Automated Data Partitioning for Highly Scalable and Strongly Consistent Transactions

Alexandru Turcu, Roberto Palmieri, Binoy Ravindran

Virginia Tech

SYSTOR 2014

Desirable properties in distribute transactional systems

High availability

Strong consistency

High performance

Programmability

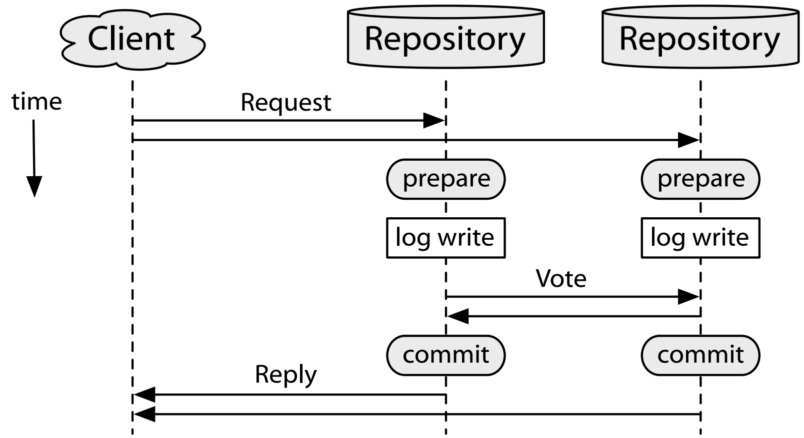
Great scalability

Fault tolerance

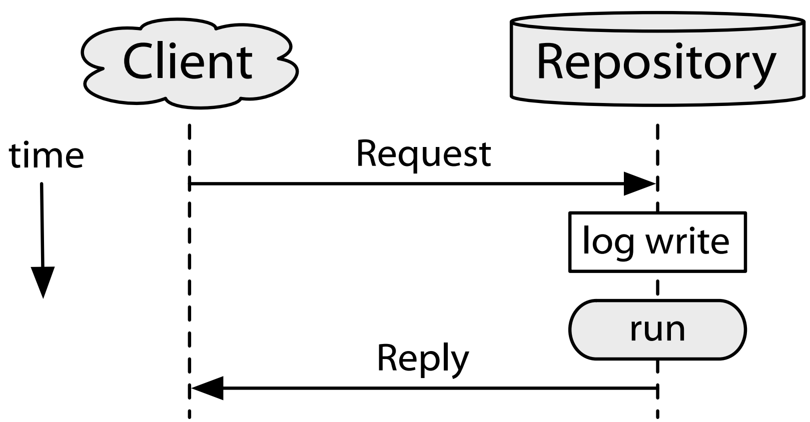
Low latency

GRANOLA: Transaction model [Cowling, Liskov at ATC'12]

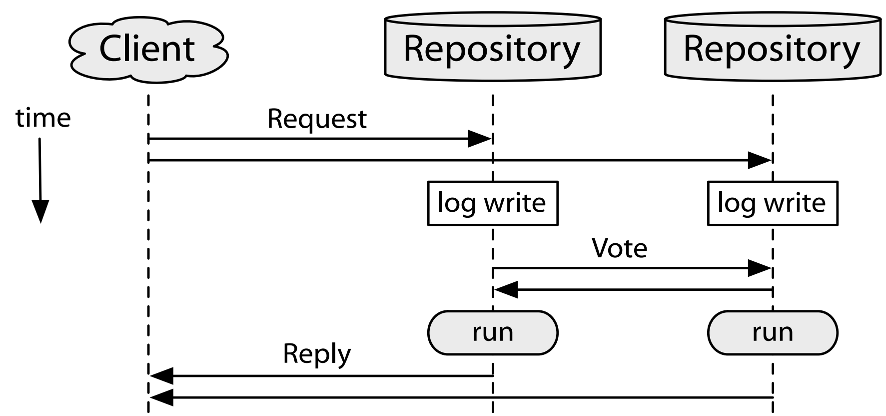
(Classical) Distributed coordinated



Single Repository

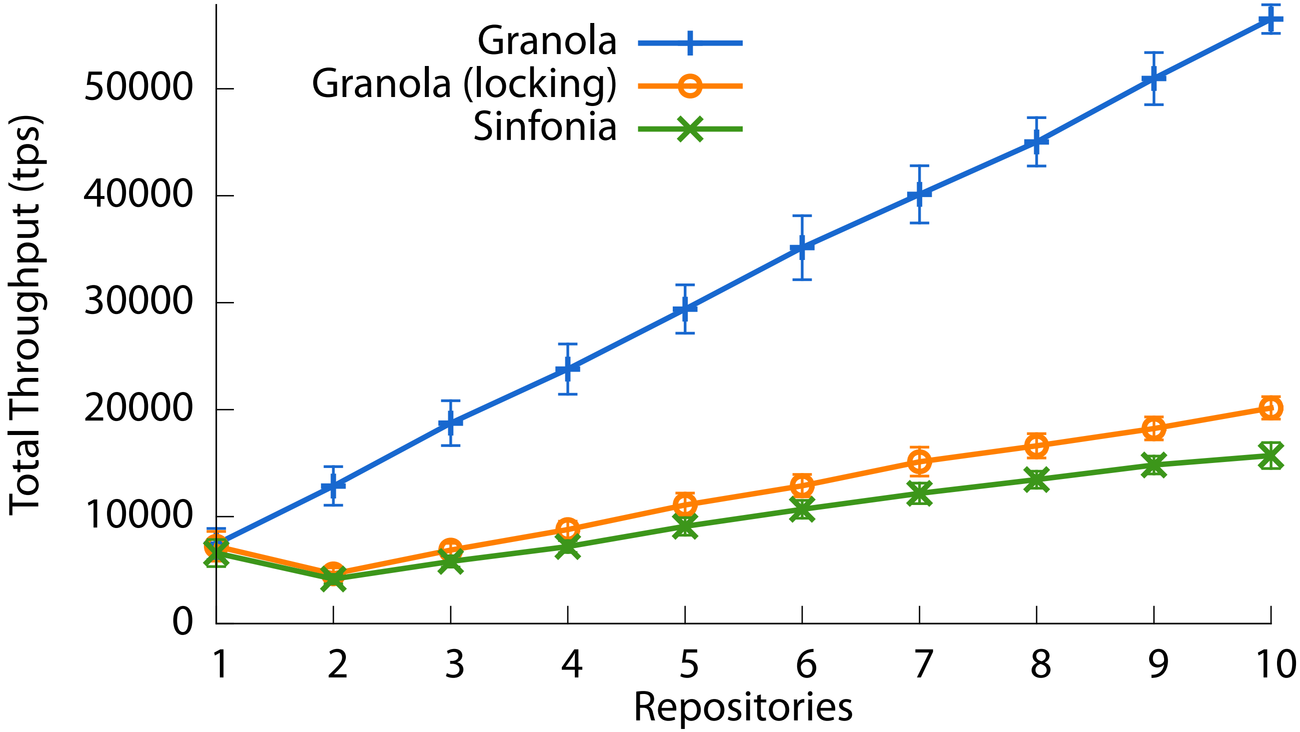


Distributed Independent



Granola Performance…terrific scalability!

[From Granola at ATC '12]



Configuration: TPC-C benchmark; increased number of clients to maximize throughput; No coordinated transactions; ≈10% of transactions are independent; ≈90% of transactions are single repository.

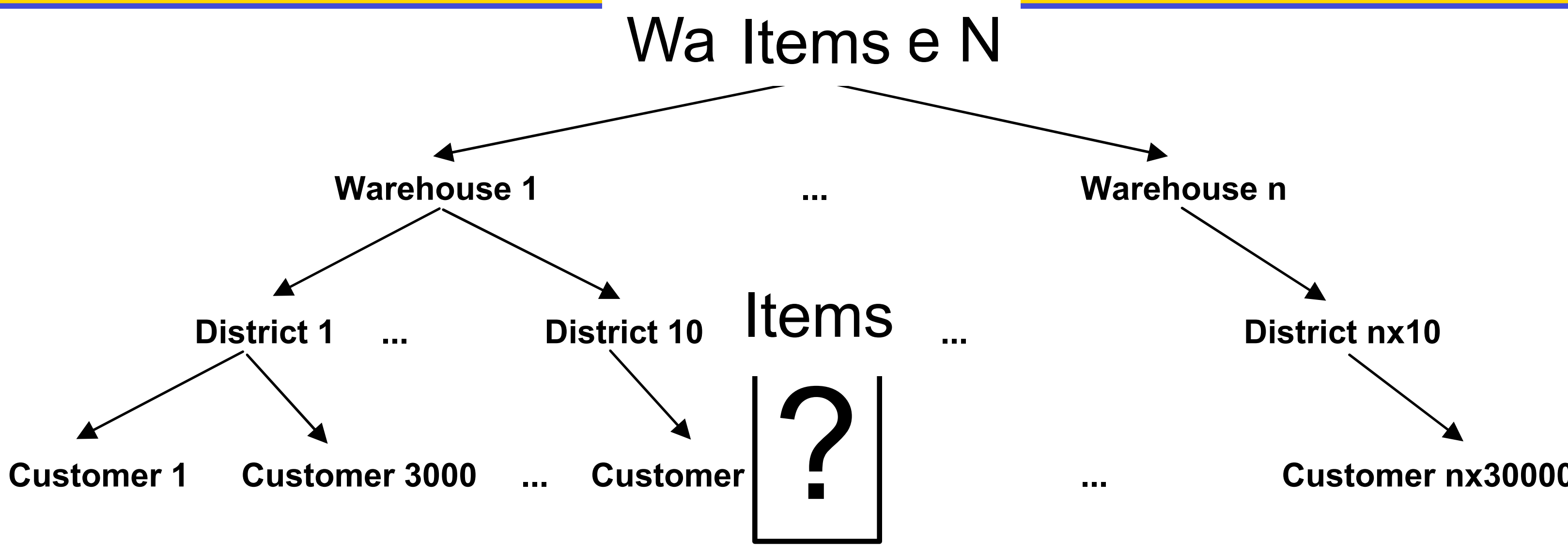
GRANOLA's lesson

- Limited Scalability with coordinated transactions:
  - Coordinated distributed transactions are implemented leveraging the classical two-phase commit
- (Almost) Perfect Scalability exploiting:
  - Single repository transactions
  - Distributed independent transactions

↓

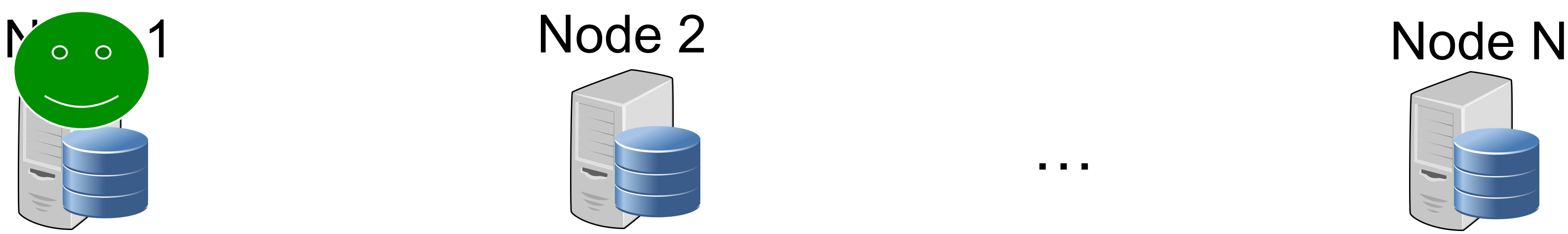
Data well partitioned

TPC-C: Example on the importance of partitioning data



<u>New-Order transaction</u>

Select Warehouse(1) -> Select District(6) -> Select Customer(11000) -> Select Items(1,2,3…) -> Do Updates



GRANOLA's limitations are our motivation

- Granola requires programmer's interventions for executing transactions e.g.,:
  - Data must be manually partitioned for maximizing the chance of executing single-repository and independent transactions
  - Programmer provides the type of each transaction invoked (either single-repository, independent transactions or coordinated).
  - Programmer provides target partitions (i.e., nodes) for each transaction invocation.

OUR GOAL

Allowing the exploitation of Granola-like transactions without involving the programmer in the process of partitioning data and instrumenting transactions

Programming Model

- Distributed Software Transactional Memory (DTM)
  - High Programmability
    - Programmer simply marks set of operations as atomic blocks (e.g., @Atomic) and the DTM library is responsible for executing those blocks (i.e., transactions) in parallel but atomically and with the given consistency level
    - Distribution and concurrency are entirely masked
  - Composability
    - Atomic operations can be composed without breaking atomicity and isolation

Partitioning Process

1. Static analysis and bytecode rewriting:
   - to collects transaction's data dependency information for verifying the compliance of the partitioning scheme with the appropriate transaction model
   - to identify whether an atomic block is abort-free or read-only
   - to tag each transactional operation with a unique identifier to help make associations between the static data dependencies and the actual objects accessed at run-time
2. Analysis of a representative trace for the current application workload
3. Generate a graph representation
4. Selection of the transactions' models

Managing the partitioning graph

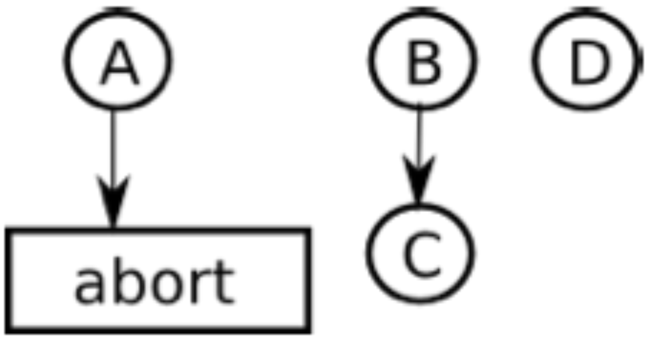
- The partitioning graph is composed of vertexes, which represent shared objects, and edges, which represent transaction's execution flow
- Principles for assigning edges' weights:
  - to fully exploit the Granola transaction model, we cannot easily allow data dependencies between partitions
  - favor single-repository transactions to any kind of distributed transactions
  - when possible, favor independent transactions to coordinated transactions

Runtime behavior

- *Placement classifiers*, in charge of maintaining the object-to-partition mapping (keeping track of the exact mapping means reproducing the entire data-set)
- *Routing classifiers*, responsible for routing transactions to correct partitions

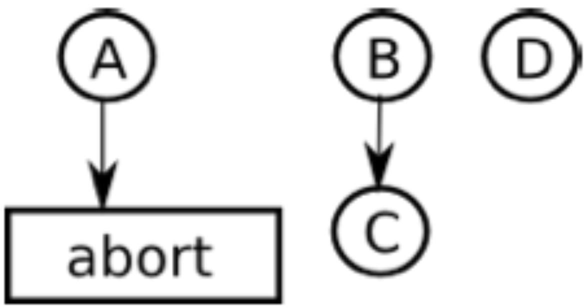
Example

```
@Atomic {
        val src1 = Open[Counter](“A”)
        If (src1.value() < 0)
                Abort-transaction
        val src2 = Open[Counter](“B”)
        val temp1 = src2.value() * 2
        val src3 = Open[Counter](“C”)
        val temp2 = src3.value() * 3
        val result = temp1 + temp2
        src3.value() = result
        val src4 = Open[Counter](“D”)
        val temp3 = src4.value() + 1
        src4.value() = temp3
        Commit-transaction
}
```

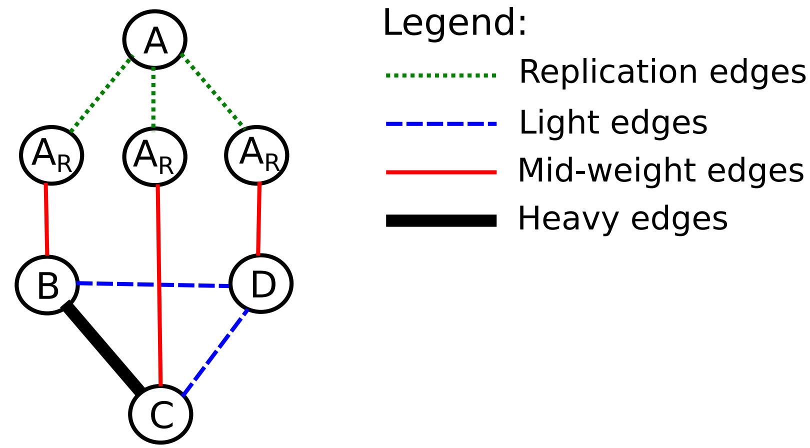


Static dependency graph

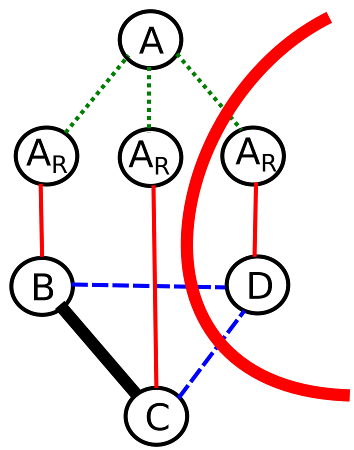
An example



Static dependency graph



Resulting partitioning graph



Possible partitioning
(pref. independent txn model)

….Summarizing…

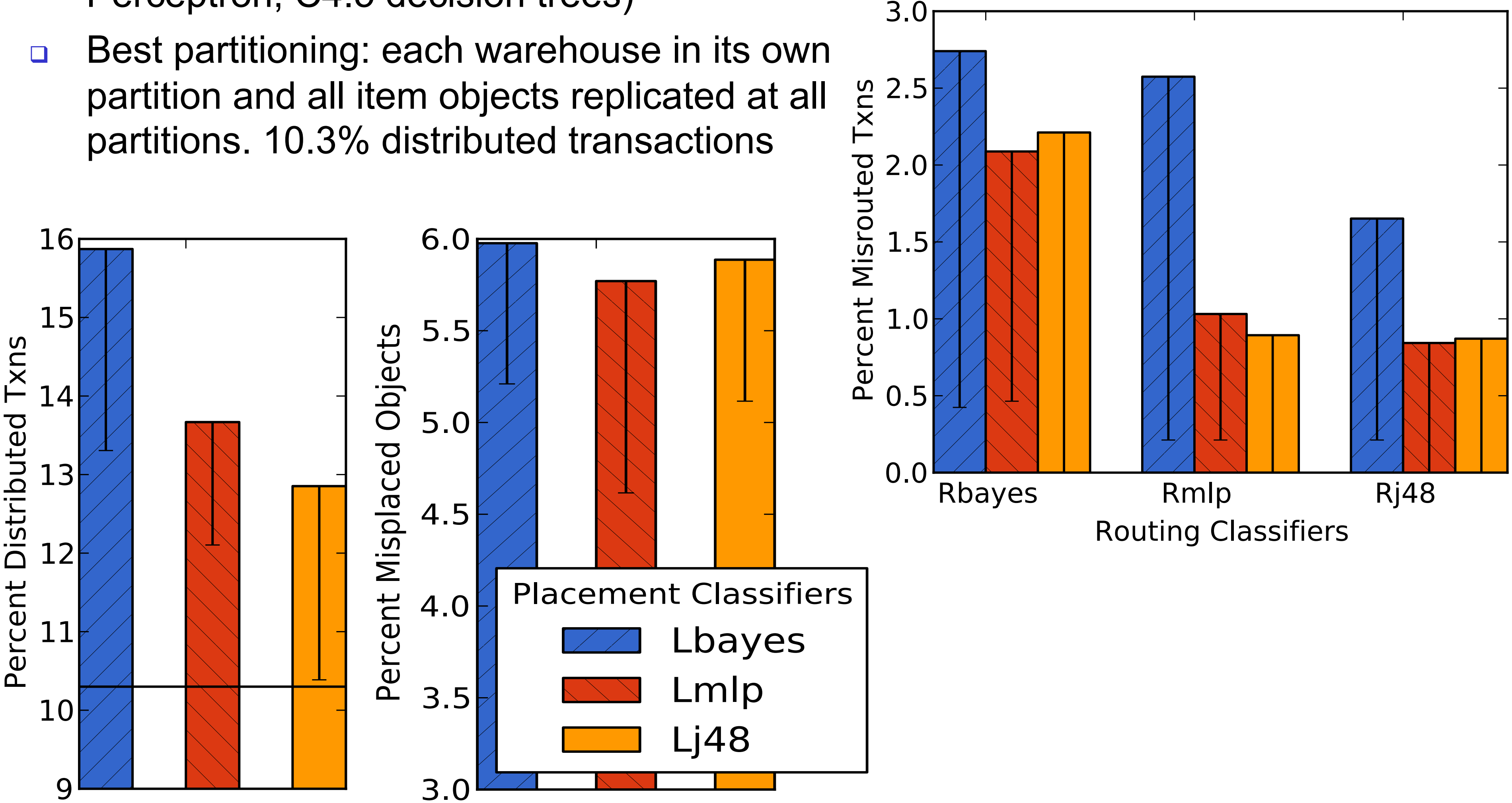
1. Bytecode analysis and re-writing
2. Gather a workload trace (e.g., running the application on a single machine)
3. Convert the trace into the graph
4. Partition the graph (using standard tools)
5. Train the placement classifiers and evaluate them (and pick the best!)
6. Train the routing classifiers and evaluate them (and pick the best!)
7. Run the population of the data-set
8. Run the application!

Evaluation

- Test-bed:
  - FutureGrid public cluster;
  - Up to 15 machines;
  - Each machine is an 8-core 2.9GHz Intel Xeon with 7GB RAM.
- Benchmark:
  - TPC-C, because its optimal partitioning scheme is known and famous.
- Performance indicators:
  - Optimality of the partitioning decisions
  - Misrouted and misplaced objects
  - Throughput and scalability
  - Partition’s quality Vs Trace Size

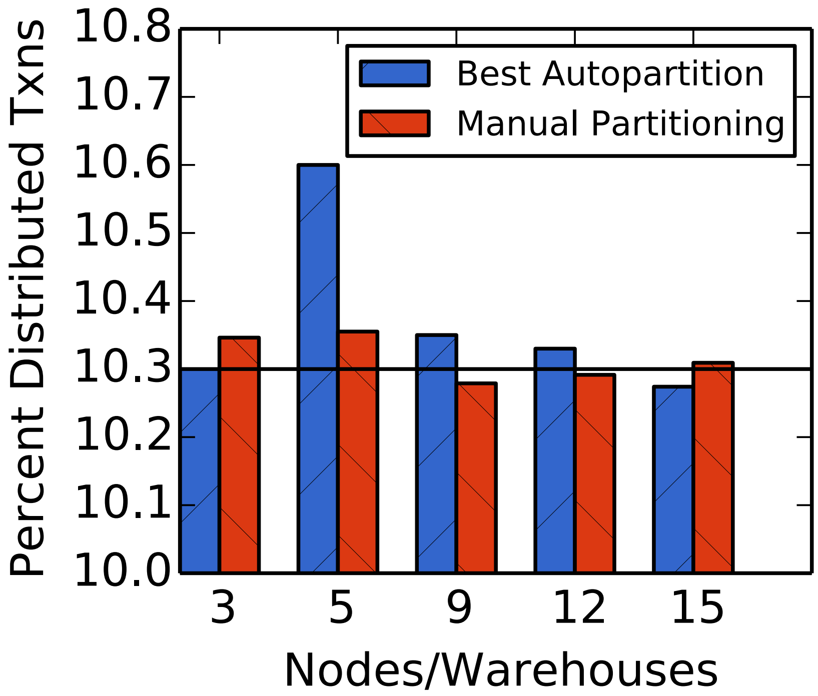
Partition and Routing Quality

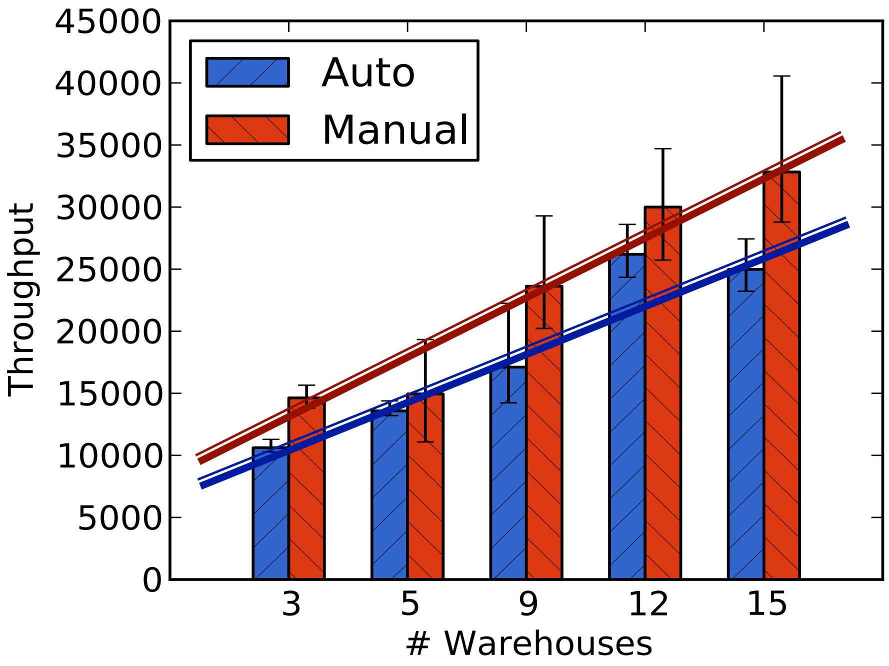
- 3 classifiers (Naive Bayes, Multi-layer Perceptron, C4.5 decision trees)
- Best partitioning: each warehouse in its own partition and all item objects replicated at all partitions. 10.3% distributed transactions



Scalability

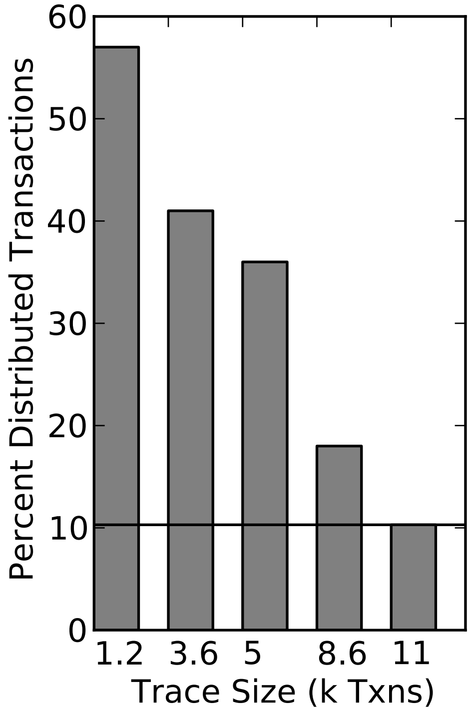
- Throughput and percentage of distributed transactions increasing the number of nodes and warehouses (and thus partitions) -- one warehouse per node/partition

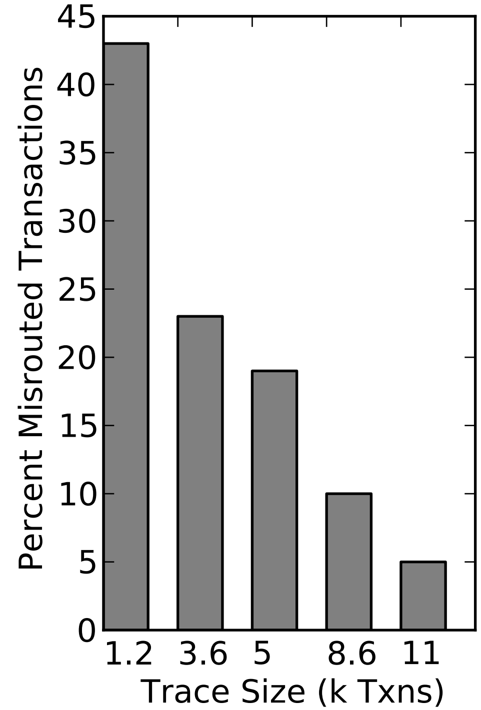




Partition Quality Vs Size of traces

- 15 warehouses
- % of misrouted transactions and distributed transactions varying the size of the trace used for computing the partitioning process and the training phase of classifiers
- The rate used for collecting samples in the execution trace

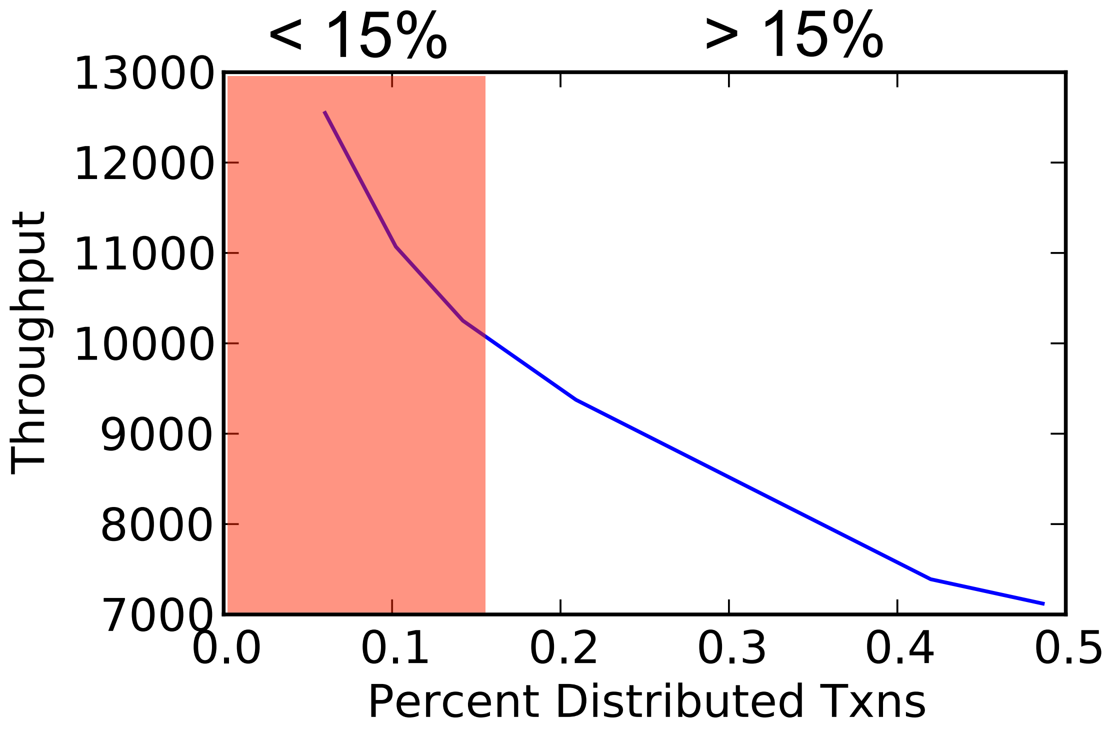




Tuple-level sampling rate	Creating graph from txn trace	METIS partitioning	Train placement classifiers	Compute partitions & train routing classifiers
5%	1m56	26s	22s	2m51s
10%	3m55	1m01s	37s	7m30s
20%	9m49	1m44s	1m02s	6m18

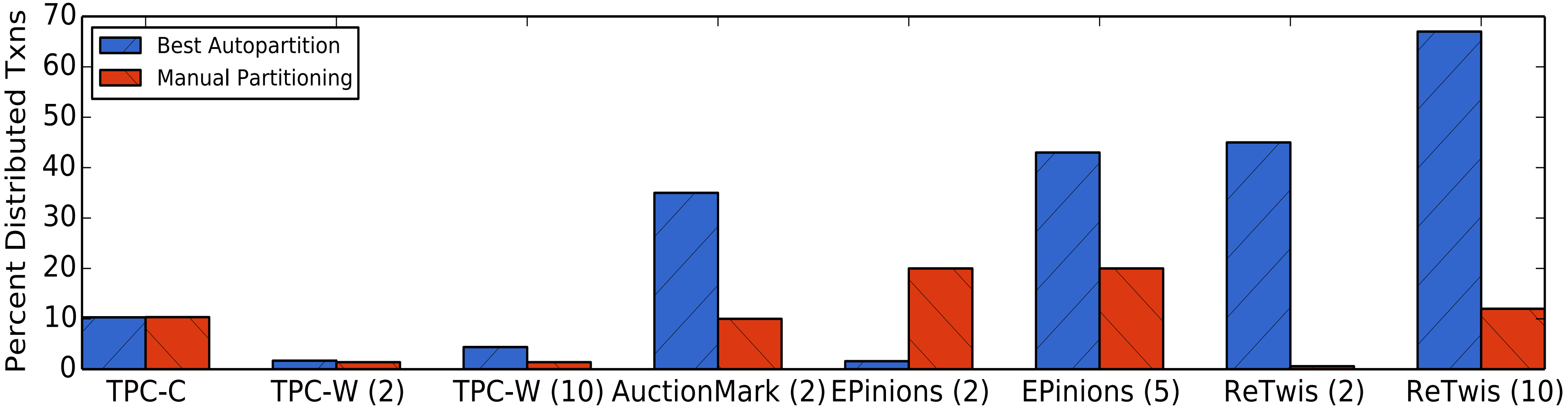
Distributed transactions and partitioning scheme

- Throughput varying the percentage of distributed transactions (we intentionally modified the transactions’ access pattern to reproduce a given percentage of distributed transactions)
- This experiment mimics also the performance of a system with non-accurate partitions



What about other benchmarks?

- We evaluated also TPC-W, AuctionMark, Epinions, ReTriss under the Granola-like transaction model to evaluate how different benchmarks can exploit independent and single-repository transactions



Thanks!

Questions?

Hyflow

Systems Software Research Group

Research project’s web-site: www.hyflow.org