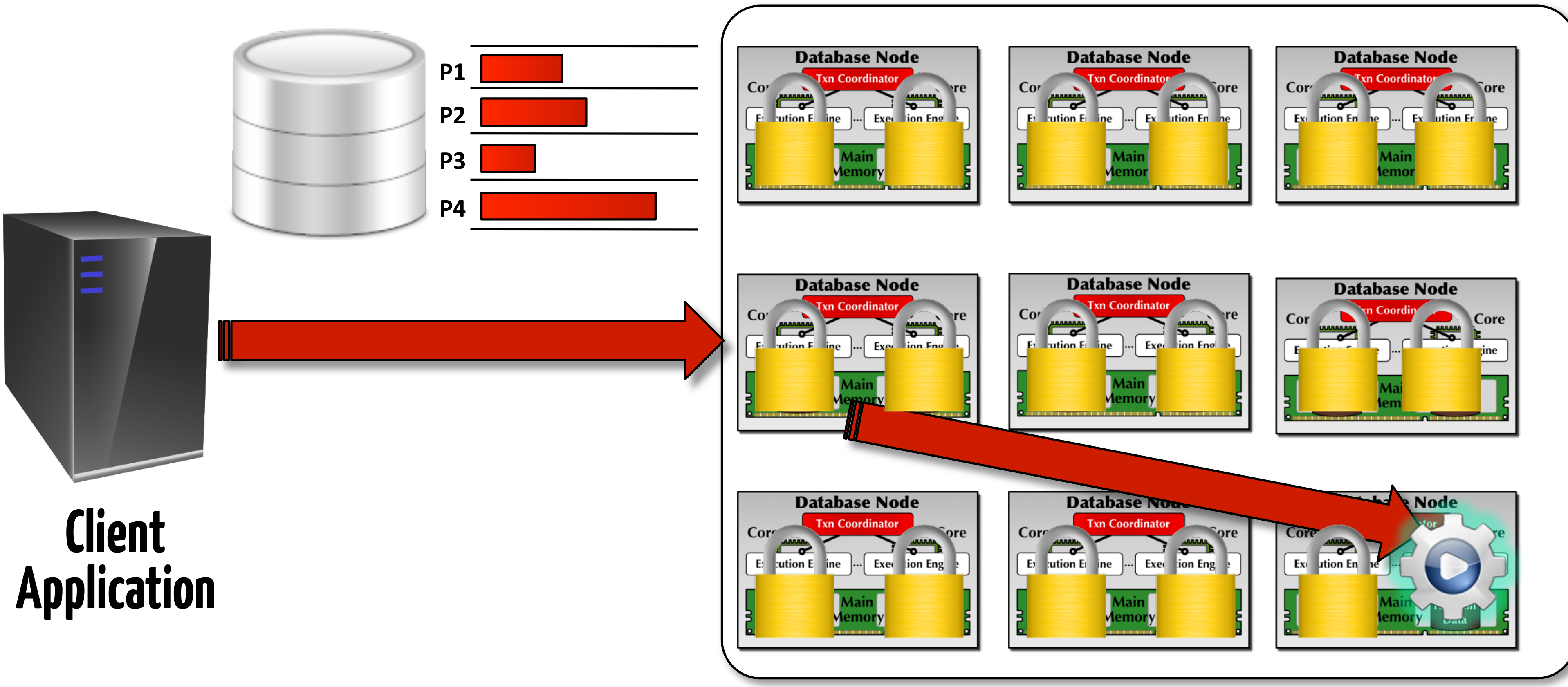
```
class NewOrder extends StoredProcedure {
  Query GetWarehouse   = "SELECT * FROM WAREHOUSE WHERE W_ID = ?";
  Query CheckStock     = "SELECT S_QTY FROM STOCK
                          WHERE S_W_ID = ? AND S_I_ID = ?";
  Query InsertOrder    = "INSERT INTO ORDERS VALUES (?,?)";
  Query InsertOrdLine  = "INSERT INTO ORDER_LINE VALUES (?,?,?,?)";
  Query UpdateStock    = "UPDATE STOCK SET S_QTY = S_QTY - ?
                          WHERE S_W_ID = ? AND S_I_ID = ?";

  int run(int w_id, int i_ids[], int i_w_ids[], int i_qtys[]) {
    queueSQL(GetWarehouse, w_id);
    for (int i = 0; i < i_ids.length; i++)
      queueSQL(CheckStock, i_w_ids[i], i_ids[i]);
    Result r[] = executeBatch();

    int o_id = r[0].get("W_NEXT_O_ID") + 1;
    queueSQL(InsertOrder, w_id, o_id);
    for (int i = 0; i < r.length; i++) {
      if (r[i+1].get("S_QTY") < i_qtys[i]) abort();
      queueSQL(InsertOrderLine, w_id, o_id, i_ids[i], i_qtys[i]);
      queueSQL(UpdateStock, i_qtys[i], i_w_ids[i], i_ids[i]);
    }
    return (executeBatch() != null);
}}
```

H-Store Cluster

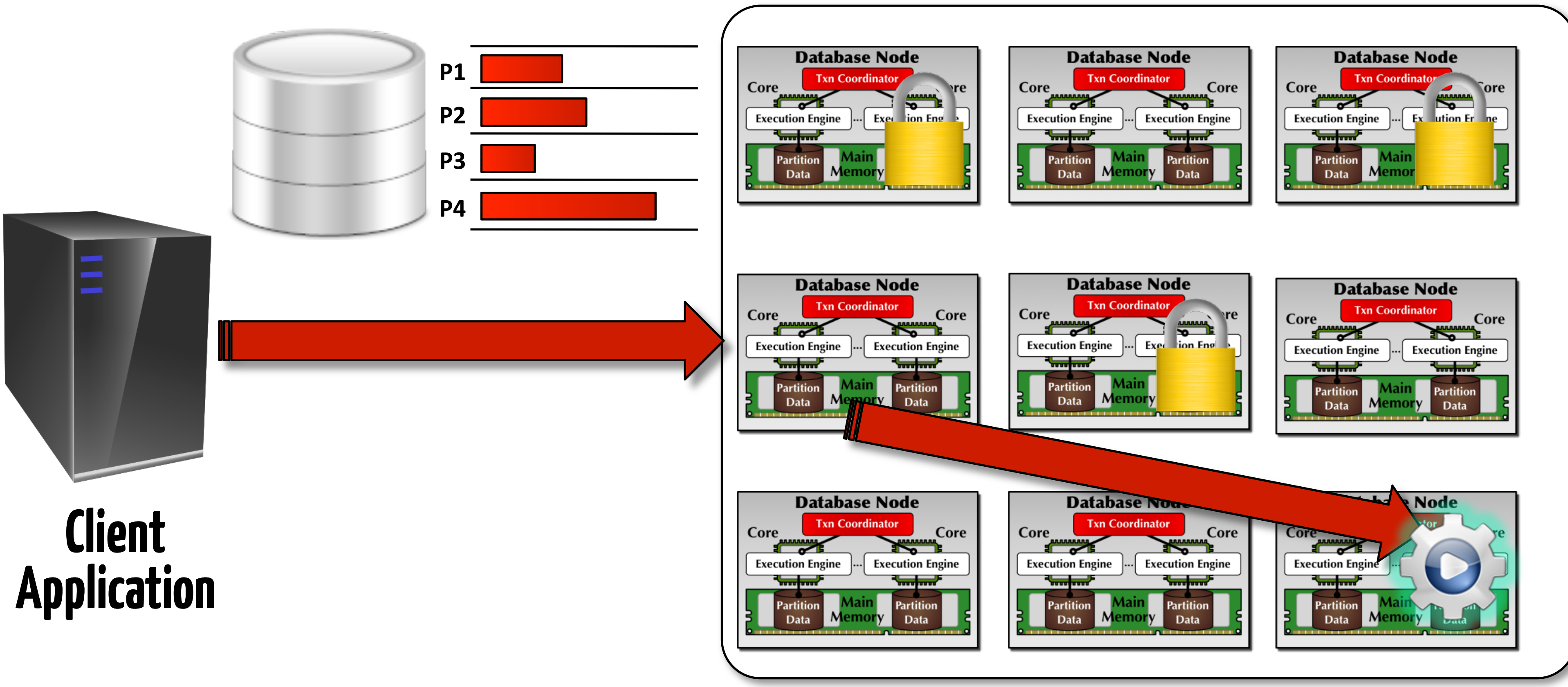



This transaction will execute 4 queries on partitions 1,3, and 6!

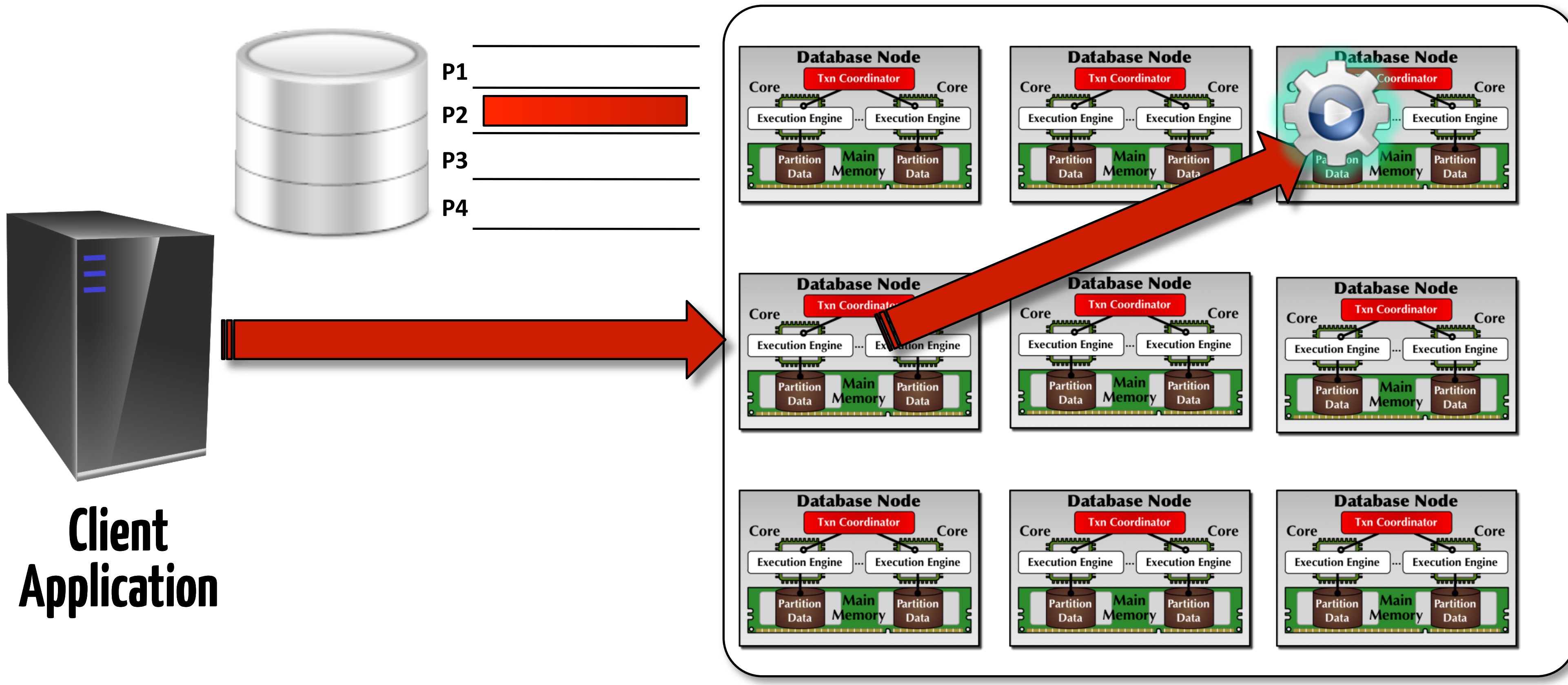
Optimization #1



Optimization #2

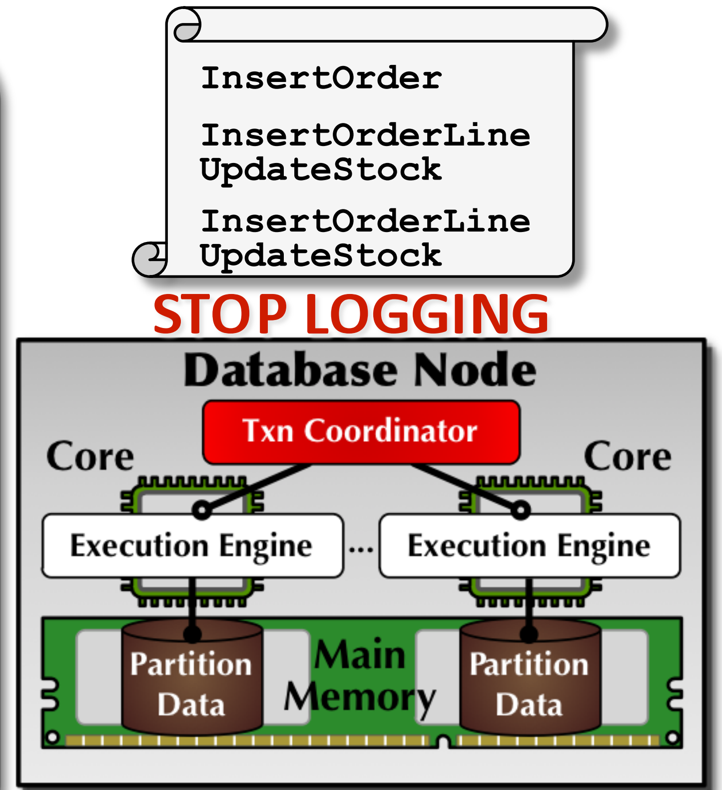


Optimization #2



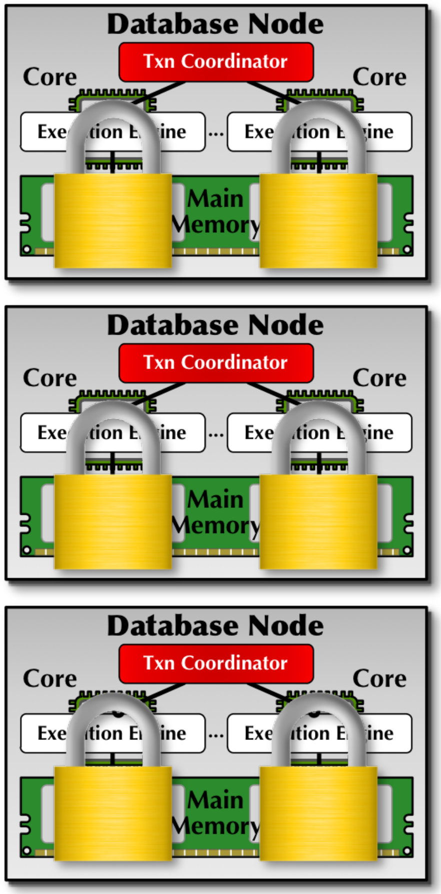
Optimization #3

```
class NewOrder extends StoredProcedure {
  Query GetWarehouse   = "SELECT * FROM WAREHOUSE WHERE W_ID = ?";
  Query CheckStock     = "SELECT S_QTY FROM STOCK
                          WHERE S_W_ID = ? AND S_I_ID = ?";
  Query InsertOrder    = "INSERT INTO ORDERS VALUES (?,?)";
  Query InsertOrdLine  = "INSERT INTO ORDER_LINE VALUES (?,?,?,?)";
  Query UpdateStock    = "UPDATE STOCK SET S_QTY = S_QTY - ?
                          WHERE S_W_ID = ? AND S_I_ID = ?";

  int run(int w_id, int i_ids[], int i_w_ids[], int i_qtys[]) {
    queueSQL(GetWarehouse, w_id);
    for (int i = 0; i < i_ids.length; i++)
      queueSQL(CheckStock, i_w_ids[i], i_ids[i]);
    Result r[] = executeBatch();

    int o_id = r[0].get("W_NEXT_O_ID") + 1;
    queueSQL(InsertOrder, w_id, o_id);
    for (int i = 0; i < r.length; i++) {
      if (r[i+1].get("S_QTY") < i_qtys[i]) abort();
      queueSQL(InsertOrderLine, w_id, o_id, i_ids[i], i_qtys[i]);
      queueSQL(UpdateStock, i_qtys[i], i_w_ids[i], i_ids[i]);
    }
    return (executeBatch() != null);
}}
```

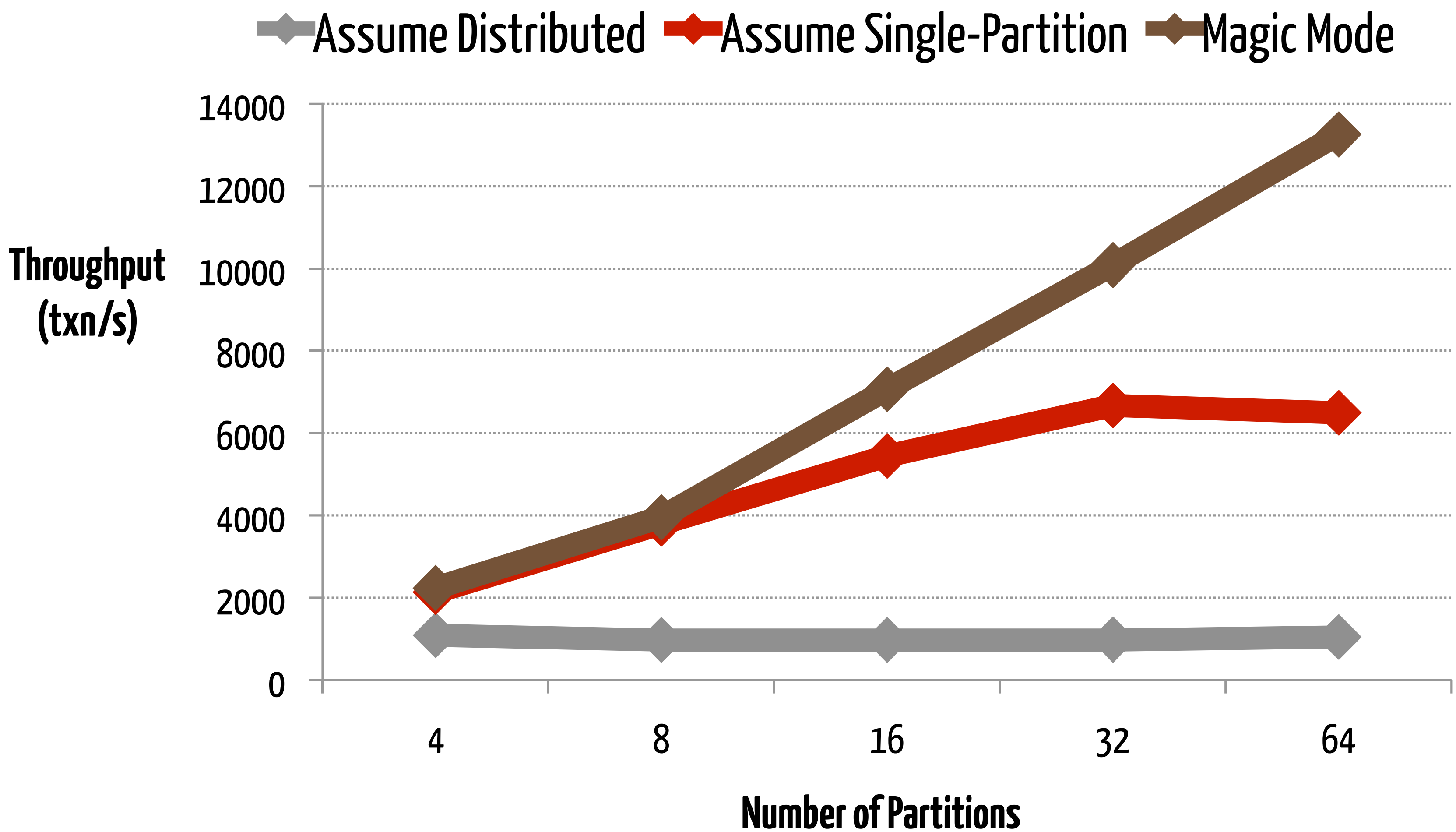


Optimization #4

```
class NewOrder extends StoredProcedure {
  Query GetWarehouse   = "SELECT * FROM WAREHOUSE WHERE W_ID = ?";
  Query CheckStock     = "SELECT S_QTY FROM STOCK
                          WHERE S_W_ID = ? AND S_I_ID = ?";
  Query InsertOrder    = "INSERT INTO ORDERS VALUES (?,?)";
  Query InsertOrdLine  = "INSERT INTO ORDER_LINE VALUES (?,?,?,?)";
  Query UpdateStock    = "UPDATE STOCK SET S_QTY = S_QTY - ?
                          WHERE S_W_ID = ? AND S_I_ID = ?";

  int run(int w_id, int i_ids[], int i_w_ids[], int i_qtys[]) {
    queueSQL(GetWarehouse, w_id);
    for (int i = 0; i < i_ids.length; i++)
      queueSQL(CheckStock, i_w_ids[i], i_ids[i]);
    Result r[] = executeBatch();

    int o_id = r[0].get("W_NEXT_O_ID") + 1;
    queueSQL(InsertOrder, w_id, o_id);
    for (int i = 0; i < r.length; i++) {
      if (r[i+1].get("S_QTY") < i_qtys[i]) abort();
      queueSQL(InsertOrderLine, w_id, o_id, i_ids[i], i_qtys[i]);
      queueSQL(UpdateStock, i_qtys[i], i_w_ids[i], i_ids[i]);
    }
    return (executeBatch() != null);
}}
```



Why this Matters

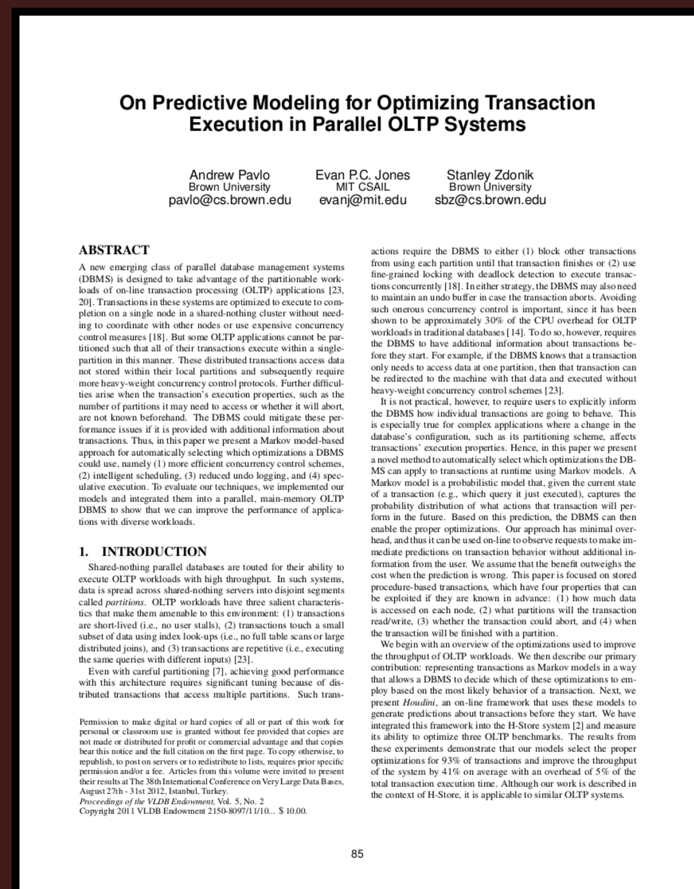


Pro Tip:

Canadians do not like unnecessary surgeries.

On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems

in PVLDB, vol 5. issue 2, October 2011



On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Evan P.C. Jones
MIT CSAIL
evanj@mit.edu

Stanley Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

A new emerging class of parallel database management systems (DBMS) is designed to take advantage of the partitionable workloads of on-line transaction processing (OLTP) applications [23, 20]. Transactions in these systems are optimized to execute to completion on a single node in a shared-nothing cluster without needing to coordinate with other nodes or use expensive concurrency control measures [18]. But some OLTP applications cannot be partitioned such that all of their transactions execute within a single-partition in this manner. These distributed transactions access data not stored within their local partitions and subsequently require more heavy-weight concurrency control protocols. Further difficulties arise when the transaction's execution properties, such as the number of partitions it may need to access or whether it will abort, are not known beforehand. The DBMS could mitigate these performance issues if it is provided with additional information about transactions. Thus, in this paper we present a Markov model-based approach for automatically selecting which optimizations a DBMS could use, namely (1) more efficient concurrency control schemes, (2) intelligent scheduling, (3) reduced undo logging, and (4) speculative execution. To evaluate our techniques, we implemented our models and integrated them into a parallel, main-memory OLTP DBMS to show that we can improve the performance of applications with diverse workloads.

1. INTRODUCTION

Shared-nothing parallel databases are touted for their ability to execute OLTP workloads with high throughput. In such systems, data is spread across shared-nothing servers into disjoint segments called *partitions*. OLTP workloads have three salient characteristics that make them amenable to this environment: (1) transactions are short-lived (i.e., no user stalls), (2) transactions touch a small subset of data using index look-ups (i.e., no full table scans or large distributed joins), and (3) transactions are repetitive (i.e., executing the same queries with different inputs) [23].

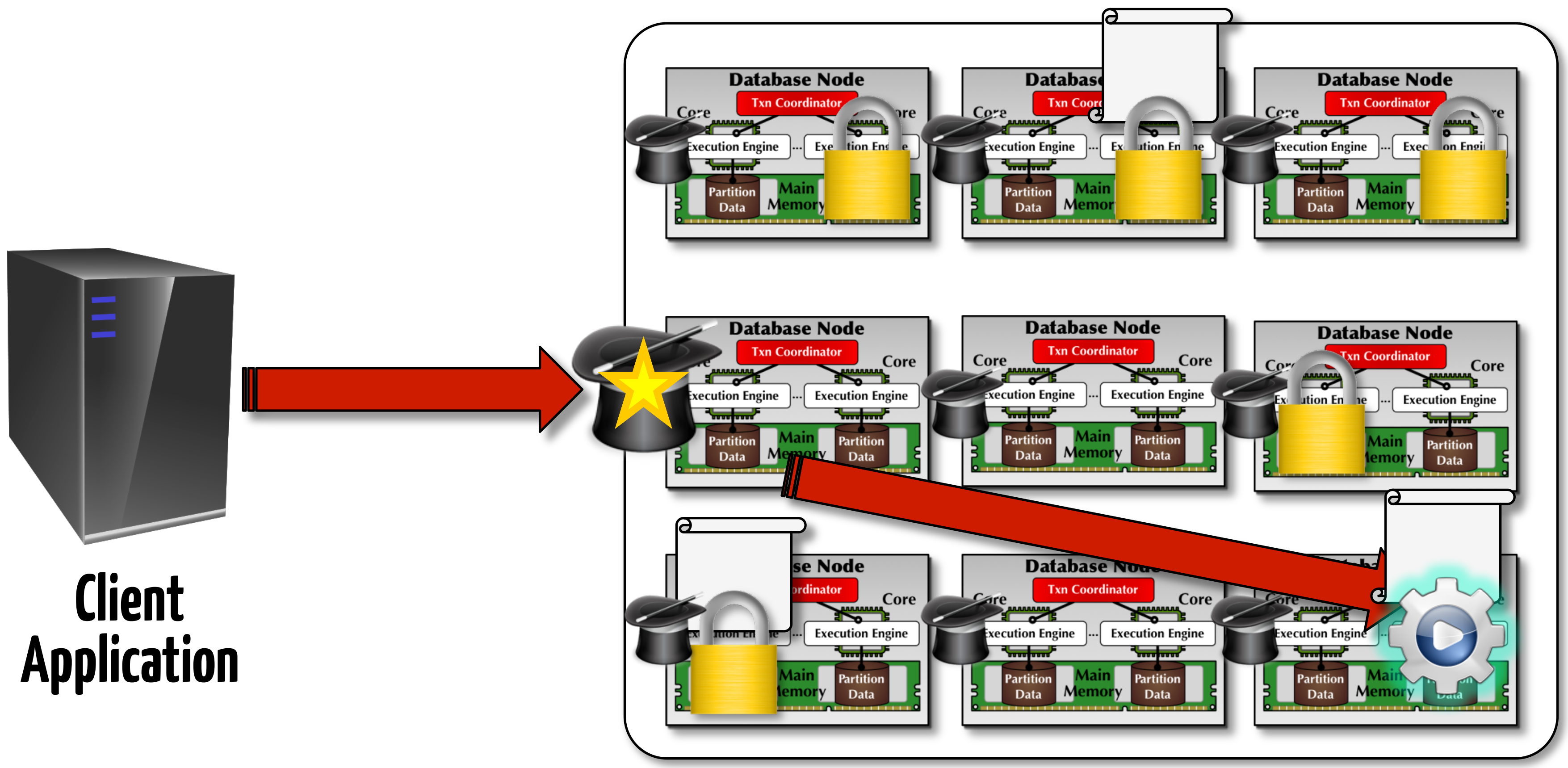
Even with careful partitioning [7], achieving good performance with this architecture requires significant tuning because of distributed transactions that access multiple partitions. Such transactions require the DBMS to either (1) block other transactions from using each partition until that transaction finishes or (2) use fine-grained locking with deadlock detection to execute transactions concurrently [18]. In either strategy, the DBMS may also need to maintain an undo buffer in case the transaction aborts. Avoiding such onerous concurrency control is important, since it has been shown to be approximately 30% of the CPU overhead for OLTP workloads in traditional databases [14]. To do so, however, requires the DBMS to have additional information about transactions before they start. For example, if the DBMS knows that a transaction only needs to access data at one partition, then that transaction can be redirected to the machine with that data and executed without heavy-weight concurrency control schemes [23].

It is not practical, however, to require users to explicitly inform the DBMS how individual transactions are going to behave. This is especially true for complex applications where a change in the database's configuration, such as its partitioning scheme, affects transactions' execution properties. Hence, in this paper we present a novel method to automatically select which optimizations the DBMS can apply to transactions at runtime using Markov models. A Markov model is a probabilistic model that, given the current state of a transaction (e.g., which query it just executed), captures the probability distribution of what actions that transaction will perform in the future. Based on this prediction, the DBMS can then enable the proper optimizations. Our approach has minimal overhead, and thus it can be used on-line to observe requests to make immediate predictions on transaction behavior without additional information from the user. We assume that the benefit outweighs the cost when the prediction is wrong. This paper is focused on stored procedure-based transactions, which have four properties that can be exploited if they are known in advance: (1) how much data is accessed on each node, (2) what partitions will the transaction read/write, (3) whether the transaction could abort, and (4) when the transaction will be finished with a partition.

We begin with an overview of the optimizations used to improve the throughput of OLTP workloads. We then describe our primary contribution: representing transactions as Markov models in a way that allows a DBMS to decide which of these optimizations to employ based on the most likely behavior of a transaction. Next, we present *Houdini*, an on-line framework that uses these models to generate predictions about transactions before they start. We have integrated this framework into the H-Store system [2] and measure its ability to optimize three OLTP benchmarks. The results from these experiments demonstrate that our models select the proper optimizations for 93% of transactions and improve the throughput of the system by 41% on average with an overhead of 5% of the total transaction execution time. Although our work is described in the context of H-Store, it is applicable to similar OLTP systems.

Proceedings of the VLDB Endowment, Vol. 5, No. 2

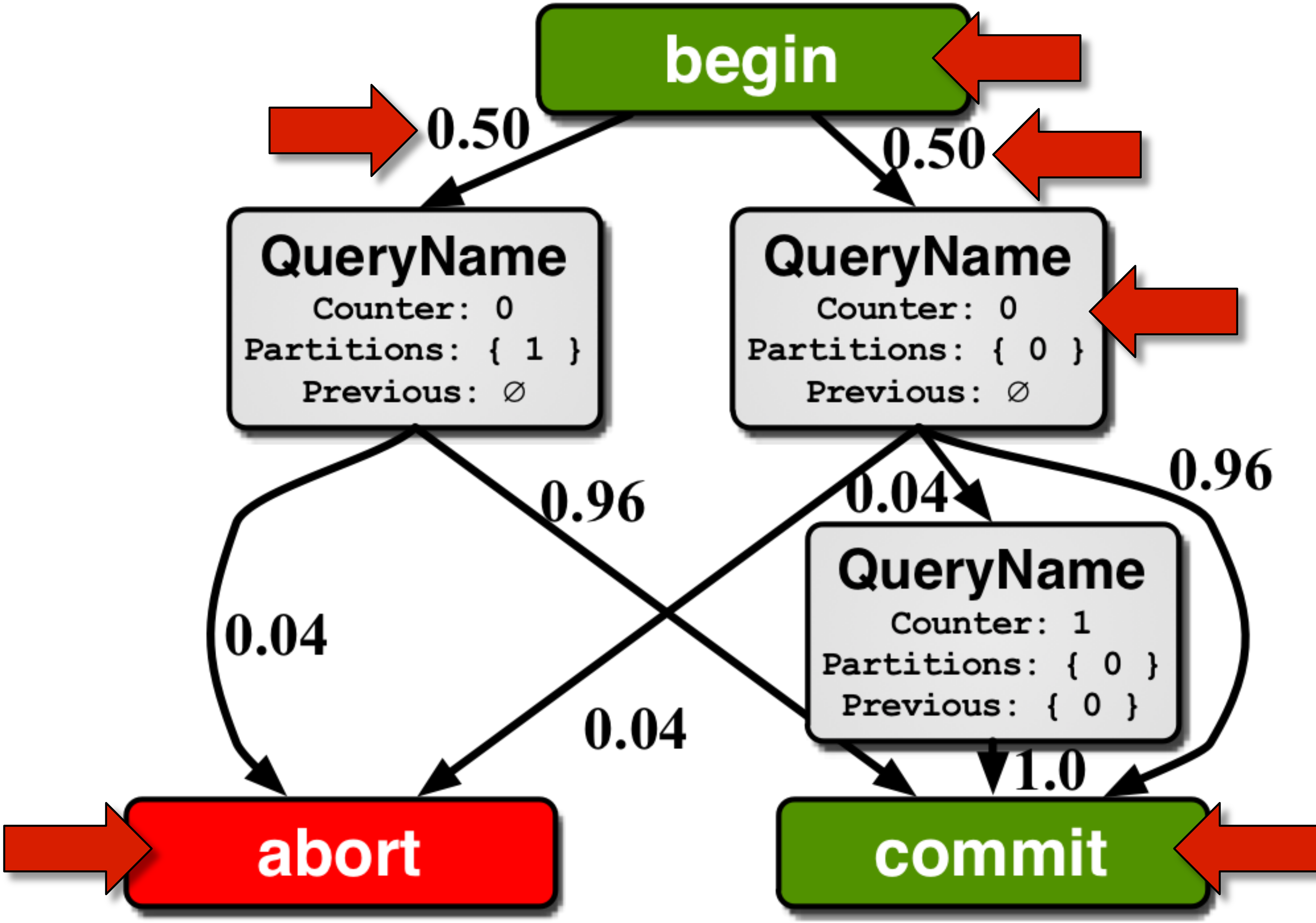
Houdini

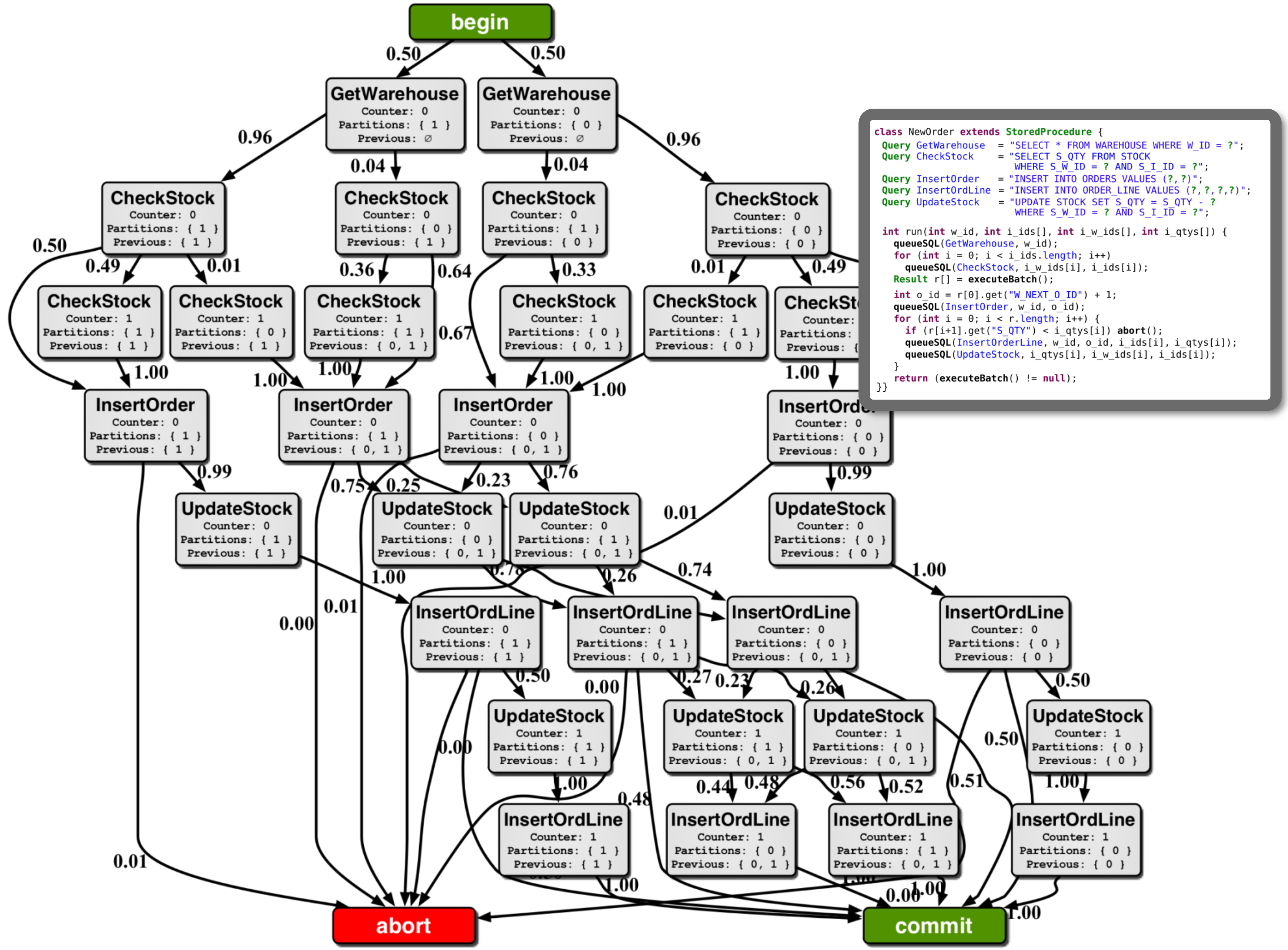


Client Application
Database Node
Txn Coordinator
Core
Execution Engine
Partition Data
Main Memory

Main Idea:

Use models to predict <u>before</u> execution.


begin
0.50
0.50
QueryName
Counter: 0
Partitions: { 1 }
Previous: ∅
QueryName
Counter: 0
Partitions: { 0 }
Previous: ∅
0.96
0.04
0.96
QueryName
Counter: 1
Partitions: { 0 }
Previous: { 0 }
0.04
0.04
1.0
abort
commit

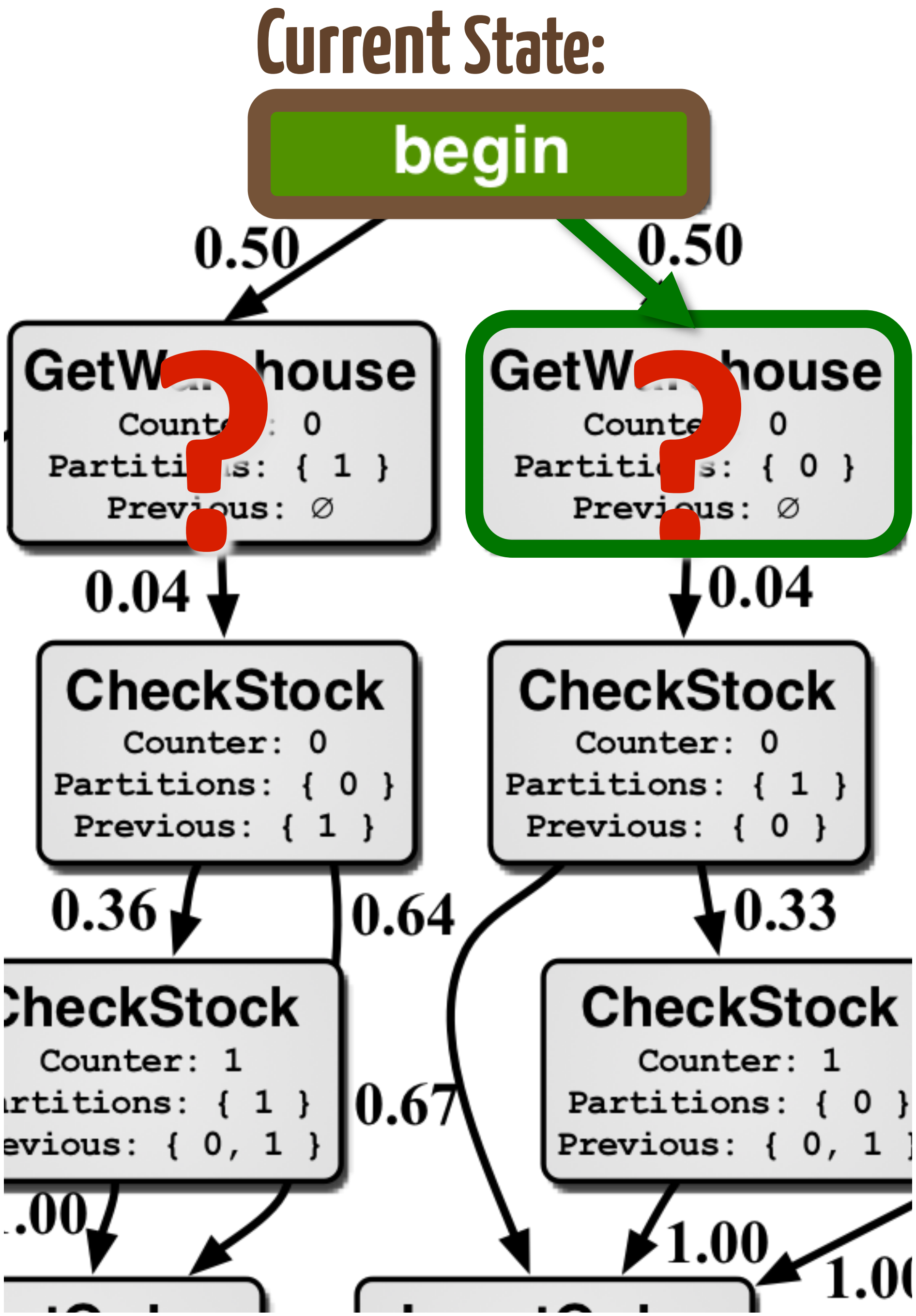

begin
GetWarehouse
CheckStock
InsertOrder
UpdateStock
InsertOrdLine
abort
commit
class NewOrder extends StoredProcedure {
 Query GetWarehouse   = "SELECT * FROM WAREHOUSE WHERE W_ID = ?";
 Query CheckStock   = "SELECT S_QTY FROM STOCK
                         WHERE S_W_ID = ? AND S_I_ID = ?";
 Query InsertOrder   = "INSERT INTO ORDERS VALUES (?,?)";
 Query InsertOrdLine = "INSERT INTO ORDER_LINE VALUES (?,?,?,?)";
 Query UpdateStock   = "UPDATE STOCK SET S_QTY = S_QTY - ?
                         WHERE S_W_ID = ? AND S_I_ID = ?";
 int run(int w_id, int i_ids[], int i_w_ids[], int i_qtys[]) {
   queueSQL(GetWarehouse, w_id);
   for (int i = 0; i < i_ids.length; i++)
     queueSQL(CheckStock, i_w_ids[i], i_ids[i]);
   Result r[] = executeBatch();
   int o_id = r[0].get("W_NEXT_O_ID") + 1;
   queueSQL(InsertOrder, w_id, o_id);
   for (int i = 0; i < r.length; i++) {
     if (r[i+1].get("S_QTY") < i_qtys[i]) abort();
     queueSQL(InsertOrderLine, w_id, o_id, i_ids[i], i_qtys[i]);
     queueSQL(UpdateStock, i_qtys[i], i_w_ids[i], i_ids[i]);
   }
   return (executeBatch() != null);
}}

Step #1:

Estimate the path that a transaction will take

Step #2:

Determine which optimizations to enable.

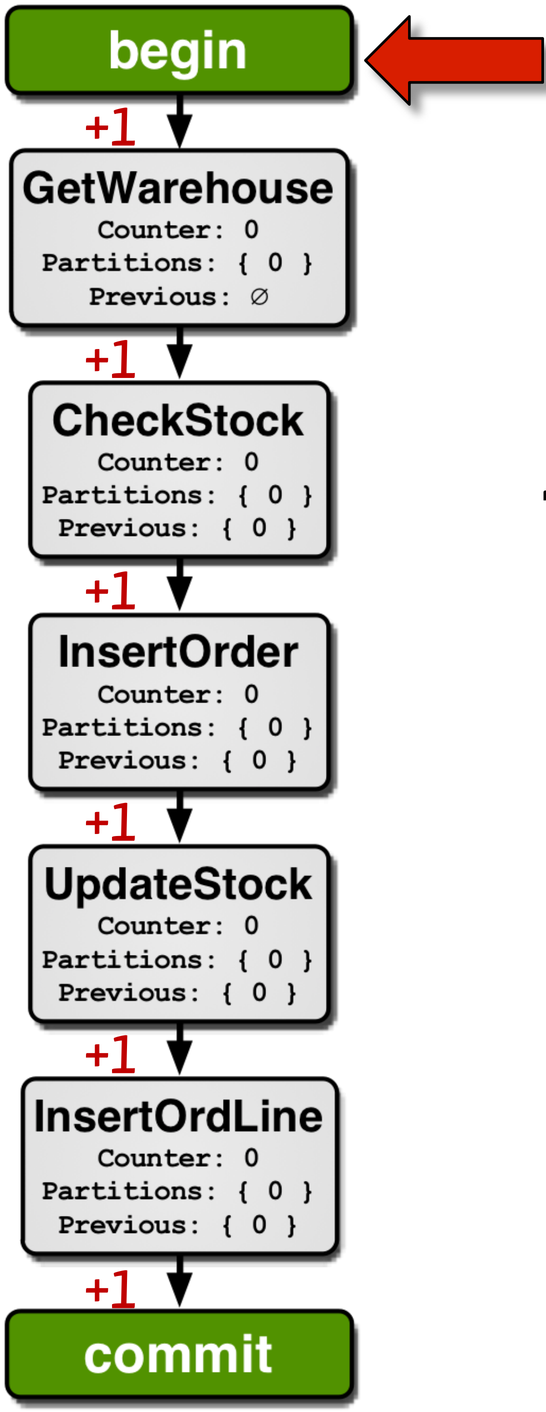


Input Parameters:

```
w_id=0
i_w_id=[0,1] i_ids=
[1001,1002]
```

GetWarehouse:

```
SELECT * FROM WAREHOUSE
 WHERE W_ID = ?
```



Input Parameters:

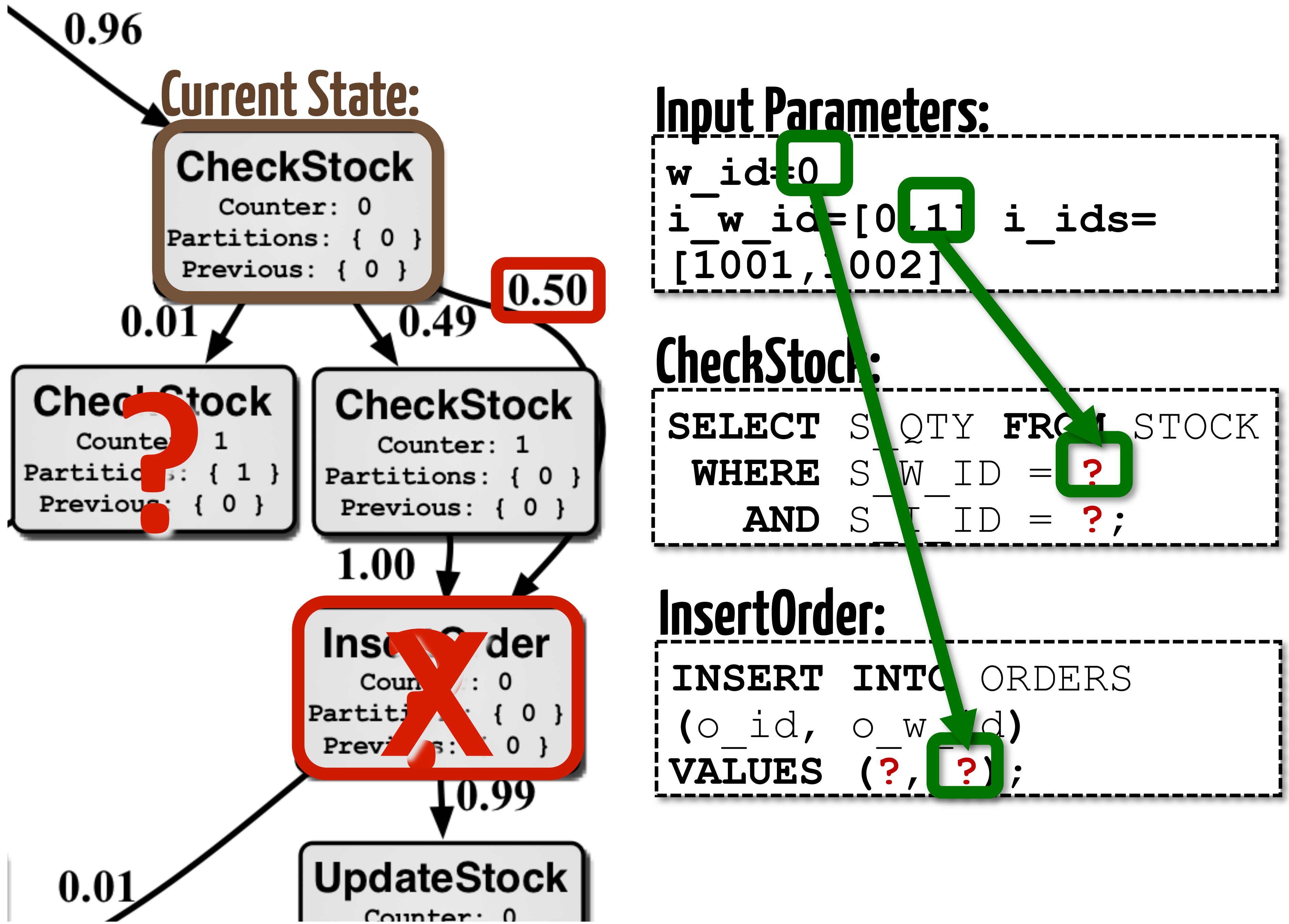
```
w_id=0
i_w_id=[0,1] i_ids=
[1001,1002]
```

Transaction Estimate:

Confidence Coefficient:	0.56
Best Partition:	0
Use Undo Logging:	Yes
Partitions Read:	{ 0 }
Partitions Written:	{ 0 }
Partitions Done:	{1, 2, 3}

Limitations:

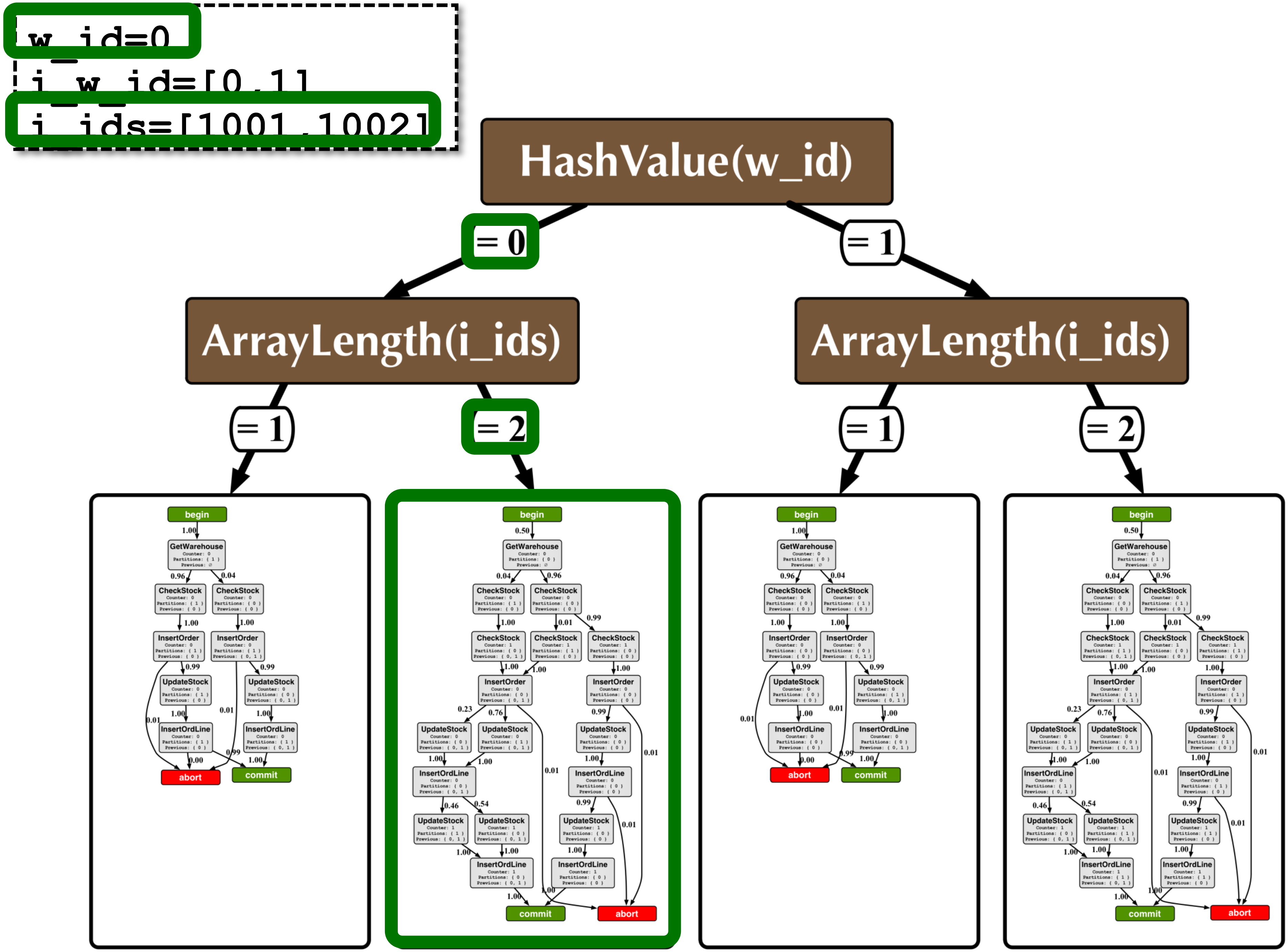
1) Long/wide models.
2) Keeping models in synch.
3) Incorrect predictions.



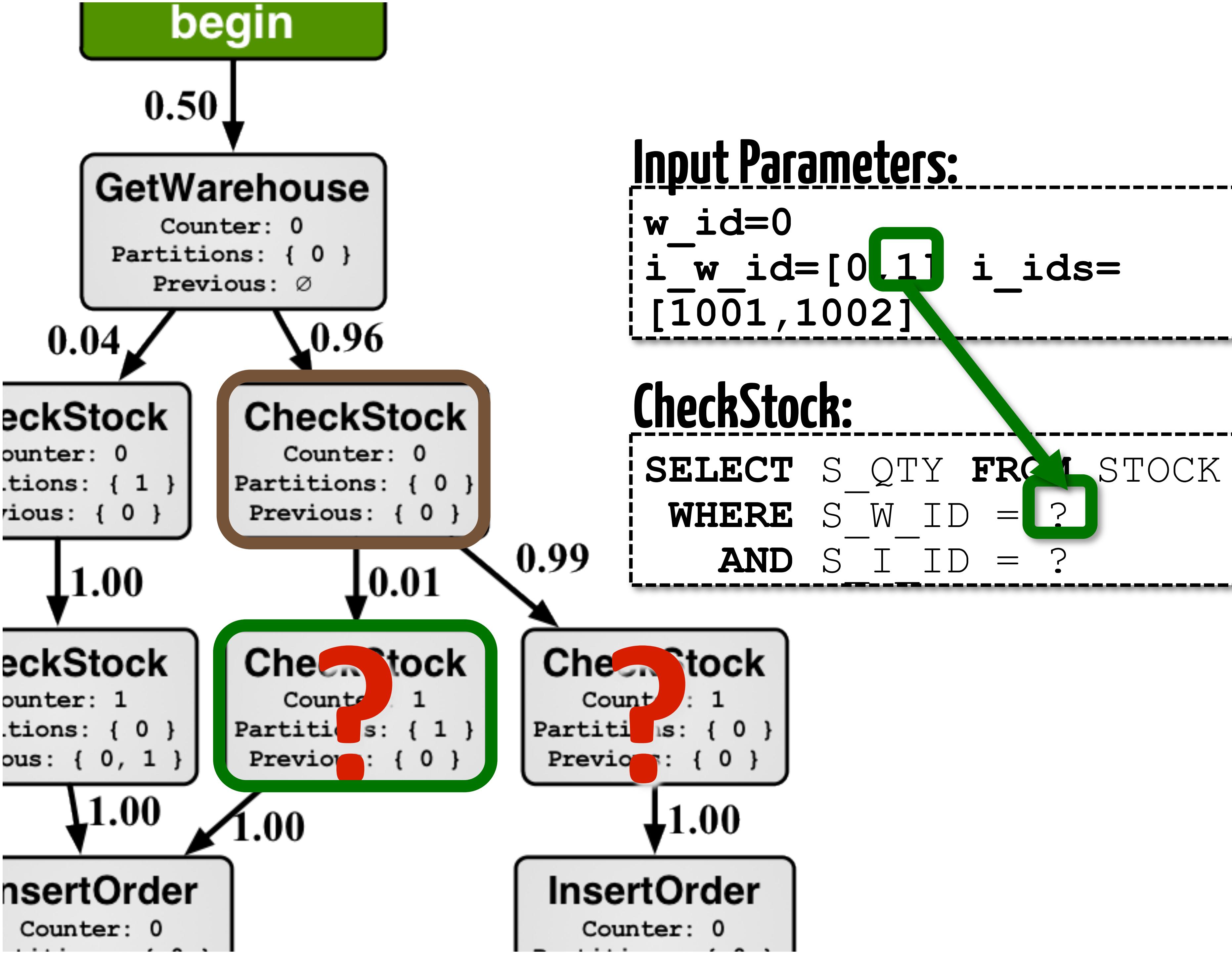
0.96
Current State:
CheckStock
Counter: 0
Partitions: { 0 }
Previous: { 0 }
0.01
0.49
0.50
Previous: { 0 }
CheckStock
Counter: 1
Partitions: { 0 }
Previous: { 0 }
1.00
0.99
0.01
UpdateStock
Input Parameters:
w_id=0
i_ids=
CheckStock:
SELECT
WHERE
AND
STOCK
InsertOrder:
INSERT
ORDERS
VALUES


THE GIFTS OF MISS CLEO
SUNDAY,
SEPTEMBER 16
9PM ET
ON iN DEMAND
PAY-PER-VIEW
Special Guest: Evan
GET A FREE 7 MINUTE TAROT READING

Refinement:
Partition models based on input properties.



w_id=0
i_w_id=[0,1]
i_ids=[1001,1002]
HashValue(w_id)
= 0
= 1
ArrayLength(i_ids)
ArrayLength(i_ids)
= 1
= 2
= 1
= 2



begin
0.50
GetWarehouse
Counter: 0
Partitions: { 0 }
Previous: ∅
0.04
0.96
CheckStock
Counter: 0
Partitions: { 0 }
Previous: { 0 }
1.00
0.01
0.99
1.00
1.00
1.00
InsertOrder
Counter: 0
Input Parameters:
w_id=0
i_w_id=[0,1] i_ids=
[1001,1002]
CheckStock:
SELECT S_QTY FROM STOCK
WHERE S_W_ID = ?
AND S_I_ID = ?

Houdini:

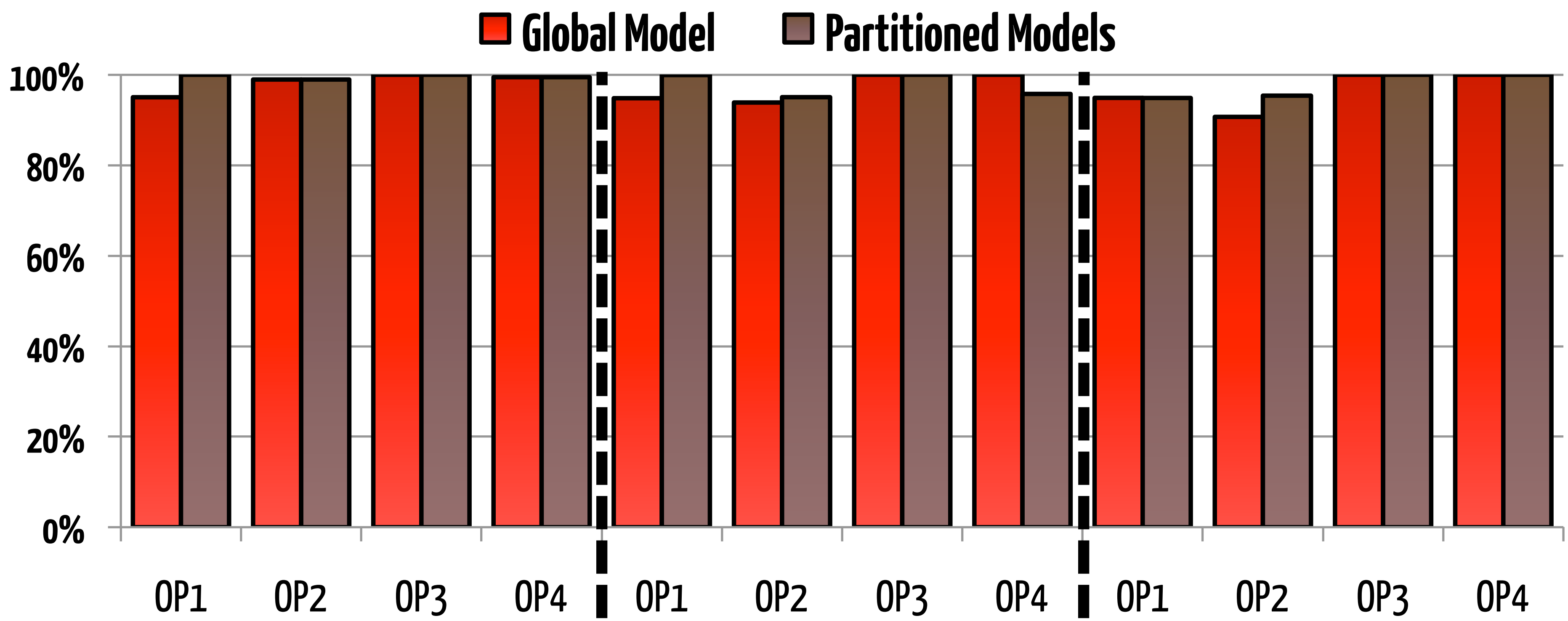
1) Execute txn at the best partition.
2) Only lock the partitions needed.
3) Disable undo logging if not needed.
4) Speculatively commit transactions.

Houdini:

5) Estimate initial path.
6) Update as transaction executes.
7) Recompute if workload changes.
8) Partition for better accuracy.

Experimental Evaluation

Model Accuracy

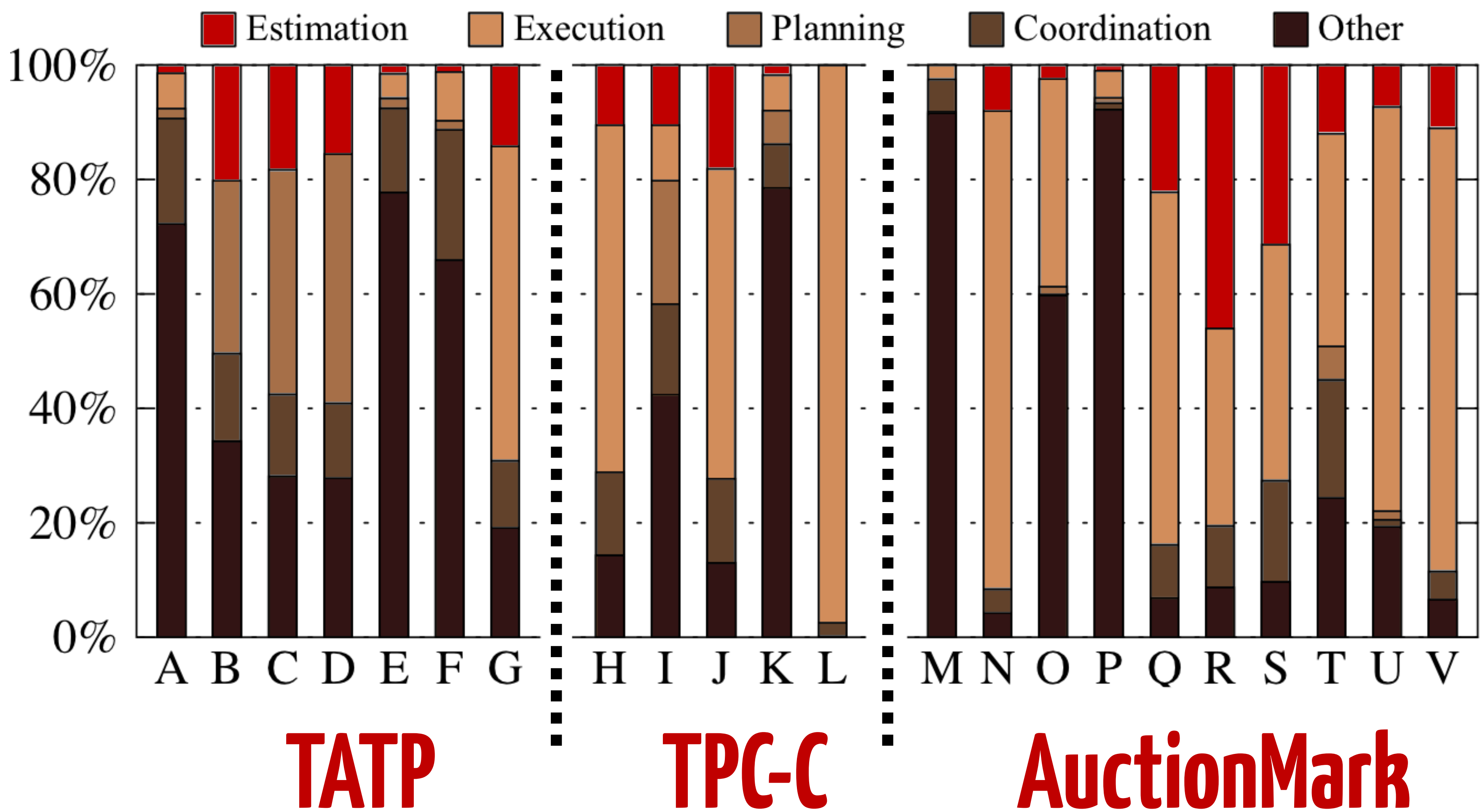


TATP 94.9%

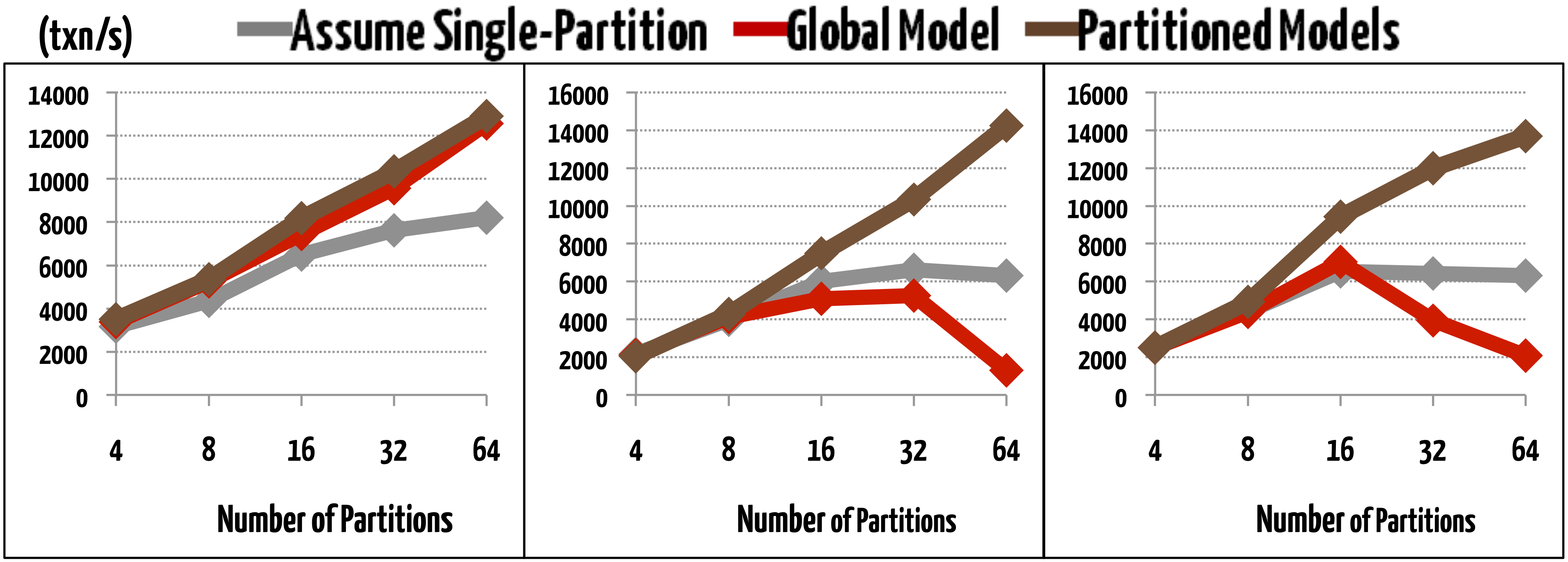
TPC-C 95.0%

AuctionMark 90.2%

Estimation Overhead



Throughput



TATP	TPC-C	AuctionMark
+57%	+126%	+117%

Conclusion:

Small overhead cost improves throughput.

h-store

hstore.cs.brown.edu

Twitter: @andy_pavlo

Future work

```
class NewOrder extends StoredProcedure {
 Query GetWarehouse  = "SELECT * FROM WAREHOUSE WHERE W_ID = ?";
 Query CheckStock    = "SELECT S_QTY FROM STOCK
                        WHERE S_W_ID = ? AND S_I_ID = ?";
 Query InsertOrder   = "INSERT INTO ORDERS VALUES (?,?)";
 Query InsertOrdLine = "INSERT INTO ORDER_LINE VALUES (?,?,?,?)";
 Query UpdateStock   = "UPDATE STOCK SET S_QTY = S_QTY - ?
                        WHERE S_W_ID = ? AND S_I_ID = ?";

 int run(int w_id, int i_ids[], int i_w_ids[], int i_qtys[]) {
   queueSQL(GetWarehouse, w_id);
   for (int i = 0; i < i_ids.length; i++)
     queueSQL(CheckStock, i_w_ids[i], i_ids[i]);
   Result r[] = executeBatch();

   int o_id = r[0].get("W_NEXT_O_ID") + 1;
   queueSQL(InsertOrder, w_id, o_id);
   for (int i = 0; i < r.length; i++) {
     if (r[i+1].get("S_QTY") < i_qtys[i]) abort();
     queueSQL(InsertOrderLine, w_id, o_id, i_ids[i], i_qtys[i]);
     queueSQL(UpdateStock, i_qtys[i], i_w_ids[i], i_ids[i]);
   }
   return (executeBatch() != null);
}}
```