First International Workshop on Business Intelligence for the Real Time Enterprise
September 11, 2006 Seoul, Korea

Transaction Reordering and Grouping for Continuous Data Loading

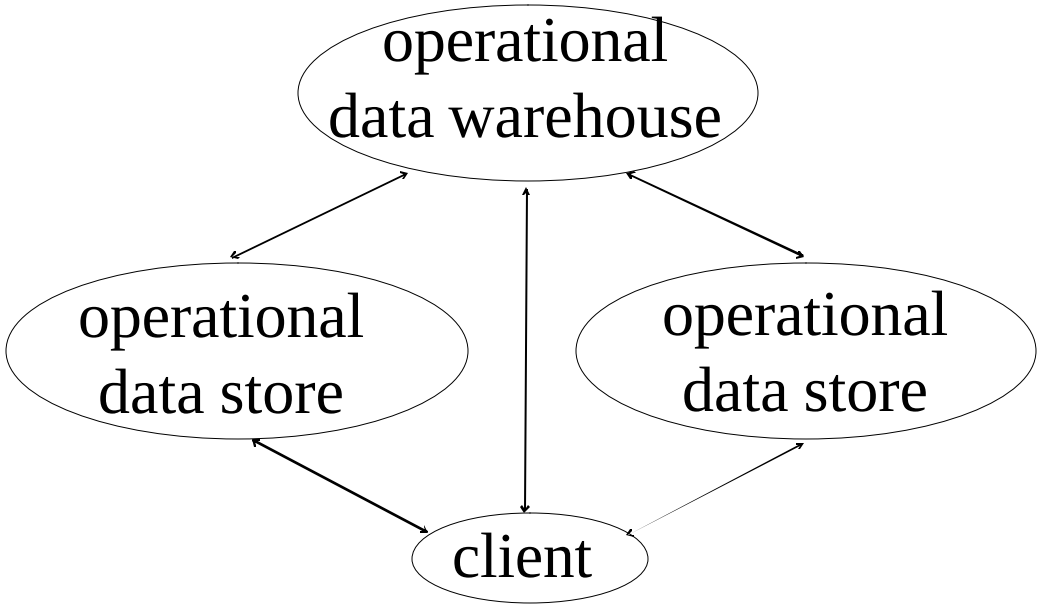
Gang Luo[1] Jeffrey F. Naughton[2]
Curt J. Ellmann[2] Michael W. Watzke[3]

IBM T.J. Watson Research Center[1]
University of Wisconsin-Madison[2] NCR[3]

luog@us.ibm.com naughton@cs.wisc.edu
ellmann@wisc.edu michael.watzke@ncr.com

Continuous Data Loading

- For operational data warehousing, it is critical that the data in the warehouse be as up-to-date as possible
- Commercial RDBMS vendors (Oracle, Teradata, etc.) provide tools for continuous data loading



Immediate Materialized View Maintenance

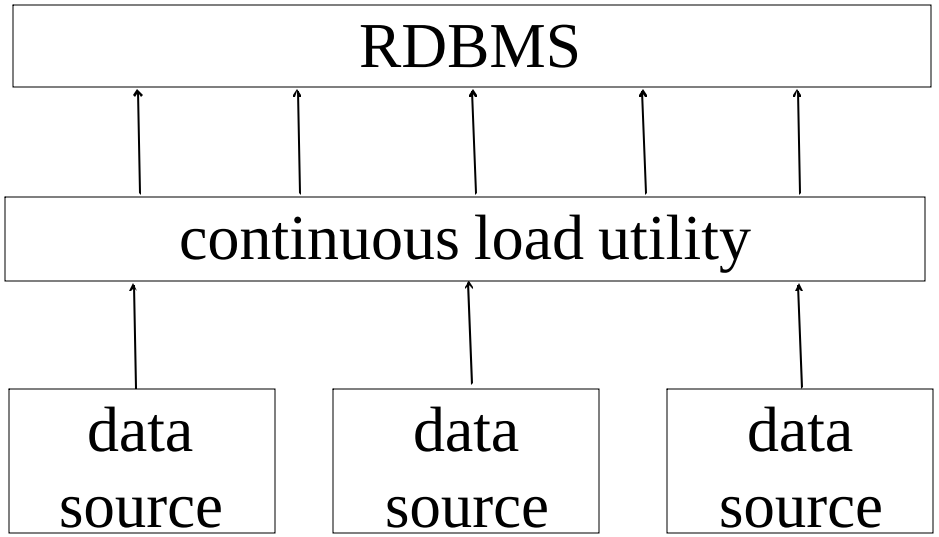
- Necessity:
  - Inconsistency between materialized views and base relations may not be acceptable to all applications (e.g., real-time decision making)
  - Mandated in the TPC-R benchmark
- Problem: Concurrent updates to multiple base relations of the same materialized join view cause deadlocks
  - Reason: Materialized join view maintenance changes update-only transactions to update-read transactions
- Solution: Reorder the updates

Outline

- Existing Continuous Load Utilities
- Problem with Immediate Materialized Join View Maintenance
- Solution with Reordering
- Performance

Continuous Data Loading Architecture

- Data comes in the form of modification operations (insert, delete, update)
- Load data in the form of transactions through sessions



Assumptions

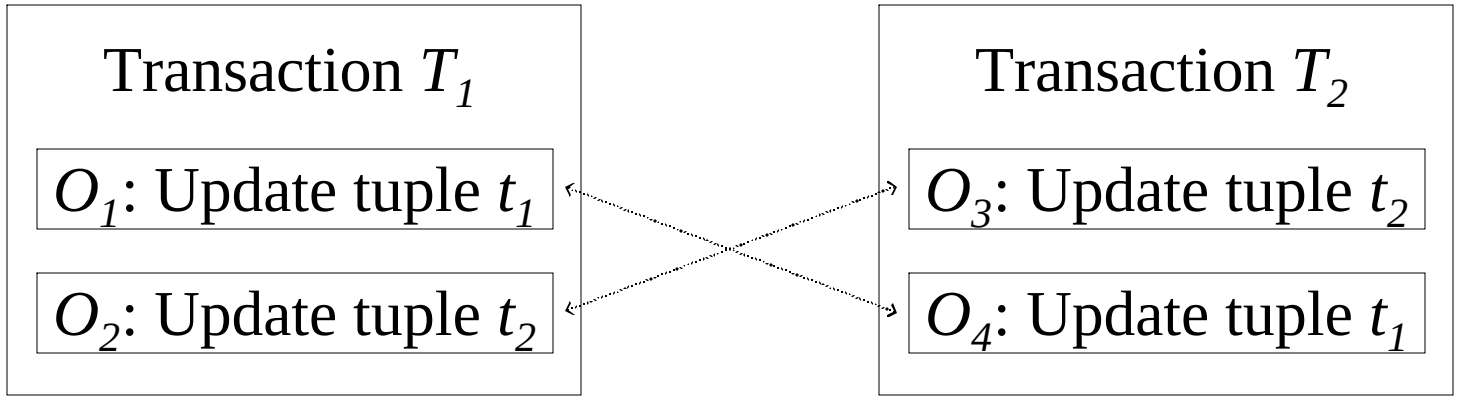
- The system has no control over the order in which modification operations arrive at the RDBMS
- The continuous load utility looks to the RDBMS like a series of transactions, each containing a single modification operation
- No order imposed or assumed for the load transactions
  - The load process can arbitrarily reorder the single-modification-operation transactions
- No requirement on whether multiple modification operations can or cannot commit/abort together
  - The load process can arbitrarily group the single-modification-operation transactions

Assumptions - Continued

- Use strict two-phase locking protocol
- Use multiple-granularity locking protocol
  - Two levels in the locking hierarchy: table-level locks and tuple-level locks
- Each modification operation can be done with tuple-level locks

Common Wisdom

- Grouping to improve efficiency
  - Combine multiple modification operations into a single transaction
- Partitioning to avoid deadlock
  - Partition tuples among different sessions
  - Modification operations on the same tuple are always through the same session



Outline

- Existing Continuous Load Utilities
- Problem with Immediate Materialized Join View Maintenance
- Solution with Reordering
- Performance

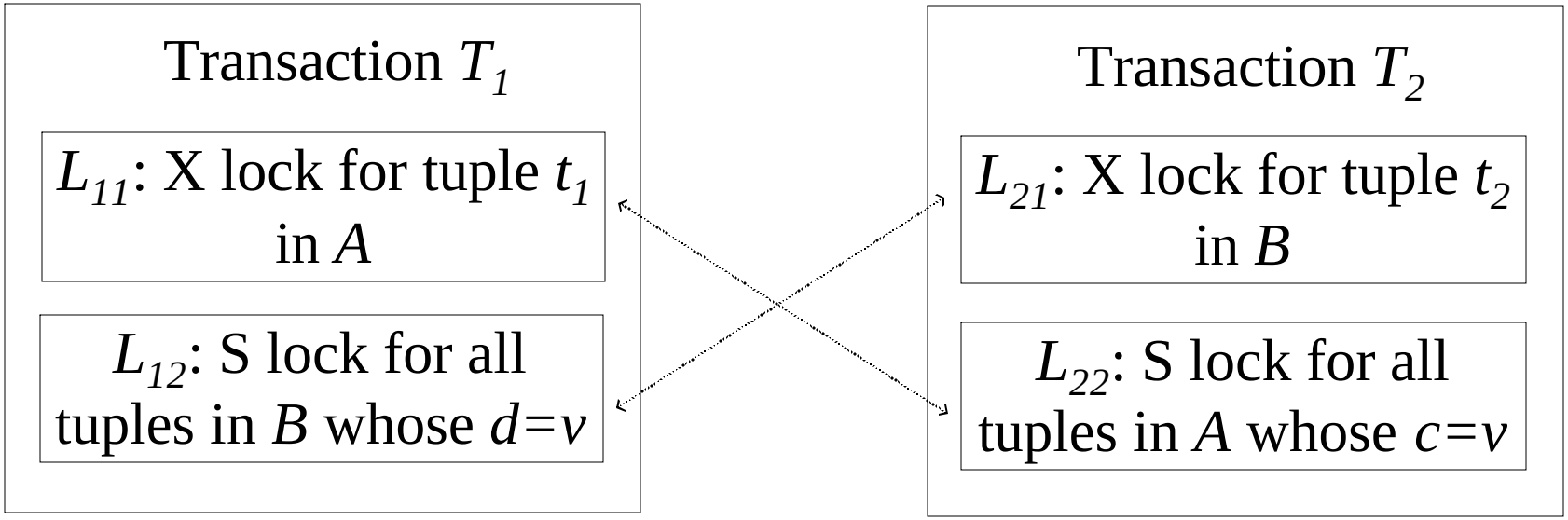
Join View

- A join view *JV* on two relations *A* and *B* pre-computes a selection and projection (and aggregation) of *A* *B*

```
create join view JV as
select *
from A, B
where A.c=B.d;
```

Impact of Immediate Join View Maintenance

- Two transactions
  - T_1: Modify tuple t_1 in A whose $c=v$
  - T_2: Modify tuple t_2 in B whose $d=v$



Deadlock Probability

- $k>1$ concurrent transactions
- Each transaction T contains n modification operations
  - T modifies A with probability p
  - T modifies B with probability $1-p$
  - Each modification operation modifies a random tuple in A (B)
- Totally s distinct values for $A.c$ ($B.d$)
- Deadlock probability: $min(1, p(1-p)(k-1)n^2/(2s))$

Outline

- Existing Continuous Load Utilities
- Problem with Immediate Materialized Join View Maintenance
- Solution with Reordering
- Performance

Intuition for Transaction Reordering

- Only run “compatible” transactions concurrently
  - Reorder transactions so that transactions updating *A* are executed, then transactions updating *B* are executed

Rules for Transaction Reordering

- **Rule 1**: At any time, for any join view *JV*, only allow data to be loaded into one base relation of *JV*
- **Rule 2**: For those modification operations on the same base relation, use the partitioning method
- **Rule 3**: Use a high concurrency locking protocol (e.g., the V locking protocol) on join views

Justification

- Rules 1 and 2 avoid deadlocks resulting from lock conflicts on the base relations
- Rule 3 avoids deadlocks resulting from lock conflicts on the join views

Data Structures for Transaction Reordering

- For each base relation R_i, maintain a number J_i
  - J_i records the number of running transactions that modify R_i
- For each session S_m, maintain a queue Q_m
  - Q_m records the transactions waiting to be run through S_m

Transaction Reordering Algorithm Outline

- Analyze potential lock conflicts using the data structures
  - Transaction starts execution: Increment J_i
  - Transaction finishes execution: Decrement J_i
- Desirable transaction updating R_i: Does not conflict with any running transaction
  - Check whether or not $J_j = 0$ for each base relation R_j of the same join view as R_i

Outline

- Existing Continuous Load Utilities
- Problem with Immediate Materialized Join View Maintenance
- Solution with Reordering
- Performance

Performance

- Testing environment
  - Commercial parallel RDBMS
  - Windows 2000 OS
  - Intel workstation, each data server node has one 400MHz CPU, 250M memory, one 8GB disk
- System configuration in the parallel RDBMS
  - Allocate a processor and a disk for each data server
  - Test system configuration has four data server nodes

Performance - Continued

- Definition of base relations and join view:

  demand (partkey, date, quantity, <u>custkey</u>, comment)
  inventory (<u>partkey</u>, <u>date</u>, quantity, extended_cost, extended_price)

  ```
  create join view onhand_demand as
  select d.partkey, d.date, d.quantity, d.custkey, i.quantity
  from demand d, inventory i
  where d.partkey=i.partkey and d.date=i.date
  partitioned on d.custkey;
  ```

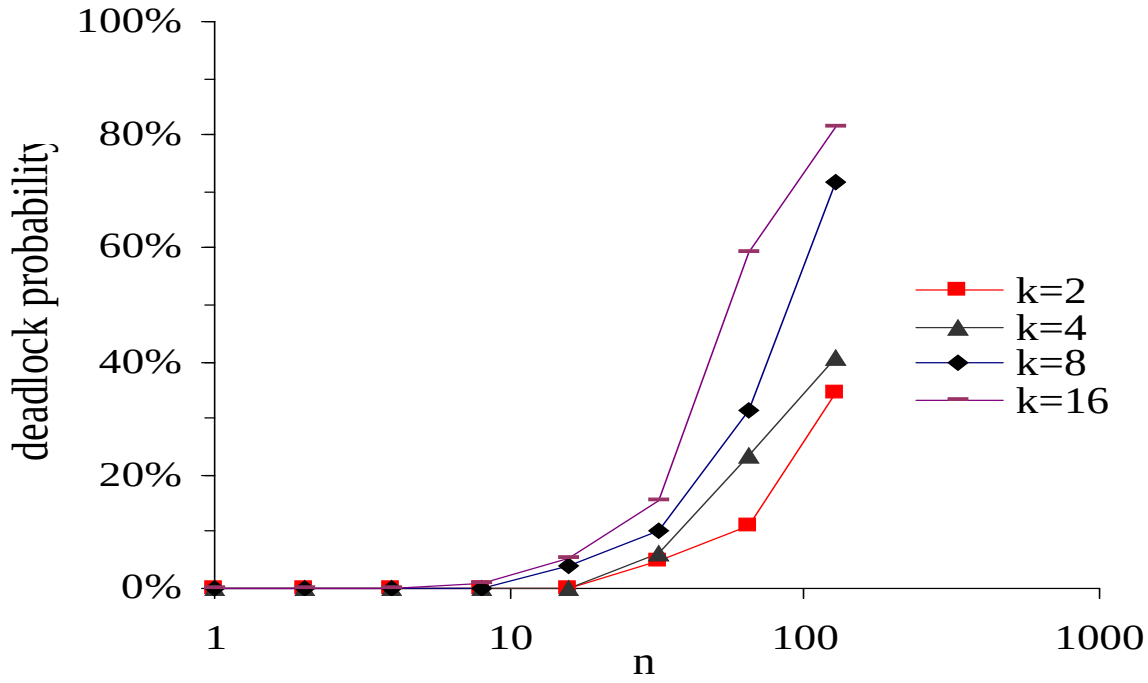
- Modification operations:
  - O_1: Update one tuple in the *inventory* relation with a specific *partkey* value and today's *date*
  - O_2: Insert one tuple into the *demand* relation with a specific *partkey* value and today's *date*

Performance - Continued

- Compare the naive method vs. the transaction reordering method
  - Run a stream of T_1's and T_2's (50% are T_1's, 50% are T_2's)
  - Only combine modification operations on the same base relation into a single transaction
  - Each transaction contains the same number *n* of modification operations
  - 10,000 active parts
  - *k* sessions
  - Measure throughput of modification operations

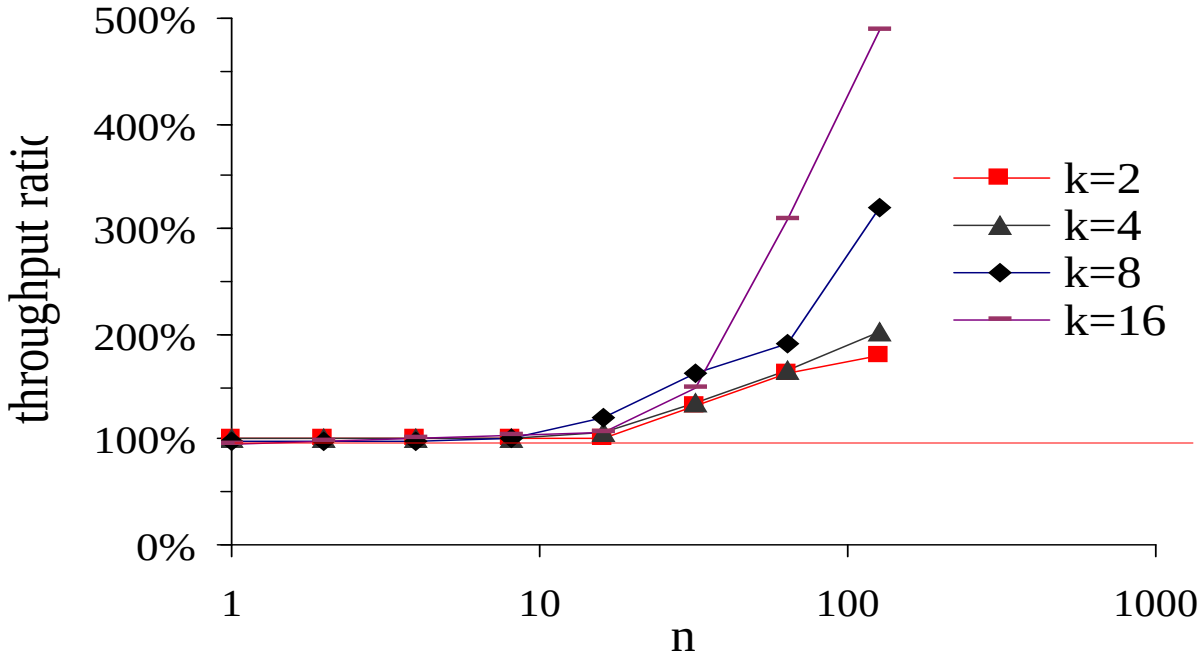
Performance - Deadlock Probability

- Deadlock probability of the naive method increases with both *n* and *k*



Performance - Throughput Improvement

- Improvement of throughput gained by transaction reordering increases with both *n* and *k*



Questions?