AALBORG UNIVERSITY

RiTE: Providing On-Demand Data for Right-Time Data Warehousing (ICDE’08)

Christian Thomsen, Torben Bach Pedersen, Wolfgang Lehner

daisy

Center for Data-intensive Systems

Agenda

- Motivation
- The RiTE package
- Producer side
- Catalyst side
- Consumer side
- Performance study
- Summary and future work

daisy

Motivation

- Traditionally DWs have been loaded at regular time intervals
  - This is done by using fast *bulk loading*
- A recent trend is to load data into the DW minutes or seconds after it arrives
  - This called "*near-realtime data warehousing*"
- A more sophisticated approach considers that some data must be fresh while other data can be less fresh
  - Parts of the data loaded quickly after arrival, other parts loaded at regular intervals
  - In other words, the data is loaded at the right time and we call this "*right-time data warehousing*"
- Bulk loading is not efficient to use when the data sizes are small
- For near-realtime and right-time data warehousing, regular SQL INSERT statements are used, leading to slow insert speed

daisy

A solution: RiTE

- There is a great need to be able to make data quickly available, but retain the insert speeds of bulk loading
- The solution should find the correct batch size between the two extremes (bulk load vs. single-row INSERT) and the right time to make data available in the DW
- Data should be available in the DW when needed, but not necessarily before
- We can exploit some DW characteristics:
  - One producer (ETL application)
  - Lower persistency requirements (the data can be reloaded from the source systems) in case of a crash during load
- We propose the middleware ***RiTE*** which provides a solution
- The producer can continuously insert data at bulk-load speed, but consumers have access to fresh data

daisy

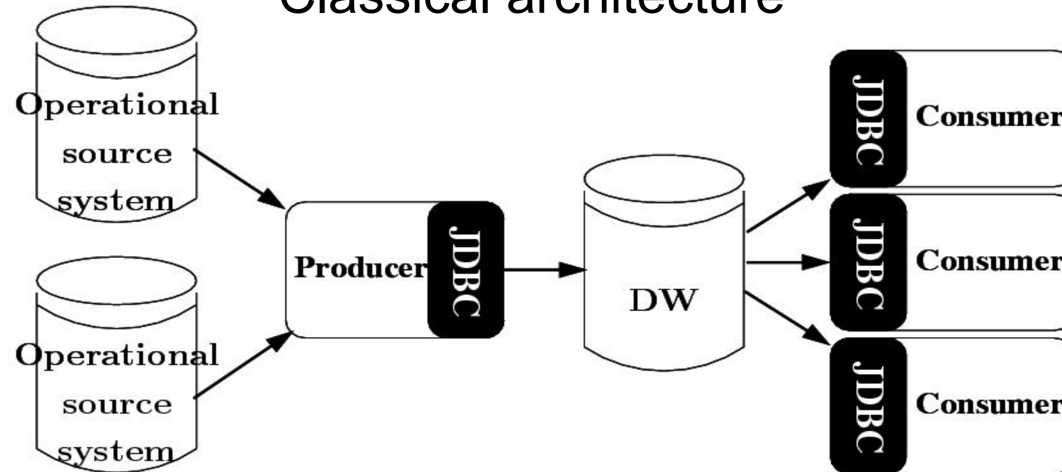
The RiTE package

- A specialized JDBC database driver for the producer
- A specialized JDBC database driver for the consumers
- A main-memory based *catalyst*
  - Provides intermediate storage ("*memory tables*") for (user-chosen) DW tables
  - Offers fast insertions
  - Offers concurrency control
  - The data can be queried while held by memory tables
    - This can be done transparently to the end user
  - Eventually the data is moved to its final target – the physical DW tables
- A PostgreSQL *table function* makes the data available in the DW

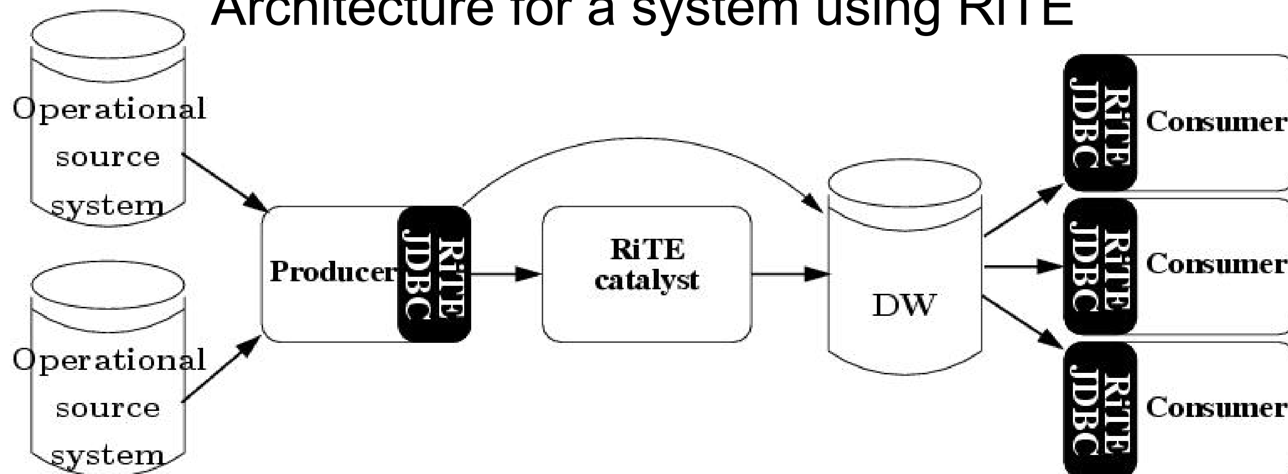
daisy

Architecture

Classical architecture



Architecture for a system using RiTE



daisy

Agenda

- Motivation
- The RiTE package
- ***Producer side***
- Catalyst side
- Consumer side
- Performance study
- Summary and future work

Producer side

- The producer uses a specialized database driver
- The driver handles INSERT and COMMIT operations specially
- Other operations are executed traditionally (but some things may happen behind the scenes before the execution)
- The prototype treats prepared statements inserting scalars into memory tables specially
  - Can handle typical DW “fact tables”
  - A more general implementation could also deal with other INSERTs
- The values to insert are not inserted directly into the DW, but instead kept in a *local buffer*
- Later the data is *flushed* and reaches the memory table
- Even later, the data is *materialized* and reaches the DW tables

daisy

Example

- ***Initial state: X mem table, Y regular table***
- INSERT INTO X VALUES ($1, $2);
  INSERT INTO Y VALUES ($1, $2);
- Flush
- Materialize

Producer — X

A	B
1	1
2	2

Catalyst — X

A	B
1	1
2	2

DW — X

A	B
1	1
2	2

DW — Y

C	D
3	3

When to flush

- The default is to flush immediately after the commit
- Another possibility is to wait and temporarily place the committed data in an *archive* at the producer side ("*lazy commit*")
- A *policy* then decides when to flush the committed data
  - A policy is a Boolean method called at regular intervals
  - For example, a policy is to flush when the CPU load is below 50%
  - The user can implement his own policies
- When lazy commits are used, the producer may hold (committed!) data needed by the consumer queries
- When this happens, the catalyst sends a *request for data* and the producer flushes the needed data *on-demand*
- The producer may also execute queries on memory tables and then flush uncommitted data
  - This data is only seen by the producer, and not by the consumers, until it is committed

daisy

Avoiding duplicates

- After a materialization, rows may be available both from a DW table and a memory table
- A consumer should only see each row once
- The catalyst assigns sequential row IDs to flushed rows
- A metadata table called the *minmax table* holds the minimal and maximal row ID a new consumer should get from the catalyst
  - min = the first row in the catalyst that is **not** materialized
  - max = the last row in the catalyst that is committed
- These row IDs are read and used transparently to the user

daisy

Agenda

- Motivation
- The RiTE package
- Producer side
- ***Catalyst side***
- Consumer side
- Performance study
- Summary and future work

daisy

Catalyst side

- The catalyst allocates a big chunk of memory for each memory table
- When a table function requests data, it requests rows with IDs in the interval [i_{min} ; i_{max}] (read from the minmax table)
  - The catalyst has a *row index* from row IDs to their positions in the memory such that it is fast to locate the needed data chunk
- A table function also gives a *time handle* t telling the *commit time* of the data to receive
  - The time handle can take a special value that indicates that *all* committed data should be received
  - The catalyst also has a *time index* τ from commit times to row IDs
  - Uncommitted data has commit time ∞
- Returned rows have row IDs i such that $i_{min} \leq i \leq \min(i_{max}, \tau'(t))$ where τ' maps to a row ID

Ensure accuracy

- If lazy commits are used, the catalyst may not hold all committed data
- A consumer can request the catalyst to ensure a given accuracy by using the `ensureAccuracy(…)` methods in the specialized driver class
  - "Give me at least the data that had been committed 10 mins ago"
  - Allows higher performance if totally fresh data is not needed
- If the catalyst does not hold fresh enough data, it requests it from the producer
  - This results in an *empty update* if the producer has no new data
  - This is still useful knowledge for the catalyst and the "commit" of 0 rows is recorded for future use by the catalyst

daisy

Ensure accuracy

- When `ensureAccuracy(…)` is called, the catalyst returns a *time handle* telling the commit time for data that has **at least** the desired freshness
  - But the data may be more fresh than what was requested if the catalyst already had the data available (no problem)
  - Transparently to the user, the table function uses this time handle in future requests
- If `ensureAccuracy(…)` has not been used, the catalyst calls `ensureAccuracy(0)`
  - “Give me *all* data committed before now”

daisy

Transactions and concurrency

- The catalyst supports one producer and many concurrent consumers
  - Consumers can read existing data concurrently with new data being inserted by the producer
- The consumer driver (transparently) registers the values that exist in the minmax table when it begins a query
  - The rows with IDs in this interval will not be deleted from the memory tables while the query runs
    - But they can be *materialized* concurrently
- The consumers see committed rows exactly once
- The producer can add data and COMMIT or ROLLBACK
  - If the producer crashes, an implicit ROLLBACK is performed

daisy

Agenda

- Motivation
- The RiTE package
- Producer side
- Catalyst side
- ***Consumer side***
- Performance study
- Summary and future work

daisy

Consumer side

- The consumer uses a specialized JDBC database driver
- This driver registers (and deregisters) with the catalyst which rows from memory tables are used
  - Done transparently to the user
- Offers `ensureAccuracy(…)` methods
- Rows are fetched from the catalyst by using a PostgreSQL table function (a stored procedure returning a set of rows)
- This can be hidden by using a view:

```
CREATE VIEW v AS
   SELECT * FROM dwtable
UNION ALL
   SELECT * FROM tablefunction('dwtable',
                 (SELECT min FROM minmax),
                 (SELECT max FROM minmax));
```

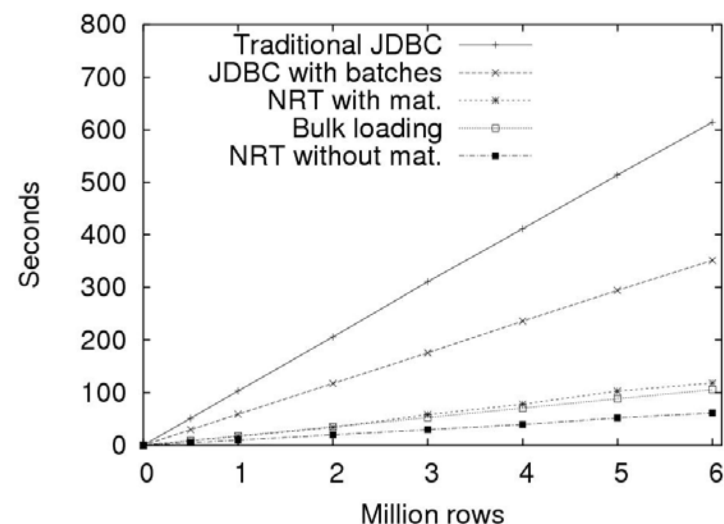
Performance study

- A prototype has been implemented:
  - Producer and consumer drivers in Java 6 (i.e., JDBC drivers)
  - Catalyst in Java 6
  - Generic C implementation of a table function for PostgreSQL 8.1
- Tested on a desktop PC with 3GHz Pentium 4 with 3.2GB RAM, and four SATA disks
- Simulates inserts into a fact table with 6 integer columns
  - Data originates from TPC-H (with a modified schema)
- Reads from memory tables a little slower than reads from (buffered) DW tables
  - 182,786 rows/second vs. 219,168 rows/second
  - Due to the many type conversions from Java types to the x86 native types that have to take place

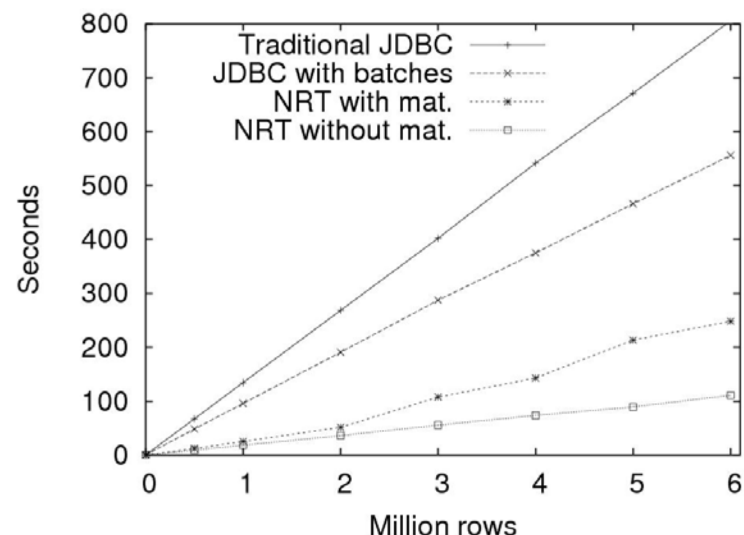
daisy

Insert performance

- Inserts only (rows/second)
  - Traditional JDBC: 9,646
  - Trad. JDBC with batches: 17,088
  - RiTE with materialization: 49,878
  - Bulk: 56,846
  - RiTE without mat.: 98,723

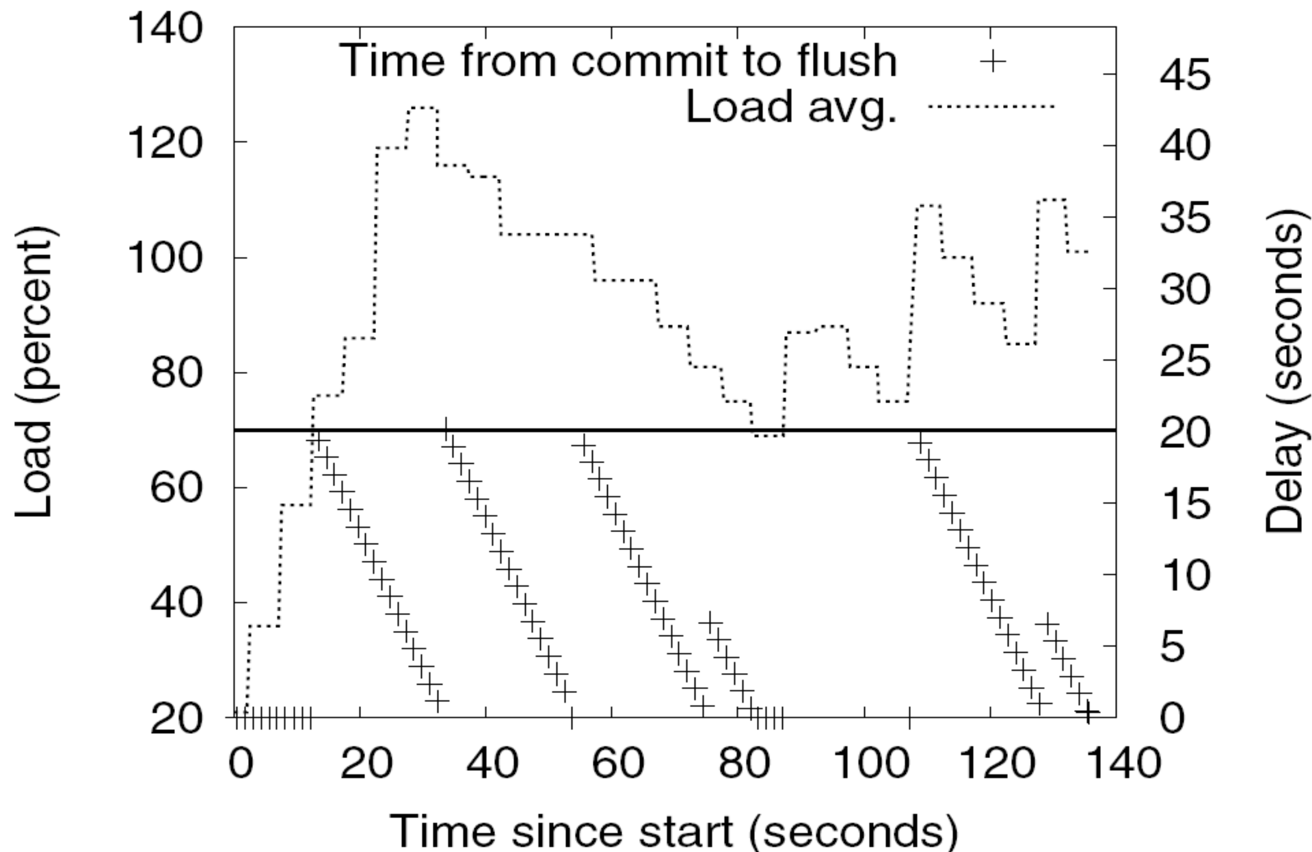


- Inserts and SELECT SUM(…) … (rows/second)
  - Traditional JDBC: 7,451
  - Trad. JDBC with batches: 10,862
  - RiTE with materialization: 22,437
  - Bulk not applicable
  - RiTE without mat.: 54,111



Lazy commit delays on a busy system

- A producer inserts rows and commits every second
- We consider a policy where a flush is done if
  - 20 seconds have passed since the last flush ***or***
  - the system's CPU load is below 70%



daisy

Summary

- A producer can insert data with bulk-load speed, but the data becomes available quickly
  - The producer still controls what units to commit together and when to commit
  - A policy can define when to flush the data, but if the data is needed by a consumer, it is made available on-demand
- Data is held by the catalyst using main memory, but RiTE can move the data to the DW tables (materialization)
- Works transparently
- As fast as bulk-load and even faster if materialization is not needed
- Pays attention to concurrency and failed transactions

daisy

Future work

- Logging
  - Could give persistency without materialization
- Support updates and deletes on memory tables
- Implement the catalyst “closer” to the DBMS
  - C/C++ implementation
  - The repeated type conversions would be avoided
- Support indexes and constraints on memory tables

daisy

Acknowledgments

- This work was in part supported by the European Internet Accessibility Observatory (EIAO) project, funded by the European Commission under Contract no. 004526

EIAO

daisy