RiTE: Providing On-Demand Data for Right-Time Data Warehousing

Christian Thomsen [#1], Torben Bach Pedersen [#2], Wolfgang Lehner [*3]

[#]*Dep. of Computer Science, Aalborg University*
Aalborg, Denmark
[1]chr@cs.aau.dk
[2]tbp@cs.aau.dk

[*]*Dep. of Computer Science, Dresden University of Technology*
Dresden, Germany
[3]wolfgang.lehner@tu-dresden.de

***Abstract*— Data warehouses (DWs) have traditionally been loaded with data at regular time intervals, e.g., monthly, weekly, or daily, using fast *bulk loading* techniques. Recently, the trend is to insert all (or only some) new source data very quickly into DWs, called *near-realtime* DWs (*right-time DWs*). This is done using regular INSERT statements, resulting in too low insert speeds. There is thus a great need for a solution that makes inserted data available quickly, while still providing bulk-load insert speeds. This paper presents *RiTE* ("Right-Time ETL"), a middleware system that provides exactly that. A data producer (ETL) can insert data that becomes available to data consumers *on demand*. RiTE includes an innovative main-memory based *catalyst* that provides fast storage and offers concurrency control. A number of policies controlling the bulk movement of data based on user requirements for persistency, availability, freshness, etc. are supported. The system works transparently to both producer and consumers. The system is integrated with an open source DBMS, and experiments show that it provides "the best of both worlds", i.e., INSERT-like data availability, but with bulk-load speeds (up to 10 times faster).**

I. Introduction

Data warehouses (DWs) [8] have traditionally been loaded with data at regular time intervals, e.g., monthly, weekly, or daily. Here, fast *bulk loading* techniques have typically been used in order to obtain sufficiently high insert speeds for the huge data volumes. In recent years, there has been an increasing demand for having very fresh data in DWs. Thus, new or updated data from the operational source systems has been inserted very quickly (within seconds or minutes) into the DWs, which are commonly referred to as "*near-realtime DWs*". A more sophisticated approach acknowledges that some data needs to be very fresh, while other data may be less fresh, and thus, based on the freshness needs, inserts data at the "right time" into the DWs, referred to as "*right-time DWs*". Bulk-loading techniques are only efficient for relatively large batches of data, and are thus not feasible for the single/few row "trickle feeds" used in the latter types of DWs. Thus, these have had to revert to classical OLTP-style inserts, using regular INSERT statements executed in small transactions. But here the unavoidable problem is that the insert speed is not high enough (often an order of magnitude lower than bulk loading).

There is thus a great need for a solution that makes inserted data available quickly, while still providing bulk-load insert speeds. A lot of work has been done on supporting *read-optimized* DWs, e.g. special multidimensional index structures, OLAP servers, etc. It is, however, equally necessary to have *write-optimized system* "before" the DW. Thus, we need a solution to asynchronously propagate data from sources to the DW (under some consistency constraints). Such a solution should strike the right batch size between the two extreme forms (bulk versus single row) and find the right time to move "micro batches" of data within the system. We note that data must be inserted at the latest *when*, but not necessarily *before*, it is needed, i.e., data should be available only on-demand. There is also a need to decouple source systems and the DW.

This paper presents *RiTE* ("Right-Time ETL"), a middleware system that provides exactly such a solution. RiTE allows a data producer to continuously insert data into a DW at bulk-load speed, but such that data *consumers* (DW clients executing queries) get access to fresh data. To do this, RiTE takes advantage of a number of special characteristics of DW systems. RiTE is thus targeted at supporting one *producer* (the ETL program) doing many INSERTs with low persistency requirements (persistency can be guaranteed if needed). RiTE includes an innovative main-memory based *catalyst* that, like a chemical catalyst, enables the insert process to be performed faster and with less effort. RiTE supports a number of policies controlling the bulk movement of data based on user requirements for persistency, availability, freshness, as well as elapsed time and CPU load. Using RiTE is transparent and requires only very few changes to producer and consumer code, in most cases only the few lines establishing database connections have to be changed.

Figure 1(A) shows a classical DW system with source systems, a producer, a DW, and consumers. The black boxes show database drivers, e.g., JDBC [7]. Figure 1(B) shows the architecture for the same system using RiTE, with the catalyst and specialized database drivers. The catalyst holds data in main memory but ensures that data is transparently available to the consumers. Data from the producer can then float to the DW either via the catalyst or directly.

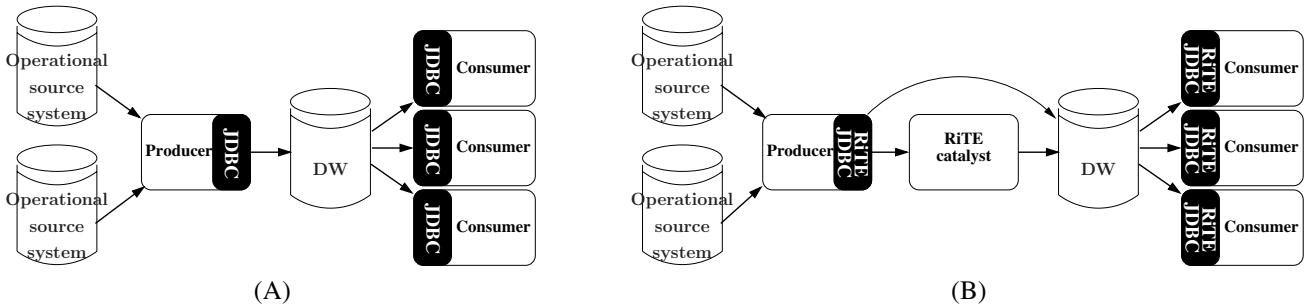


Fig. 1. Architectures for (A) a classical system and (B) a system using RiTE

Performance studies of the PostgreSQL-based prototype shows that RiTE improves insert time by up to an order of magnitude. Rows are transparently read from the RiTE catalyst with only a small overhead. Thus, RiTE provides INSERT-like data availability, but with bulk-load speeds.

The remainder of the paper is structured as follows. Section II describes RiTE from a user perspective. Section III describes the producer database driver. Section IV describes the catalyst. Section V describes the table function and the consumer database drivers. Section VI presents experimental results. Section VII presents related work and Section VIII concludes and points to future work.

II. User-Oriented Operations

We now give short, informal introductions to the operations that are treated specially by the RiTE package. These operations and other operations used internally by RiTE are all exemplified and described in details in the following sections. Note that other classical database operations that are not handled specially by RiTE can still be performed.

A. Producer Operations

The two producer operations *insert* and *commit* are handled specially by RiTE. From the user's point of view, *insert* operations work as normal inserts but are faster. Behind the scenes, RiTE temporarily keeps the inserted values locally at the producer side and later moves them towards the DW in bulk. The strategy about when to move data in bulk is based on different policies that are explained later. It is, however, done such that the data always is available from the DW when it is needed for querying.

The *commit* operation makes inserted data available for consumers. But when using RiTE, the user decides if committed data is written to the DW's tables. If this is done, the commit is called a *materialization*. If the user does not have strict persistency requirements (e.g., if the data can be re-extracted from the sources), it is also possible to commit the data without doing a materialization which then can be done later. This is faster, but still makes the data available for consumers. Such a commit can be done in different ways that affect when the bulk moving of data takes place.

B. Consumer Operations

For a consumer, there are also two operations that are handled specially: *read* and *ensure accuracy*. From the user's point of view, a *read* is done by using SELECT. Behind the scenes, transparently to the user, the read is not necessarily just a read from tables in the DW.

The only new operation introduced by RiTE is *ensure accuracy*. This is relevant for a consumer that does not necessarily need data that is as fresh as possible and thus can help the system to get a better performance. For example, it may be acceptable for a daily status report to consider all sales data that existed 10 minutes ago but not newer data. By using the ensure accuracy operation, the consumer is guaranteed that it at least sees the data that existed 10 minutes ago.

III. Producer Side

In this section, the specialized database driver for the producer is described.

A. Setup

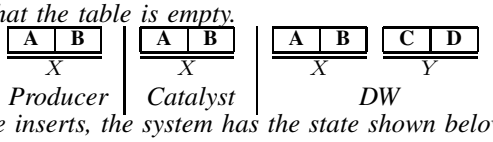
The RiTE producer driver is defined by an extension of the standard Java JDBC Connection interface. That means that to start using it from an existing Java application, only the lines where the connection to the database is made must be changed. The driver must be told which of the DW's tables the catalyst provides intermediate storage for (so-called *memory tables*). Inserts to these tables are then handled by the driver. Statements not handled specially by the RiTE driver are executed via a traditional JDBC Connection implementation.

B. Insert

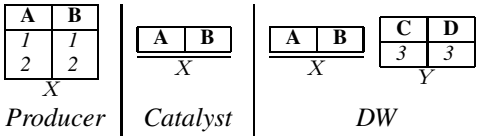
When a prepared statement is made, the driver detects if the statement inserts scalar values into a memory table. If so, the driver takes the values to insert from the statement when this is executed and stores them in a local buffer.

Example 3.1 (Insert): *Consider an example where the DW has two (empty) tables,* $X(A, B)$ *and* $Y(C, D)$*. RiTE is used such that a memory table is made for* X*. (This setup is used as a running example in the paper.) Now, assume that the producer code with prepared statements inserts the rows* $(1, 1)$ *and* $(2, 2)$ *into* X *and* $(3, 3)$ *into Y. Before these inserts, the system has the following state where the local buffer for* X *is shown to the left, the catalyst's memory table for* X *in the*

middle and the DW's tables (from now on referred to as the DW tables*) to the right. A double line in the bottom of a table shows that the table is empty.*



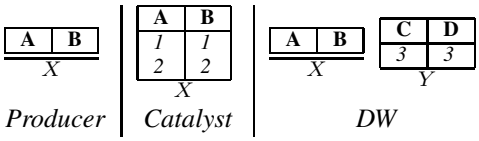
After the inserts, the system has the state shown below where the two new X rows are held in the local buffer and the new Y row is in the DW table Y.



C. Flush

The new rows from the prepared statement remain in the producer driver's buffer until a commit operation is done by the producer or optionally until the producer executes a query that should consider (uncommitted) data inserted by the producer itself. The held rows are then *flushed* to the catalyst (not the DW) in a bulk operation.

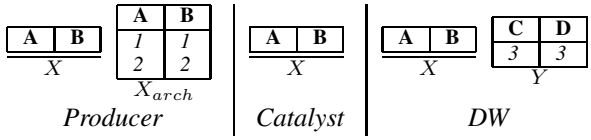
Example 3.2 (Flush): *Consider again the state obtained in Example 3.1 and assume that the producer commits the data such that a flush is initiated. This results in a state where the X rows have migrated to the catalyst.*



D. Lazy Commit

It is also possible for the producer driver to keep rows locally *after* a commit whenever a *policy* defines to do so. When committed data is not flushed immediately, we have a *lazy commit*. When a lazy commit appears, the producer driver records the *commit time* at which commit operation was invoked and places all rows in the buffer in an *archive* which holds committed, but not flushed, rows. The archive is flushed later as explained below.

Example 3.3 (Lazy commit): *Consider again the state obtained in Example 3.1. If the producer performs a lazy commit, we get the following state where X_{arch} is an archive.*



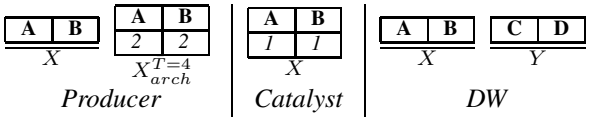
Compare this to the state obtained in Example 3.2. In the current example, the X rows are not migrating to the catalyst but remain on the producer side. After a flush is performed, the state resembles the situation of Example 3.2.

E. Requests for data

It is possible for the producer driver at the same time to have several archives with different commit times. These archives hold committed data that eventually should be flushed. At the latest, the flush is done when the connection to the DW is closed, but it may also happen before. When one or more archives exist, the producer driver sets up a background thread that listens for requests for data from the catalyst. As will be explained later, such a request occurs because a consumer has a demand for fresh data. The catalyst might ask only for parts of the archived data in which case only the requested parts are flushed. The recorded commit times are used to decide which parts to flush.

Example 3.4 (Request for data): *Consider again the running example and assume that lazy commits are used for the following sequence of events. The numbers shown to the left are (abstract) time stamps. Before the shown events, nothing has happened.*

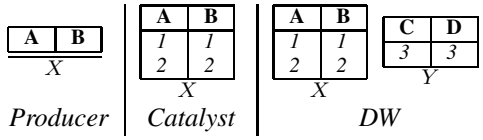
1) *The row $r = (1, 1)$ is inserted into X by the producer.*
2) *The producer commits, resulting in the archive $X_{arch}^{T=2}$ for X. This archive holds the row r.*
3) *The row $s = (2, 2)$ is inserted into X by the producer.*
4) *The producer commits. This results in that the archive $X_{arch}^{T=4}$ is made for X. This archive holds the row s.*
5) *The consumer requests the catalyst to hold data for X that is maximally 2 time units old. This means that the catalyst should at least hold the data committed at time $5 - 2 = 3$. To fulfill this, the catalyst sends a request for data to the producer. The producer then flushes the data in $X_{arch}^{T=2}$ (the only archive with data committed at time 3). Row r (committed at time 2) is then available from the catalyst, whereas row s (committed at time 4) is not. This gives the state shown below.*



F. Materialize

Data from the archives is also flushed when the producer wishes to *materialize* the rows such that they are written to the DW tables. This is done to make the rows reach their final target (the DW table), to make space for other rows in the catalyst, and to guarantee persistency. Persistency is not guaranteed when rows are stored by the catalyst. In case of a crash, the rows in the catalyst will be lost. Recall that in typical DW environments this is not a problem since the data can be reloaded from the operational systems. When rows on the other hand have been materialized, the usual persistency guarantees given by the DW DBMS apply. Note that the producer thus controls the persistency guarantee since the catalyst does not do "implicit" materializations. To make materialization possible, the RiTE producer driver extends JDBC's Connection class with the method `commit(boolean)` which performs a commit operation and where the argument decides whether the rows should be materialized to the DW tables before the commit operation is performed in the DW. To make the rows ready for materialization, the producer driver first has to transfer them to the catalyst. Note that since a materialization only happens together with a commit operation, data held in the producer driver's local buffer is flushed at the same time.

Example 3.5 (Materialization): *Assume that the state is as obtained in Example 3.2. A materialization then gives the following state where the X rows are inserted into the DW.*



Note that the rows are still present in the catalyst after the materialization. However, it is automatically ensured that a consumer only sees each row instance once (this is explained in Section V). When space is needed, the now materialized rows will eventually be deleted from the catalyst.

G. Policies

Finally, data in archives is flushed when a *policy* has defined that it is time to do so. A policy is simply a function that returns a Boolean value. When the return value of the policy is `true`, the rows are flushed and vice versa. The producer invokes the policy and checks the return value at regular user-definable intervals. By using policies, it is for example possible to make the producer less intrusive on busy systems by considering the load average. A possible policy is thus only to flush if the load average for the last minute has been below 80% or if 10 minutes have passed since the last flush.

The RiTE package includes policies 1) for flushing immediately after a commit (this is the default), 2) for waiting as long as possible, i.e, only flush on-demand, and 3) for load-aware policy-based flushing when the load average is below some percentage or a certain time interval has passed since the last flush. Further, an interface that the user can implement to define her own policies is included. The interface has two functions: One for the policy itself, i.e., a function returning a Boolean value, and one used to inform the implementation that the data has been flushed for another reason, e.g., a request for data from the catalyst.

To start using lazy commit with a given policy, the user only has to define which policy to use. Thus, it only requires one line of code to start using a policy. The rest is handled transparently by the RiTE drivers.

H. The minmax Table

When rows are flushed to the catalyst, the catalyst implicitly assigns *row IDs* to the rows and returns the maximal assigned row ID to the producer driver. The producer driver then updates a special metadata table, called the *minmax table*, in the DW. The minmax table holds data about the minimal and maximal row ID for rows that a consumer should get from the catalyst. Note that these row IDs are handled completely transparently by the RiTE software and are never seen by the producer or consumer code. So after a flush, rows with new row IDs are available and the information about the maximal available row ID is updated. As explained later, this only affects later consumer queries. Already running queries are unaffected and will not see the new rows that were committed after they started.

After a materialization, the producer driver similarly updates the information about the minimal row ID of rows that a new consumer query should get from the catalyst. The reason is that rows with lower row IDs now, after the materialization, have migrated to the tables in the DW.

Example 3.6 (The minmax table): *Consider again Example 3.2 where data was flushed. Assume that the row* $(1,1)$ *is assigned row ID 1 and the row* $(2,2)$ *is assigned row ID 2. After the flush, the minmax table has the content shown to the left below. After the materialization in Example 3.5, it has the content shown to the right.*

min	max
1	2

After Ex. 3.2

min	max
3	2

After Ex. 3.5

Note that after the materialization, the minmax table tells that consumers should get the empty set of rows from the catalyst since no row has an ID such that both $ID \leq 2$ *and* $ID \geq 3$ *hold. The consumers should now get the rows from the DW table instead.*

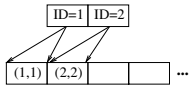
IV. Catalyst Side

We now describe the catalyst. The purpose of the catalyst is to provide fast, intermediate storage for data. It does so by storing rows in main memory. It can serve one producer driver and many consumer drivers and their table functions at the same time. Note that the consumer driver itself does not fetch rows. Instead it (transparently to the user) informs the catalyst about which rows should be readable by a table function. A table function is the remedy that makes rows accessible in the DW. The catalyst is independent of the used DBMS as its sole functions are to 1) store rows for a producer, 2) deliver them to a table function, and 3) delete them when they are marked as unused (i.e., no consumer currently uses them and they have been materialized).

A. The Row Index

The catalyst allocates a user-adjustable amount of memory for each memory table and uses this to store the memory table's rows. Whenever a producer driver adds rows, the rows are implicitly assigned row IDs by the catalyst. All row IDs are taken from the same sequence such that there are no duplicates among row IDs for different memory tables. The catalyst maintains a *row index* that is used to map between row IDs and start and end positions for the data of the rows. The row IDs are only stored in the row index, not together with the data of the rows.

Example 4.1 (The row index): *Consider again Example 3.2 and assume again that the row* $(1,1)$ *is assigned the row ID 1 and the row* $(2,2)$ *the row ID 2. Then the row index will be as shown below.*



(Note that although the row index here is shown as a list, a tree-based index is used in the implementation.)

When a table function reads data, it gives the minimal and maximal needed row IDs (recall that these were made available in the minmax table by the producer driver). By using the row index, it is then very easy for the catalyst to find the chunk of memory to transfer to the table function.

B. The Time Index

When a producer driver adds rows, it must tell the catalyst when the rows were committed at the producer side. For each memory table, the catalyst maintains a *time index* that for a commit time t maps to the row ID of the last row that was committed at time t. When a producer driver adds rows that are not yet committed (this is an option for a producer that needs to query its own uncommitted data), it gives them the special time stamp ∞.

Example 4.2 (The time index): *Consider again inserts into the memory table X in the running example and assume that the rows r_1 and r_2 are committed at time t_1 and the rows r_3 and r_4 are committed at time t_2. Assume that the row r_n gets the row ID n. The time index τ is then a partial function from time stamps to row IDs such that $\tau(t_1) = 2$ and $\tau(t_2) = 4$.*

A producer driver must transfer rows in a way where for a single memory table, all rows that were committed at time t_1 are flushed in one operation and before rows committed at time t_2 for $t_1 < t_2$. It therefore holds that when a producer driver adds uncommitted rows, it must already have added all its committed rows since they have commit times less than ∞. On the other hand, when new rows with a time stamp $t \neq \infty$ are added, all rows with the time stamp ∞ can implicitly be assumed to also be covered by this new commit and can have their time stamp updated to t. In case of a rollback, the catalyst simply has to discard all rows with the time stamp ∞. The chunk of memory that holds these rows is easy to identify by using the time and row indexes.

C. Ensuring Accuracy

A consumer can tell the catalyst to ensure that it holds the data with a certain accuracy (i.e., the data that was committed by the producer a certain time interval ago) for a subset of the memory tables. The default is that the catalyst should have all data, but with a one-line change in the consumer code, the consumer can ease the work of the producer and catalyst by only requiring data of a certain freshness.

When the catalyst receives such a wish, it sees if this can be fulfilled with the data it currently has. If the producer does not do lazy commits, this is trivially true. The catalyst knows whether the producer driver has connected to listen for requests for data. If it has not, it can be assumed that the producer driver does not do lazy commits. If the producer on the other hand uses lazy commits and the catalyst is instructed to ensure that it at least has data committed at time t for the set of memory tables M, it must ensure that it has the data or request the producer driver to flush that data. This is the case in Example 3.4 where the producer is requested to flush data committed at time 3. If the catalyst already has rows with the time stamp t' where $t \leq t' \neq \infty$ for all $m \in M$, it also has the committed data for t for $m \in M$ due to the flush order rule explained above. It can even be the case that for every $m \in M$, the catalyst has data committed at time $t' > t$. This data can also be used as the operation is meant to ensure that the catalyst's data is not older than the data that was committed at the given time stamp.

It might, however, be the case that the catalyst has no data committed at or after the wished time stamp t for (some of) the memory tables in M for which accuracy should be ensured. When this happens, the catalyst finds the tables that do not have sufficiently accurate data and requests the producer driver to transfer data for these. It might then be the case that for a memory table m no rows are held in the producer driver's archives in which case the producer driver sends an *empty update* for m, i.e. adds and commits zero rows. For the catalyst, this is still valuable information as the time index can be updated and the accuracy ensured.

Example 4.3 (Empty update): *Consider again Example 4.2 and assume that no further rows are inserted into X, but that there is a lazy commit at time t_3. If the catalyst sends a request for data committed at time t_3, the producer driver will make an empty update such that the time index maps the time stamp t_3 to the row ID 4: $\tau(t_3) = 4$. Note that we then have $\tau(t_2) = \tau(t_3)$ since no rows were added to X between the commits at t_2 and t_3.*

If the catalyst is instructed to ensure that it at least has all data committed at time t for the memory tables M, it goes through Algorithm 1 where $\mathcal{C}(m,t) = \{\tilde{t} \mid \tilde{t} \in \mathcal{T}(m) \wedge \tilde{t} \geq t\}$ and $\mathcal{T}(m)$ is the set of commit times different from ∞ in the time index for memory table m.

Algorithm 1 Find time stamp to consider

Input: A time stamp t and a set of memory tables M

1: **for** $m \in M$ **do**
2: **if** $\mathcal{C}(m,t) = \emptyset$ **then**
3: Request from the producer driver all the unflushed data for m that was committed before or at time t
4: $\Omega_m \leftarrow \{t\}$
5: **else**
6: $\Omega_m \leftarrow \mathcal{C}(m,t) \cup \{t\}$
7: **return** max $\left(\bigcap_{m \in M} \Omega_m\right)$

The return value of Algorithm 1 is the newest time stamp for which data can be considered. That means that if the algorithm is invoked for a time stamp t and a set of of memory tables M and returns $\tilde{t}$, it holds that $\tilde{t} \geq t$ and that the catalyst now holds all data that was committed at time $\tilde{t}$ for all $m \in M$. Line 2–3 of the algorithm ensure that the catalyst at least has all the (possibly empty) data sets committed at (or before) time t for each $m \in M$. So we know that data from time t can be considered. But if all $m \in M$ have newer committed data available, the algorithm picks the maximum time stamp that every m has data for. The found time stamp is returned to the consumer driver which (transparently to the user) ensures that it is used when data is read from the catalyst the next time.

D. Reads

When a table function reads data, it must also give the catalyst a time stamp that decides what data to include in the result. The time stamp is needed to ensure that data that is too new is not included in the result set as illustrated in the following example.

Example 4.4 (Problems in* not *using the time stamp): *Recall the setup for the running example but now assume that both tables X and Y have memory tables. Now consider a scenario where the producer uses lazy commit and the following events take place. (The numbers show how many minutes have passed since the system was started).*

1) *The producer inserts X and Y rows and commits.*
2) *Data for Y is flushed.*
3) *The producer inserts X and Y rows and commits.*
4) *Data for X is flushed.*
5) *A consumer wants 4 minutes accuracy for X and Y.*

The time stamp to use is then for the first commit (4 minutes ago). Note that the last flush for X was 1 minute ago (so all committed data for X is available in the catalyst) while the last flush for Y was 3 minutes ago (so only data committed 4 minutes ago is available in the catalyst). If the catalyst did not use the time stamp and naively returned all data, it would return possibly inconsistent data since X contains data committed 1 minute ago but Y does not.

So by using the time stamp, the catalyst ensures that a consistent snapshot of the committed data is used when returning data to a table function. Based on the time stamp and the time index, the last row to include is found. The last row returned is the row with the biggest row ID that is less than or equal to the minimum of the requested max row ID and the row ID found from the time stamp. Formally, if the minimal requested row ID is i_{min}, the maximal requested row ID is i_{max}, the time stamp is t, and $\hat{\tau}(t)$ is a function giving the time index mapping from the biggest time stamp smaller than or equal to t to a row ID (or -1 if this is undefined), then all returned rows have their row IDs in the set

$$\Delta = \{n \mid n \in \mathbb{N}, i_{min} \leq n \leq \min(i_{max}, \hat{\tau}(t))\}$$

Note that the number of returned rows may be different from $|\Delta|$. For a single memory table it is not given that it has all (or even any of) the rows with row IDs in Δ.

If the catalyst has not been instructed to ensure a certain accuracy, the table function will use a special time stamp that says that all committed data must be considered (i.e., the catalyst must hold data committed at or before the current time and the time stamp is set to the current time).

E. Registering Rows as Being Used

A consumer driver can *register* rows with row IDs in a given interval as being used to ensure that they are not deleted from the catalyst while a consumer query should be able to read them there. To register rows as being used, corresponds to getting a shared lock. Rows that are registered as being used cannot be deleted from the catalyst. Note that it is not enough to consider rows currently being read as used. A single consumer query may need data from different memory tables or from the same table more than once. In between two reads, the catalyst should not have deleted rows that were within the desired interval of rows in the first read. Therefore, rows should be registered as being used before the query starts and *deregistered* after it finishes (the consumer driver does this automatically and transparently as will be explained in Section V).

Only rows that are not already materialized can be registered as being used by a consumer. Already materialized rows, can be read from the DW tables and should not block the catalyst from freeing memory. Rows can, on the other hand, be materialized while they are still registered as used. When this happens, the rows will for some time be available both in the DW and in the catalyst. But due to the consumer driver's use of the minmax table, a consumer will only see one instance of each row. This is explained in Section V.

When the producer has performed a materialization, the producer driver informs the catalyst about this. The catalyst uses this to decide which rows it can delete. Rows that are materialized and not registered as being used, can safely be deleted such that the memory can be reused. Deletion is done automatically by the catalyst when more space is needed. Since materialization happens together with commit, it is the case that the rows to materialize have row IDs within a given interval. It is therefore also the case, that the catalyst only has to free one continuous block of memory for each memory table and there is no need to use maps over free regions or similar techniques.

V. Consumer Side

In this section, the consumer driver is described. Like the producer driver, the consumer driver is defined by an extension of the JDBC Connection interface. This extension adds methods for defining how accurate data read from the catalyst has to be. Further, the consumer driver (transparently to the user) ensures that rows are not deleted from the catalyst while they are needed by a consumer query.

From the consumer's point of view, the consumer driver is executing queries with the READ COMMITTED isolation level. To implement this such that it works as expected for both data in the DW and in the catalyst, the driver actually executes queries towards the DW in the REPEATABLE READ isolation level.

A. Registering Rows as Being Used

Before a query is executed, the consumer driver has to register row IDs as used. As explained in the previous section, this is done to ensure that the rows that exist in the catalyst when the query starts, continue to exist while the query is executed. The row IDs to register as used are those in the range defined by the minmax table, i.e., from the first row that is not materialized when the query starts to the last row that is committed when the query starts. To make sure that rows will not disappear from the catalyst while a query is running,

the consumer driver will whenever a method executing a query is invoked, first read values from the minmax table and try to register them with the catalyst. This might fail if a materialization is done between the time the consumer driver reads the values and the time it gives them to the catalyst (recall that the catalyst only allows row IDs of non-materialized rows to be registered as used). In that case, the consumer driver ends the transaction, starts a new transaction and reads values from the minmax table and tries to register them. To avoid starvation problems, the catalyst gives priority to consumers retrying to register values. When the values from the minmax table are registered, the query is executed. After the query is executed, the consumer driver deregisters the values.

B. Ensuring Accuracy of Read Data

The consumer driver also provides the consumer with methods that determine how old data from the catalyst is allowed to be. This is relevant when the producer uses lazy commits. A consumer can then explicitly tell the catalyst how accurate data it needs. If data of the given accuracy or newer data already exist in the catalyst, the producer and the catalyst are released from the burden of flushing data. If the catalyst, on the other hand, does not hold sufficiently fresh data, it requests the producer to flush the needed data. But this happens on-demand and only for the needed data. Note that if these methods are not used, the default is that the catalyst holds all committed data.

Concretely, the JDBC Connection interface is extended with methods `ensureAccuracy(...)` that take a time interval and memory table names as arguments. When these are invoked, the consumer driver passes the wanted accuracy to the catalyst that returns a time stamp for for which it has the committed data and that is accurate enough. The value is stored in the DW in a session variable such that it is available for the table function.

C. Reading Data with the Table Function

The consumer driver itself does not read rows from the catalyst. Instead the DW reads rows through a table function, i.e., a stored procedure that returns a set of rows with a structure like rows in a table in the DW. The table function takes as arguments the name of the memory table to read data from and the minimal and maximal row ID of rows to read. When the table function wants to read rows from the catalyst, it also gives the catalyst a time stamp that defines how fresh the data must be (as explained in Section IV). Although the row ID arguments can be used to limit the result set in other ways, the normal usage of the minimal row ID is to avoid that the catalyst returns rows that are already materialized when the query begins. This value is defined such that rows with lower row IDs have already been materialized and should be read from the DW. Only from the found value and up, the rows should be read from the catalyst. The normal usage of the maximal row ID is to avoid that the catalyst returns rows that are not committed when the query begins. It is defined to mean that rows with a greater row ID are not committed yet. If this value is read once and reused, it does not affect the query if more rows are committed later.

Example 5.1 (Use of the minmax table): *Consider again the state of the minmax table after the materialization in Example 3.6 and assume that the producer inserts and commits two rows that get the row IDs 3 and 4, respectively. In the minmax table, the min value is then 3 and the max value is 4.*

A consumer driver now reads these values from the minmax table and successfully registers them. When the table function is given these values, it reads the two new rows from the catalyst. Rows with a row ID less than 3 should not be read since they were already in the DW table when the query started. Now assume that the consumer's query is expensive and involves reading data from the memory table twice. After the first time data is read, but before the second time, the producer inserts and commits some new rows that get row IDs greater than 4. These rows did not exist when the query started. To avoid that the query sees them, the table function is still given the previously read values (i.e., $min = 3$ and $max = 4$).

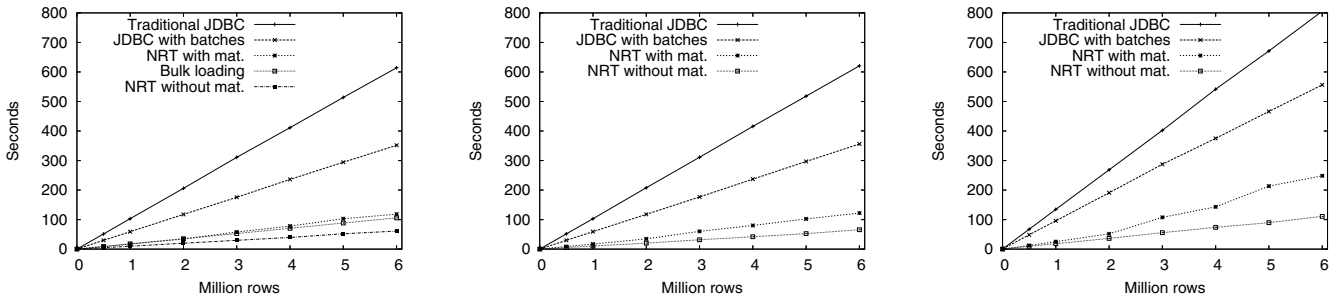
Finally, assume that while the consumer's query is executing, the producer performs a materialization such that all the new rows also become available in the DW table. The consumer query is still able to read the rows from the catalyst (since they are registered as used and thus cannot be deleted). The consumer query does not get the same rows from the DW table (since it is running in REPEATABLE READ mode and the rows were not in the DW when the query began). So the consumer sees every row that existed when the query began exactly once. The rows that were committed after the query began are not seen.

D. Transparency

To make these things transparent to the end user, a view over a DW table and its associated memory table can be defined. If the view definition uses the minmax table directly, we find the same rows in the view every time the view is used within one query (recall that the consumer connection is put in REPEATABLE READ mode). So for each DW table for which a memory table also exists, a view should be defined as

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

If the view `v` is used instead of `dwtable` in queries, the end user does not have to think about if rows are read from the tables in the DW or from the catalyst. Since the consumer driver behind the scenes is using the REAPEATABLE READ isolation level, a single query that uses the view many times sees the same values from the minmax table and thus the same set of rows in the view. But the consumer driver starts a new



(A) Commit after the last row (B) Commit per 10,000 rows (C) Concurrent producer and consumer

Fig. 2. Performance results

transaction for each query and some rows might have been updated when a query is re-executed. In other words, non-repeatable reads are possible such that the isolation level in effect is READ COMMITTED as promised by the driver.

VI. Performance Study

A. Setup

We now present a performance study of the RiTE prototype. The prototype (www.cs.aau.dk/∼chr/RiTE) consists of 1) Java JDBC database drivers for producers and consumers, 2) the catalyst (Java), and 3) a C implementation of a PostgreSQL table function. The prototype shows a working solution for a DW based on PostgreSQL [12] version 8.1 running on a Linux x86 platform. However, the applied principles are general and could be used for most DBMSs. The catalyst is completely DBMS-independent while the JDBC drivers have few (marked) PostgreSQL dependencies. The table function is, of course, highly dependent on the hosting DBMS platform. The experiments have been carried out on a 3GHz Pentium 4 PC with 3.2GB RAM and four SATA disks of which one is used for DW data, one for PostgreSQL's write-ahead logs, one for source data and one for binary executables and swap area. The PC is running Ubuntu Linux 6.10, Java 6SE, and PostgreSQL 8.1.4. The PostgreSQL configuration can be found at www.cs.aau.dk/∼chr/RiTE. We simulate a producer filling a fact table. The source data originates from TPC-H [15], with the schema modified to a star schema. Rows are inserted into the typical fact table *lineitem* with 6 integer columns (*custkey*, *datekey*, *orderkey*, *partkey*, *suppkey*, and *quantity*).

B. Long Transactions

We first consider the performance when inserting many rows into one table with insert statements. We consider a producer application, both when using RiTE and the traditional JDBC driver, and compare this to applications that load the same data set by doing multirow inserts with JDBC *batches* and bulk loading, respectively. Prepared statements are used where applicable. The values to insert are read from a text file. The producer runs in one long transaction and commits after the last insert. The same producer application is used throughout, with only the lines setting up the DW JDBC connection and doing the final commit changed. A suitably modified JDBC application is used to test JDBC batches with a batch size of 10,000 rows. Bulk loading is done by letting a modified application write the data to a comma separated file and then let the PostgreSQL server read the file directly.

The graph shown in Figure 2(A) shows the results, which are 9,646 rows/second (traditional JDBC driver), 17,088 rows/second (JDBC batches), 49,878 rows/second (RiTE with materialization), 56,846 rows/second (bulk loading), and 98,723 rows/second (RiTE without materialization). As the systems scales linearly, the speeds are based on the line slopes. The best throughput is obtained when using RiTE without materialization. The throughput is then 74% higher than for bulk loading.

C. Short Transactions

The experiment is now repeated, but with commits for every 10,000 rows. As bulk loads do not commit during the load, they are not used. The results plotted in Figure 2(B) show that JDBC's performance is not affected. For JDBC batches, the throughput drops slightly (to 16,841 rows/second). With RiTE, the producer can now insert 47,686 rows/second with materialization and 90,356 rows/second without materialization. Similar results are obtained for commits for every 100,000 rows.

D. Influence from a Consumer

The 10,000 row commit experiment is repeated, but now a consumer application simultaneously performs the query `SELECT SUM(quantity)` (reading all rows) on the *lineitem* table (which has a memory table when using RiTE). The query is re-executed right after returning its results, so the system is constantly loaded. The results plotted in Figure 2(C) show that the JDBC application can insert 7,451 rows/second whereas JDBC with batches can insert 10,862 rows/second. For RiTE, the producer can insert 22,437 rows/second with materialization and 54,111 rows/second without materialization. Thus, performance is affected, but the relative advantage of RiTE remains.

E. Read Performance

We now compare how fast data can be read from a DW table and a memory table. The data sets used in the previous

experiments are loaded once (into a memory table or a DW table, depending on what is being tested). Then all rows are read 6 times from PostgreSQL's terminal and the time usages measured. The first recorded time usage is not considered as this is used to let PostgreSQL buffer the data to make fair comparisons. The performance results are plotted in the graph shown in Figure 3. From the slopes of the lines, we

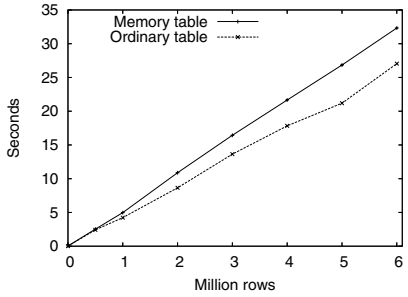


Fig. 3. Read performance results

estimate that the system reads 219,168 rows/second from a non-memory (but buffered) table whereas it reads 182,786 rows/second from a memory table. The difference is due to that when data is read from a memory table, type conversions from Java types to the host machine's native types are performed and data is transferred from the catalyst to the DBMS. There is thus a small overhead for RiTE reads.

F. Lazy Commit Delays

We now consider a producer that constantly inserts rows and commits once per second. The producer uses lazy commit and its flush policy is to flush when the system load is below 70% or 20 seconds have passed since the last flush. While the producer runs, a load simulator generates randomness in the CPU load. In the graph shown in Figure 4, the dotted line shows the CPU load (to be read relatively to the left Y axis) at different times while a cross at (x, y) shows that data committed at time x waits y seconds before it is flushed (where y should be read relatively to the right Y axis). The solid horizontal line shows where 70% is on the left Y axis and where 20 seconds is on the right Y axis, i.e., it shows the "limits" for the policy.

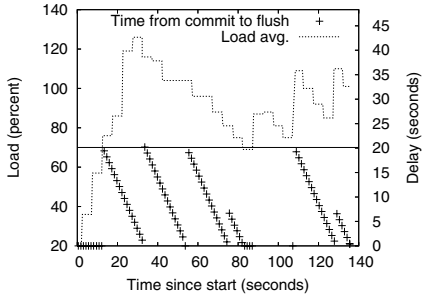


Fig. 4. Lazy commit delays and CPU load

It is seen that at first, the CPU load is below 70% and data is flushed with no delay after a commit. After appx. 12 seconds, the load gets higher than 70% and there are up to 20 second delays between commit and flush. When a flush is done, all committed data is flushed (notice how the crosses lie on lines with a negative slope). After appx. 82 seconds, the CPU load gets below 70% and data is flushed before 20 seconds have passed since the last flush (see the short line of crosses around 80 seconds). Also when the producer terminates, it flushes all data, so the delay is below 20 seconds.

G. Summary

From the experiments it is clear that RiTE provides a significant performance increase: between 4 and 10 times for inserts and 2 to 6 times for inserts with concurrent reads.

VII. Related Work

The issue of moving data from one place to another has a long tradition in both research and industry. The ETL process may be implemented in a materialized or virtual way. Linking external data sources into a target system is discussed in the context of federations. Using wrapper-like technologies [13], DW systems gain access to the underlying data. Selection and transformation routines are directly applied to the external data; the result directly goes into the DW tables. Materialization implies the physical movement of data into the target system. Techniques are ranging from import of flat files to (a)synchronous replication [11]. While replication may conceptually provide functionality somewhat similar to RiTE, current replication techniques are (unlike RiTE) limited to simple transformations and certain (cooperative) source systems, and put additional overhead on the data sources. In comparison, RiTE takes advantage of the special characteristics of right-time DWs, and provides quickly-available data at bulk-load insert speeds. This can be provided for any type of source system and any type of transformation, as these parts are handled by the ETL code. With RiTE, the producer decides when to make data available to all consumers and when to move data around (by using the commit materialize and operations, respectively).

From a conceptual point of view, incorporating external data into a single DW database requires a consistent global view. Starting with database snapshots [1], significant research was devoted to that problem in recent years under the notion of materialized views [6]. Initial work like [16], [2] investigated methods to establish a consistent view over multiple sources or updating multiple views with data coming from a single source [4]. All these mechanism are orthogonal to RiTE and may be applied on top of our middleware. More closely related is research documented in [14] rolling global DW states forward to certain points in time. However, this approach requires an explicit trigger while our approach is fully demand-driven. A similar approach with implicit instructions based on the notion of policies is outlined in [5]; in contrast, we focus on the efficient implementation (catalyst) in combination with

a transactionally consistent view on the data source and thus go much further.

The state of the art of continuous loading is summarized in [9]. Compared to that, RiTE gives the producer full control over the units of work to commit together and is flexible with respect to persistency guarantees versus load speed. Further, RiTE is more flexible with respect to freshness of data and offers lazy commit which can make data available in the DW on demand.

The MySQL [10] DBMS offers a memory storage engine for fast, but non-persistent, storage and access. Unlike MySQL, RiTE has functionality for migrating rows from memory to the database (i.e., materialization). The MySQL main memory storage engine also obviously does not scale to DW data volumes. Additionally, RiTE allows rows to be added while other rows are read, whereas MySQL uses table locking when rows are inserted into a memory table. MySQL also offers INSERT DELAYED syntax where many inserts can be bundled and written in one block when the target table is not in use. This holds back INSERT data similarly to RiTE, but in RiTE the producer controls when to make the rows available (at commit time). INSERT DELAYED is slower than normal INSERT if the target is not in use and should be used carefully. RiTE provides a speed-up for the producer also when no consumers exist.

VIII. Conclusion and Future Work

Motivated by the need for a solution that makes inserted data available quickly, while still providing bulk-load insert speeds, this paper presented the middleware RiTE ("Right-Time ETL"). A data producer (ETL) can insert data that becomes available to data consumers on demand. To make this possible, RiTE introduces an innovative main-memory based catalyst and supports a number of policies that control the bulk movement of data based on user requirements for persistency, availability, freshness, etc. RiTE works completely transparently to both producer and consumers. A prototype has been integrated with an open-source DBMS, and experiments have shown that RiTE provides "the best of both worlds", i.e., INSERT-like data availability, but with bulk-load speeds (up to 10 times faster).

There are many interesting directions for future work. Logging could be added to the catalyst such that persistency guarantees can also be given without materialization. Possibilities for letting rules provide transparent updating and deletion of rows inserted into memory tables are also relevant. Fast inserts could then be performed on the fly and a data cleansing procedure could correct mistakes or delete bad rows before materialization. The catalyst could also be implemented as a module in the underlying DBMS since an even better performance could be obtained if no repetitive type conversions from Java types to the DBMS' native types would have to take place. A related task is to allow indexes and constraints to be declared on memory tables.

Acknowledgments

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

References

[1] M.E. Adiba and B.G. Lindsay: "Database Snapshots". In *Proc. of VLDB'80* pp. 86-91.

[2] L.S. Colby, A. Kawaguchi, D.F. Lieuwen, I.S. Mumick, and K.A. Ross: "Supporting Multiple View Maintenance Policies". In *Proc. of SIGMOD'97* pp. 405-416.

[3] H. Engström, S. Chakravarthy, and B. Lings: "A Heuristic for Refresh Policy Selection in Heterogeneous Environments". In *Proc. of ICDE'03* pp. 674-676.

[4] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng: "Optimizing Refresh of a Set of Materialized Views". In *Proc. of VLDB'05* pp. 1043-1054.

[5] H. Guo, P.-Å. Larson, and R. Ramakrishnan: "Caching with 'Good Enough' Currency, Consistency, and Completeness". In *Proc. of VLDB'05* pp. 457-468.

[6] A. Gupta and I.S. Mumick: "Maintenance of Materialized Views: Problems, Techniques, and Applications". *IEEE Data Eng. Bull. 18(2): 3-18 (1995).*

[7] `java.sun.com/javase/technologies/database`. Last accessed Nov. 19 2007.

[8] R. Kimball and M. Ross: *The Data Warehouse Toolkit*, 2nd ed., Willey 2002.

[9] G. Luo, J.F. Naughton, C.J. Ellmann, and M.W. Waltzke: "Transaction Reordering and Grouping for Continuous Data Loading". In *Proc. of BIRTE'06* pp. 34–49.

[10] `mysql.com`. Last accessed Nov. 19 2007.

[11] T.M. Özsu and P. Valduriez: *Principles of Distributed Database Systems* 2nd ed., Prentice Hall, 1999.

[12] `postgresql.org`. Last accessed Nov. 19 2007.

[13] M.T. Roth and P.M. Schwarz: "Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources". In *Proc. of VLDB'97* pp. 266-275.

[14] K. Salem, K.S. Beyer, R. Cochrane, and B.G. Lindsay: "How To Roll a Join: Asynchronous Incremental View Maintenance". In *Proc. of SIGMOD'00* pp. 129-140.

[15] `tpc.org/tpch/`. Last accessed Nov. 19 2007.

[16] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener: "The Strobe Algorithms for Multi-Source Warehouse Consistency". In *Proc. of PDIS'96* pp. 146-157.