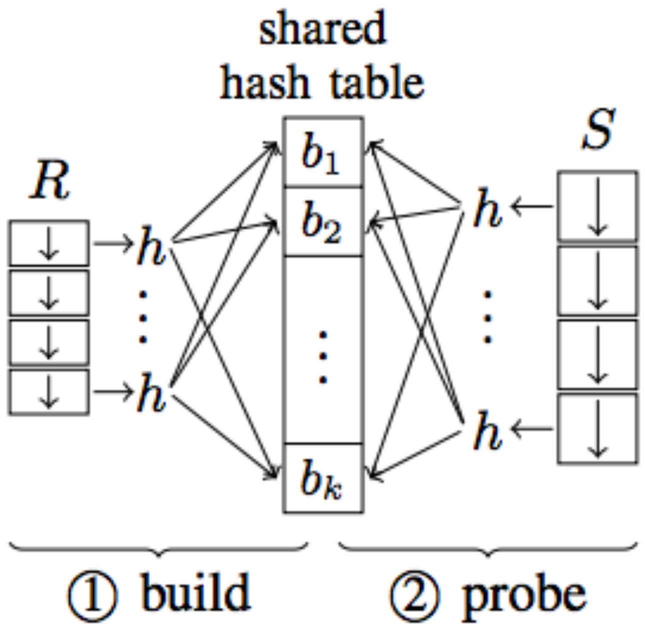
Locality-Adaptive Parallel Hash Joins using Hardware Transactional Memory

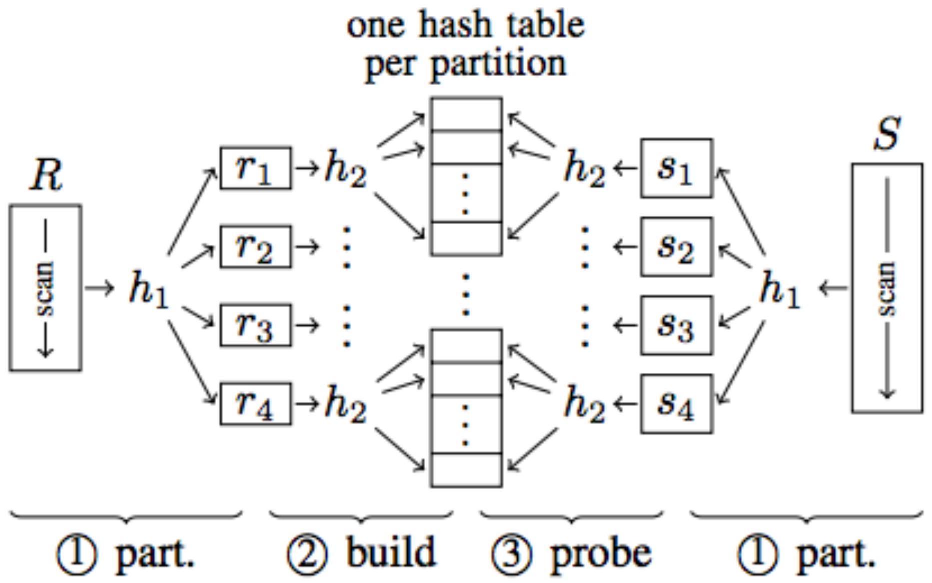
ANIL SHANBHAG, HOLGER PIRK, SAM MADDEN

MIT CSAIL

History of Parallel Hash Joins



Shared Hash Table based Join



Radix Partitioning based Join

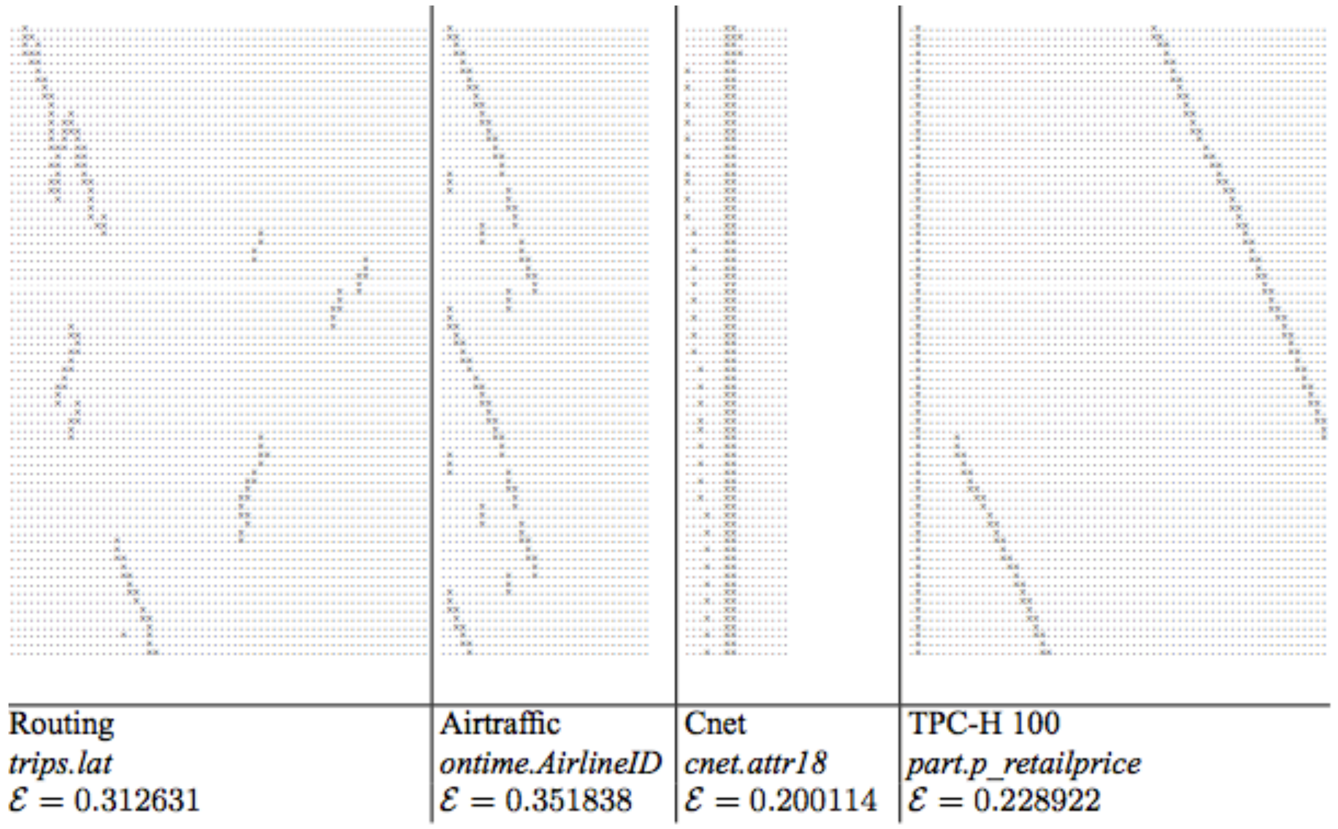
Pictures from “Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware” Balkesen et al.

Motivation

Data can have spatial locality

May arise because of :

- Periodic bulk updates => Locality in date and correlated attributes
- Trickle loading in OLTP systems => Locality in date
- Automatically assigned IDs => monotonically increasing counters



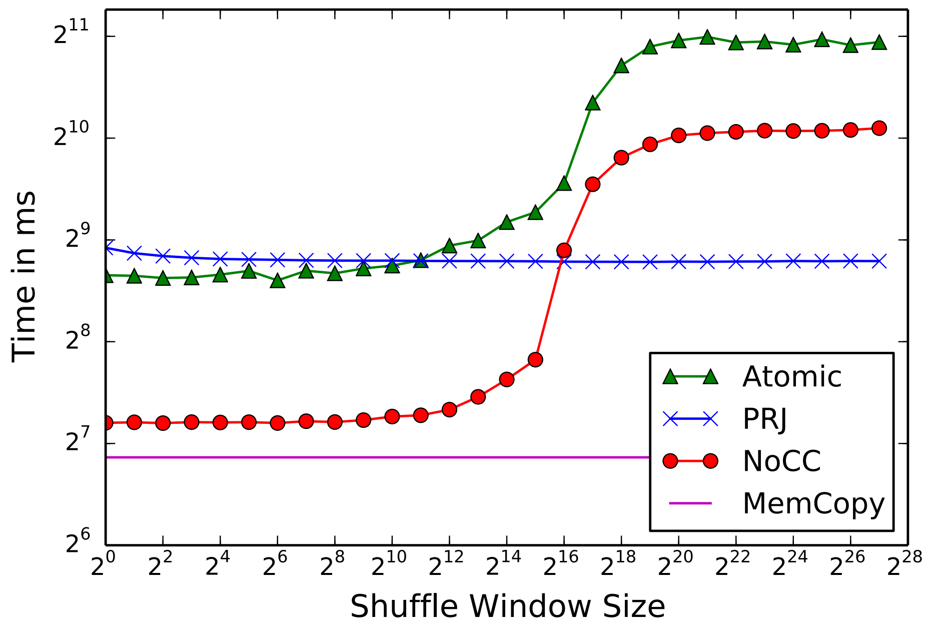
From “Column Imprints: A Secondary Index Structure” Sidirourgos et. al, SIGMOD 13

Motivation

Simple experiment: Compare the time of hash building phase of 3 approaches:

- Global hash table using atomics (Atomic)
- Parallel Radix Join (PRJ)
- Global hash table with no conc. Control (NoCC)

NoCC is incorrect; existing approaches are > 3x slower than it.



Can we do as good as NoCC ?

Yes we can !

Rest of this talk:

- Using HTM to achieve better performance
- Making HTM-based hash join self-tuning
- Adaptively fall back to Radix Join

Hardware Transactional Memory

Sequence of instructions with ACI(D) properties

```
Balance Transfer {
    Lock()
    A_balance -= 10
    B_balance += 10
    Unlock()
}
```

Using Global Lock

```
Balance Transfer {
    A.lock() B.lock()
    A_balance -= 10
    B_balance += 10
    B.unlock() A.unlock()
}
```

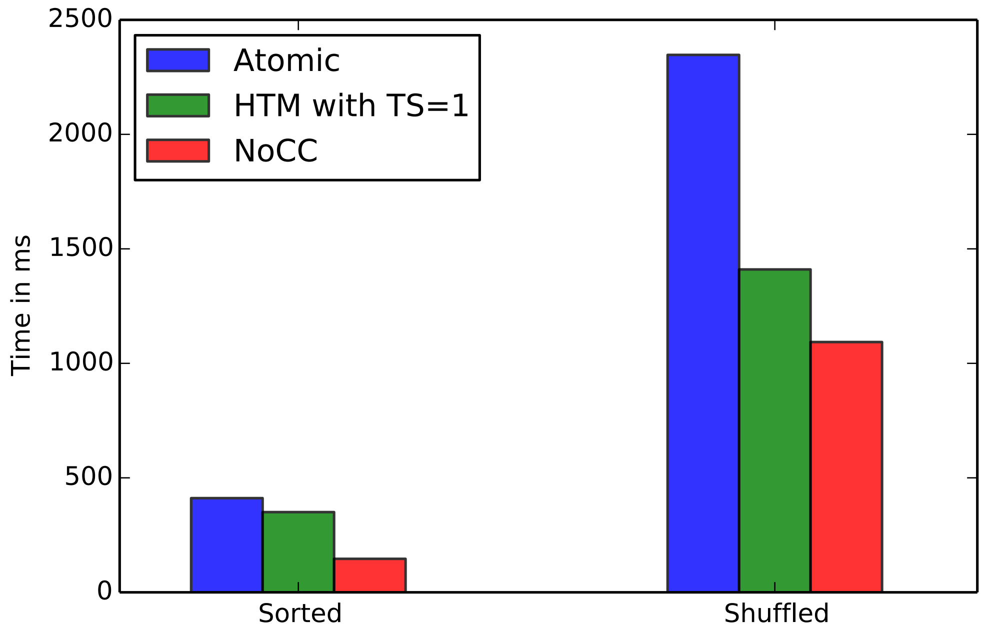
Using Fine Grained Locks

```
Balance Transfer {
     _xbegin()
    A_balance -= 10
    B_balance += 10
    xend()
}
```

Using HTM

Intel Haswell uses L1 Cache as staging

HTM vs using atomics

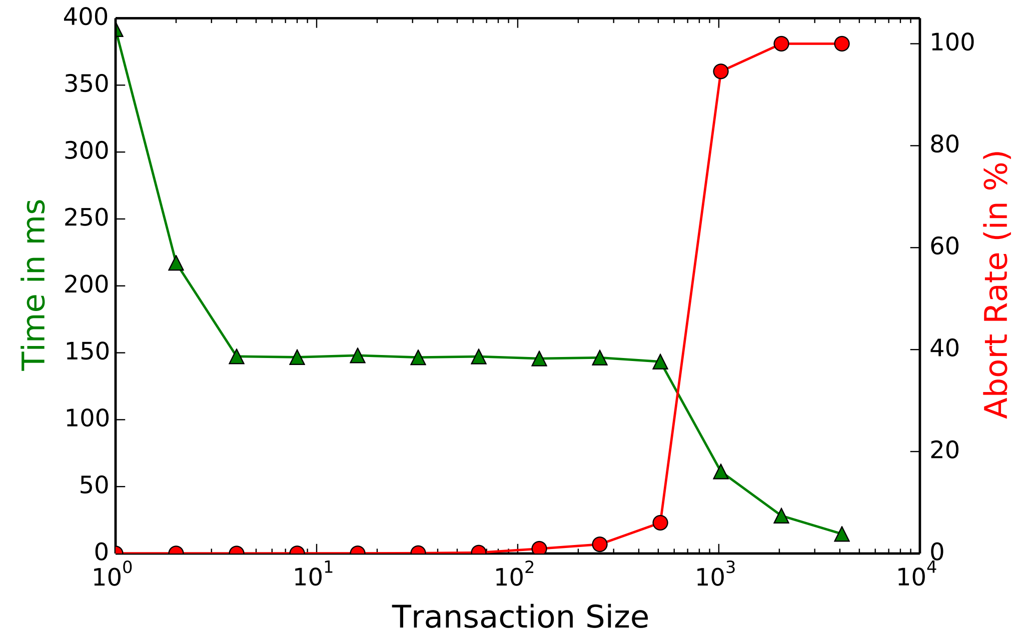


Gap between HTM and NoCC is the overhead of using HTM

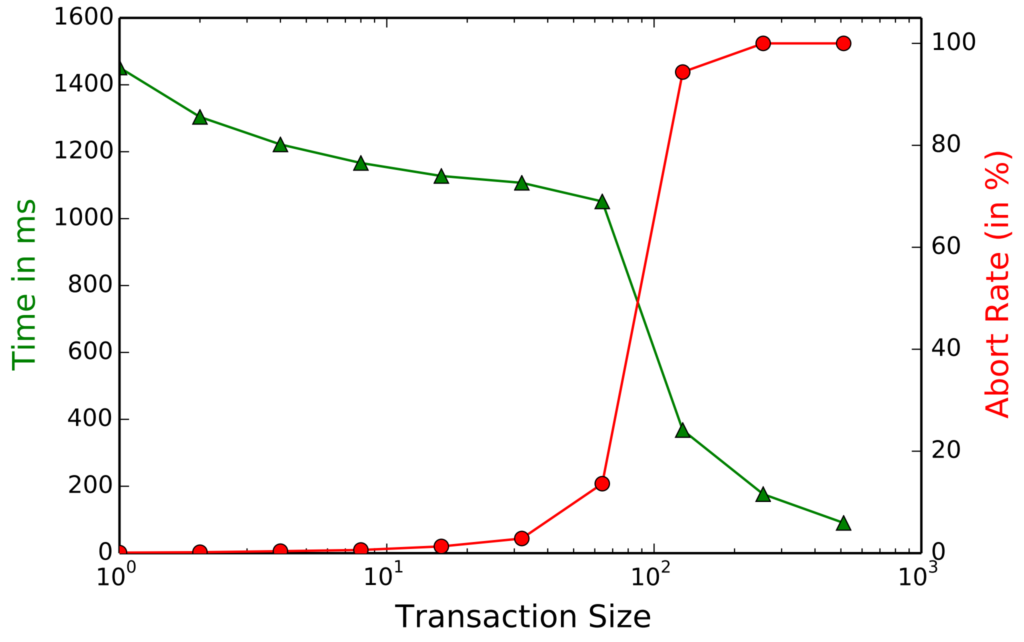
HTM does better than Atomic always. The larger gap for shuffled data shows the overhead of doing atomic operation vs optimistic load/store.

Reducing Transaction Overhead

To reduce the transaction overhead, do multiple insertions per transaction.

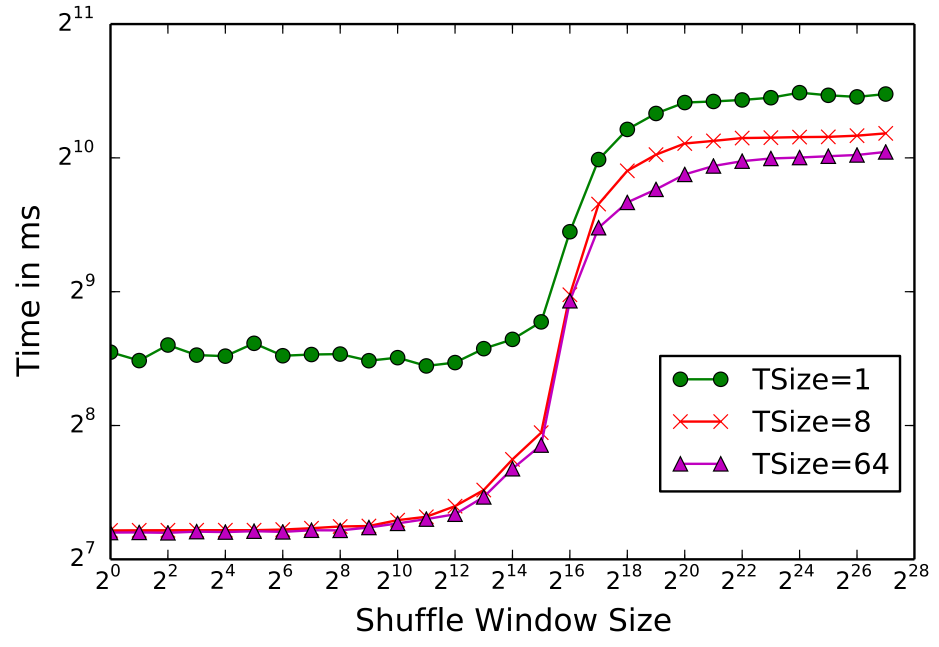


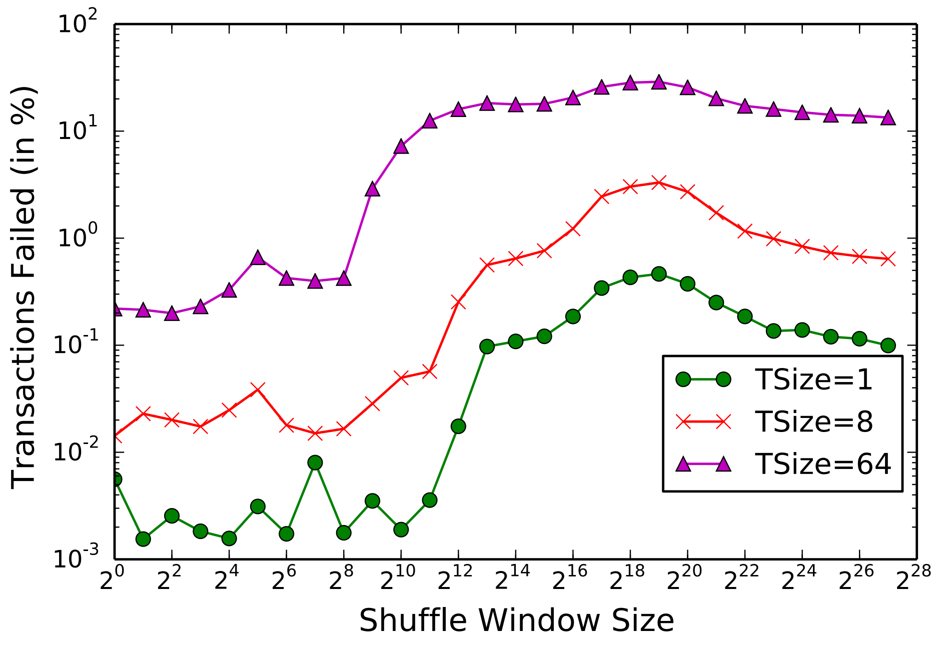
Sorted Data



Shuffled Data

Wrt Data Locality





Our Hash Table So-Far

```
struct Bucket:
  Tuple tuples[3]
  int count
  int nextBucketIndex

HashTable table
table.buckets = Bucket[ceil(numTuples/3)]
```

```
// Each thread gets an input range [start,end)
for (i = start; i<end; i += transactionSize):
  status = _xbegin()
  if status == _XBEGIN_STARTED:
    for (j = i; j < i + transactionSize; j++):
      slot = hash(tuple[j].key)
      for (k = slot; k < slot + probeLength; k++):
        if table.buckets[k] is not full:
          table.buckets[k].add(tuple); break
      if not inserted:
        overflow.add(tuple)
    xend()
  else: // Transaction Failed
    failedTransactionRanges.add(i)
```

```
// Wrap-Up
for (i in failedTransactionRanges)
  // Insert tuples[i] to tuples[i + transactionSize]
for (i in overflow)
  // Insert overflow[i]
```

Adaptive Transaction Size Selection

Transaction size remains a variable that would require manual tuning

Optimal performance hinges on appropriate selection of the transaction size

Our simple adaption strategy:

- Start with TS = 16
- Process input in batches of 16k tuples and monitor abort rate
- If abort rate > high-watermark: TS /= 2
- Else if abort rate < low-watermark: TS *= 2

We chose 0.4% as low and 2% as high

Fallback for fully-shuffled data

With sufficient locality, the HTM-based approach performs best

For large shuffle windows, radix join performs better

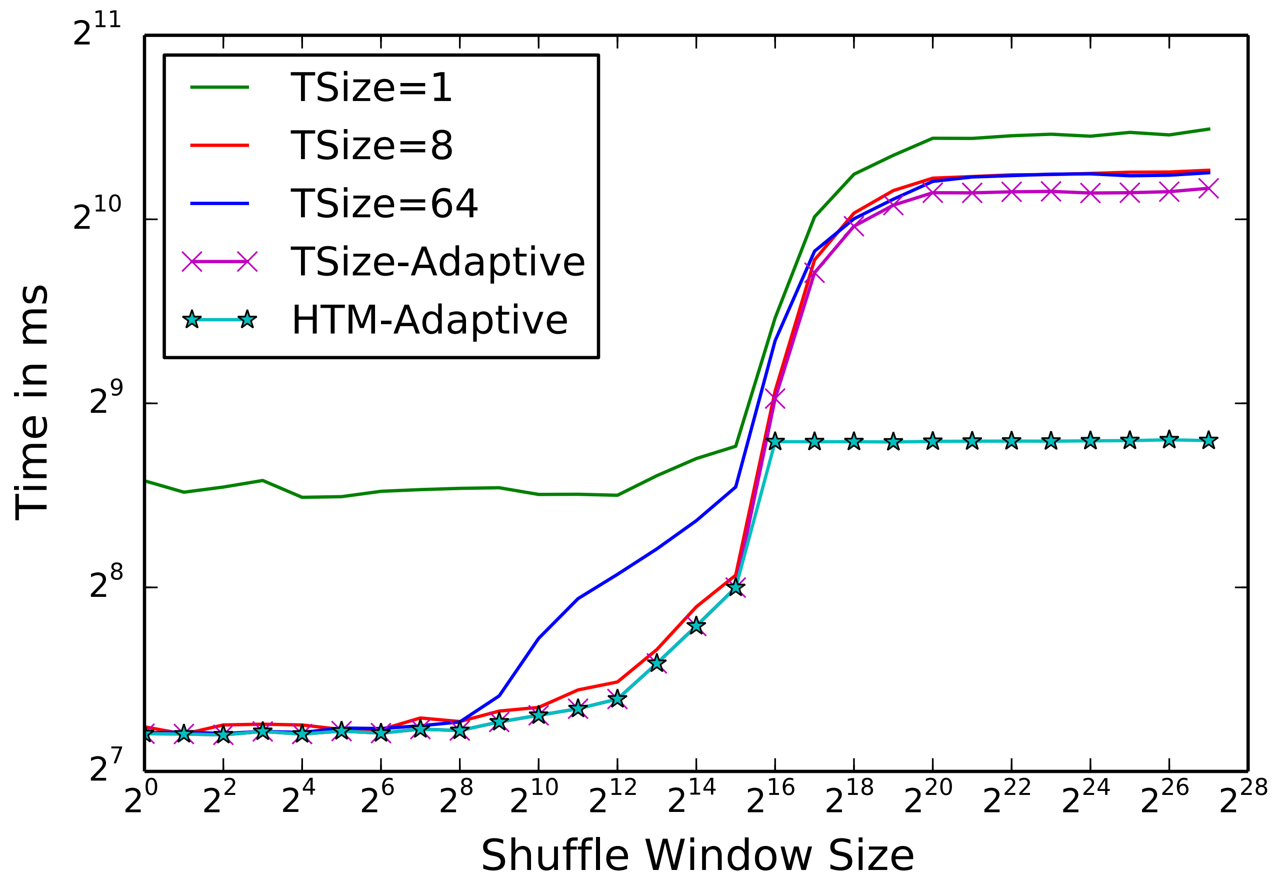
Key Insight: Larger shuffle windows also coincide with high transaction abort rates

Hybrid approach:

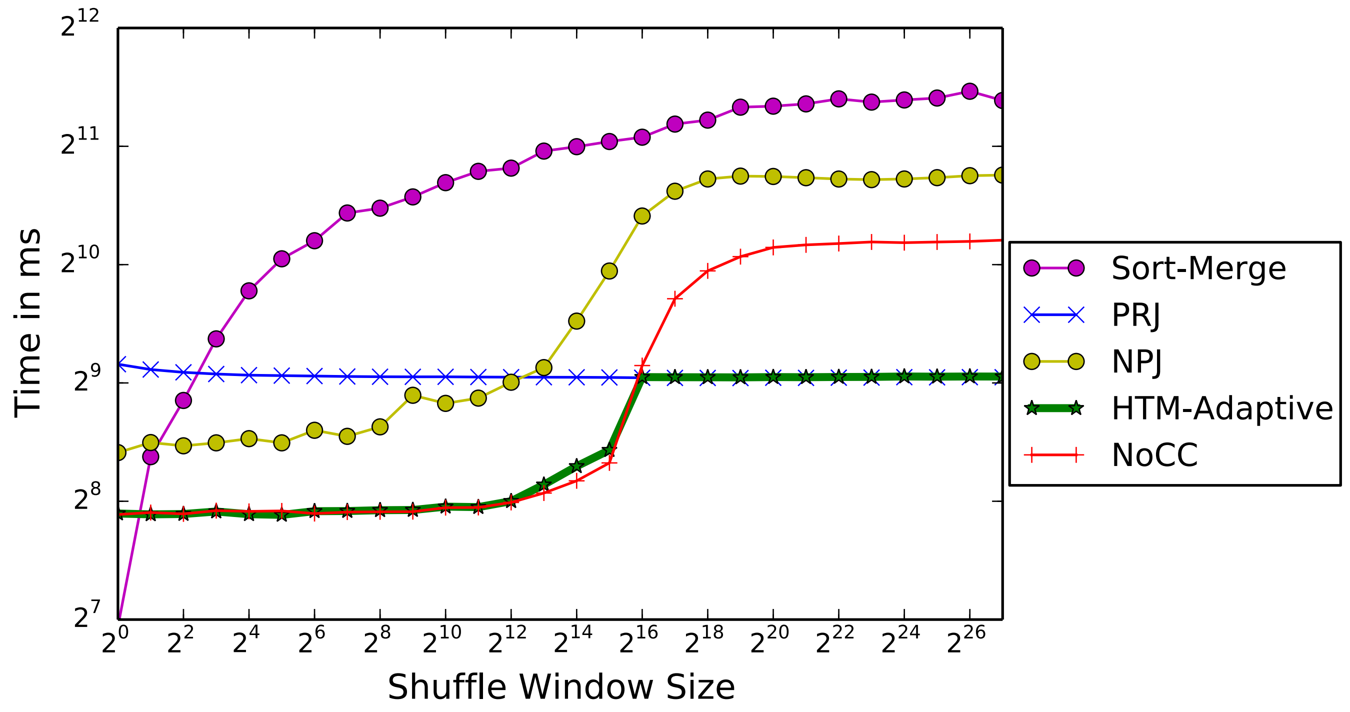
- Process first batch of 16k tuples on each thread and inspect abort rate (takes ~ 4ms)
- If abort rate > threshold: Switch to do radix join

We found threshold = 4% appropriate for our experiments

Build Phase Performance



Complete Hash Join (with probe)



Also compare against No-Partitioning Join (implemented by Balkesen et al.) and Sort Merge Join based on TimSort

HTM-Adaptive matches/beats all the approaches

Conclusion

HTM is great for low-overhead fine-grained concurrency control

HTM-based hash building with adaptive transaction size comes very close to memory bandwidth for data with locality

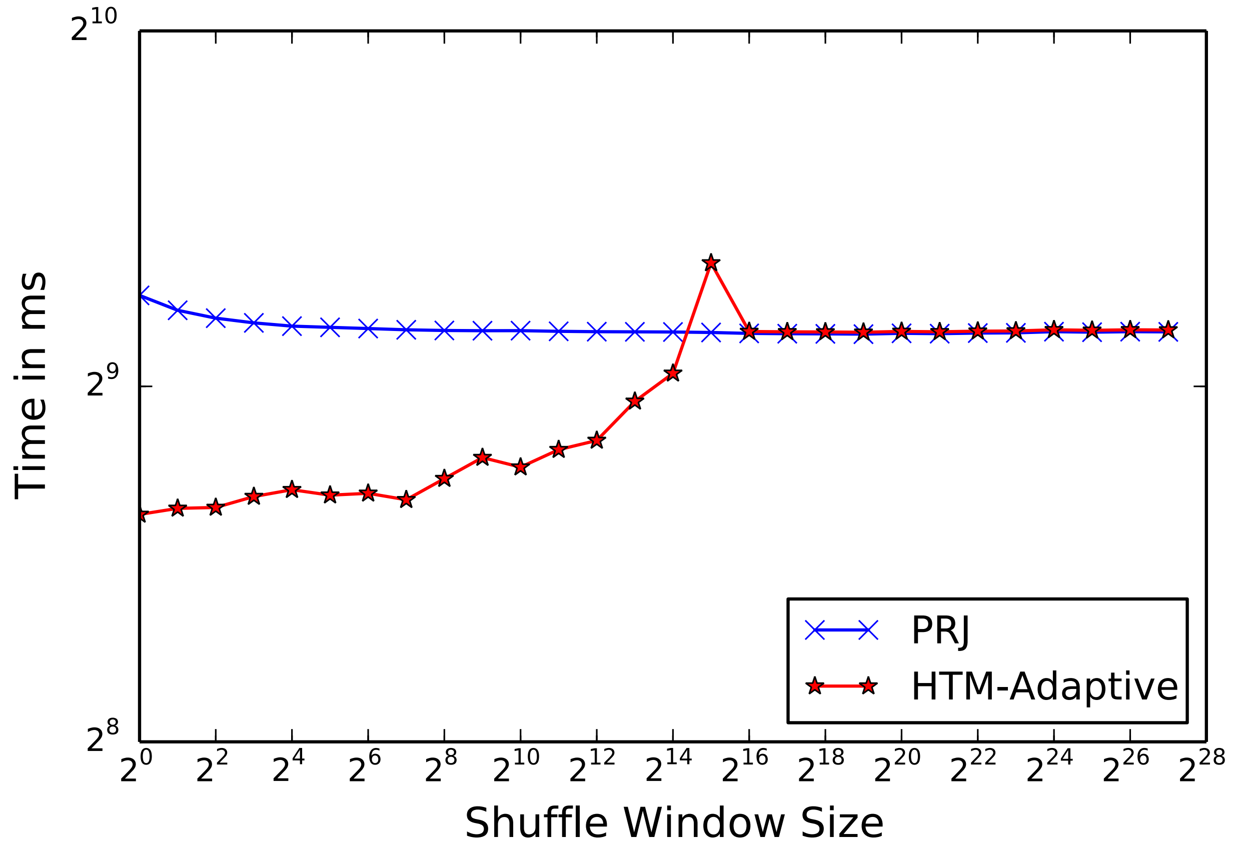
Abort rates can be used to detect lack of locality and fallback to radix join

The resulting join algorithm is the best global hash table based approach

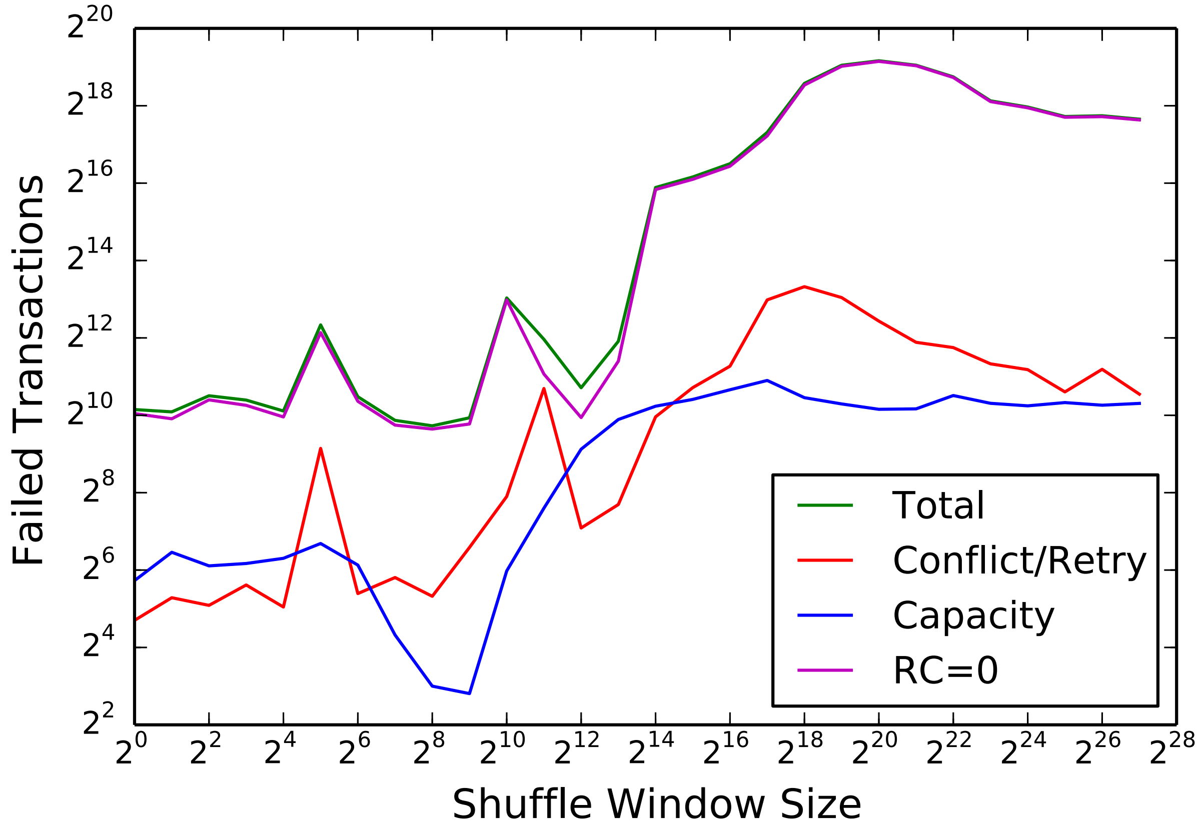
- Beats radix join by 3x on data with locality
- Falls back to radix join in the absence of it.

Thank You ☺

Performance on Uniform Data



Abort Code ?



Failed Transactions
Shuffle Window Size
Total
Conflict/Retry
Capacity
RC=0
2^{2}
2^{4}
2^{6}
2^{8}
2^{10}
2^{12}
2^{14}
2^{16}
2^{18}
2^{20}
2^{0}
2^{2}
2^{4}
2^{6}
2^{8}
2^{10}
2^{12}
2^{14}
2^{16}
2^{18}
2^{20}
2^{22}
2^{24}
2^{26}
2^{28}