Abstract - Recently, there is an increasing number of applications that require support for real-time processing of rapidly arriving data streams. The high volume and fast arrival rate make it infeasible to install every data item of a stream into the database. In this study, we assume transactions running over data streams execute periodically to produce new results as temporal data to be installed into the database. We call these transactions continuous transactions. They resemble continuous queries in data stream management systems plus reading non-temporal data for creating and writing temporal data. Continuous transactions are continuous, standing and persistent with pre-declared data access pattern. Conventional real-time concurrency control protocols are not designed for systems with both continuous transactions and user transactions without prior knowledge of data access pattern. We need to ensure that user transactions can read fresh data and commit before their deadlines. On the other hand, we need to issue sufficient privilege for continuous transactions to keep temporal data fresh without violating database consistency and significantly affecting the performance of user transactions. Two new time-cognizant forced wait protocols are proposed to handle the interplay between these two types of transactions with different characteristics. These protocols are evaluated through a series of simulation experiments. We have demonstrated that taking advantage of continuous transaction and temporal data semantics in transaction scheduling can significantly improve the performance of user transactions without jeopardizing the freshness of temporal data maintained by continuous transactions.

I. INTRODUCTION

In recent years pervasive computing, refers to a scenario where a large number of sensors are distributed in our surrounding environment, is becoming more popular. These sensors collect real-time information, continuously and dynamically upstream them to a centralized database for various purposes. However, due to the relatively high-speed of incoming data streams, it is not worthwhile (if not impossible) to install every sensor data into the database. We can find numerous applications of such sensor system in our everyday life: network traffic management, financial analysis and wild life habitats monitoring. They are also known as data intensive applications. Most data intensive applications involve both transactions and data with time constraints. On the one hand, user transactions need to be executed in time to meet the transaction’s deadline. On the other hand, it is necessary to refresh the database on time to ensure that it is reflecting the real world status. By preferring user requests to sensor updates, the deadline miss ratio is improved. However, the freshness might be reduced. Alternatively, the freshness increases if updates receive a higher priority. A good scheduling algorithm is needed to balance between these two requirements. Several algorithms [4] such as update first, transaction first have been proposed to handle this problem. These algorithms consider the update rate not so rapid such that installing every update is still fine. However, due to the changing of application domain, installing every update in data intensive applications is no more practical. Making the database to reflect the real world status tightly is difficult and sometimes not necessary for some data intensive applications. To reduce the update burden, “aggregated value” of sensor data can be installed instead. This aggregated value represents a snapshot view of the sensor data at a specified interval. A continuous transaction which is similar to continuous query in Data Stream Management System (DSMS) [1] can be used to perform the aggregation and return an aggregated value as a temporal data to be installed into the database. These kinds of applications usually do not require keeping raw sensor data. Instead, continuous transactions can be used to process the raw sensor data to derive more meaningful values to be installed into the database for user transactions to access. Traditional database model does not fit these applications that consist of these two types of transactions with different characteristics.

II. RELATED WORKS

In the past decades, lots of papers have been published on different issues in real-time database systems (RTDBS). Considerable research effort has been devoted to designing effective protocols for scheduling or maintaining temporal consistency of real-time transactions. In RTDBS, data are modeled as persistent relations while transactions are submitted by users from time to time. The system does not have any advance information about (1) when a transaction will be submitted and (2) what data items will be accessed. Transactions are admitted into the system without control. Data conflicts are resolved during execution, which causes unpredictability.

In view of the increasing demand for real-time data

---

*The work described in this paper was substantially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 1204/03E]
services, Kang, Son, and Stankovic [8] proposed a QoS management architecture for RTDBS in the presence of unpredictable workloads and access patterns. They pointed out that transaction timeliness and data freshness requirements are both important but they could conflict with each other. Xiong et al. [9] introduced the concept of data-deadline for maintaining the temporal consistency. In their study, temporal consistency is defined in terms of the validity interval of a version of temporal data object, after which the data version is regarded as invalid. Based on the assumption that estimates of remaining transaction execution time and response time is available, two sophisticated forced wait policies are proposed. Simulation results showed that their proposed policies improve performance. The main focus of their work is on transaction scheduling in traditional real-time databases which is not suitable for data streams applications.

Besides the conventional RTDBS, in these few years, there is another new research area that concerns about the DSMS, data stream management system, which aims to resolve the problem of high speed update stream. One of their approaches is - instead of writing each of the sensor data individually, a temporal aggregated average of these sensor data is written into the database[5]. Using this approach, it can save resources from writing hundreds of individual incoming sensor data into a database and at the same time, it maintains a certain freshness level. However, the focus of these researches is mainly on stream processing, most of them do not consider the effect of other co-existing user transactions in the system. In this paper, we will extend the ideas in these papers to develop a new protocol that aims to reduce the data conflict and thus the restart rate.

III. DATABASE AND TRANSACTION MODEL

We consider our database a main memory database (MMDB). MMDB [3] is commonly used in real time applications because of its relatively high processing speed and decreasing cost in memory. Our database consists of a collection of data, partitioned in three sets, namely sensor data, temporal data and general data.

Sensor data represent real time continuous changing information received from data streams. Each sensor data item is associated with a timestamp showing the time of the real-world status represented by the data item. In data stream applications, the arrival rate of data streams is always rapid and unpredictable. Installing all the updates may incur great workload that exceeds the computational capacity of the system. Different techniques have been proposed in DSMS for handling these high input data rates. One of the most commonly used approaches is to cache a small window of most recent sensor data in a temporary working storage (buffer) and apply various types of non-blocking query operators to extract useful aggregated results [1] while processed tuples will be discarded. To extend this idea of continuous query further, continuous transactions can be used to further process and write these aggregate values into the database as temporal data.

Temporal data are database objects whose status will become invalid with the passage of time. Each of these temporal data is associated with a time interval indicating when this data item remains valid. To maintain the freshness of temporal data, a continuous transaction is associated with each temporal data item. In each period as defined by the window size, the continuous transaction performs aggregation operation on sensor data within the window and accordingly updates the value of the associated temporal data item. This period is also the validity interval of the temporal data item.

General data are data objects found in conventional databases whose status is independent with time. General data are either static data or data created by user transactions sporadically. General data can be read and written by user transactions. Continuous transactions may also read general data for computing the values of temporal data.

There are two classes of transactions in the system: continuous transactions (CT) and one-time transactions (OT). CT forms the "base load" of the system which will be invoked at fixed time intervals. OT is the "transient load" generated in response to user’s requests.

One Time Transaction (OT) has much in common with transactions in traditional RTDBS. An OT consists of a sequence of read and write operations. It reads temporal data or general data and writes general data. OT needs to commit before its deadline. Users submit OT to the system on demand. The execution time and data access pattern of OT are not known in advance. OT acquires resources dynamically which may lead to blocking, aborts and restarts in case of data conflict. Since OT reads temporal data, thus in addition to transaction deadline, OT also needs to meet the data deadline. The data deadline of an OT is defined as the earliest expiry time of those temporal data items accessed by the OT. In other words, an OT must commit on or before its own deadline as well as within the validity intervals of the temporal data it has accessed. Thus the effective deadline of an OT is the minimum of its transaction deadline and data deadline.

Continuous Transaction (CT) is a new class of transactions in data stream applications. CT has two distinct features that are different from traditional database transactions. First, CT is a set of standing transactions that are evaluated continuously and periodically. Second, CT contains pre-declared information about data accesses. Since CT operates on real time data, thus similar to other real time transactions, CT has to fulfill time constraint requirement. CT must complete by the release time of the next invocation. If CT fails to complete by its deadline, the freshness of the temporal data item will be compromised. The second distinction between
traditional transaction and continuous transaction is
the pre-declared data access pattern. That is, the read-
write sets for CT can be established a priori. Thus, CT
can be scheduled in a conflict-avoiding manner by
pre-acquiring all necessary resources. Pre-declaration
protocol helps to save resources wasted in transaction
abort and restart. Conventional real-time concurrency
control protocols are not designed to handle the
existence of these two types of transactions with
different characteristics. In the next section, we will
introduce new concurrency control protocols for data
intensive applications.

IV. CONCURRENCY CONTROL PROTOCOLS

In this section, we first briefly review the traditional
concurrency control protocol HP-2PL, which is
commonly used in real-time database systems and will
be used as a baseline algorithm for comparison. Then,
we propose three other concurrency control protocols
that are adapted from HP-2PL to improve the
performance of transaction processing in monitoring
applications.

A. HP-2PL

In lock based concurrency control protocols,
transactions need to obtain a lock before it can access
data item. When a transaction (T_x) wants to access a
particular data item that is already locked by another
transaction (T_y), we need to decide which transaction
can get the lock. HP-2PL resolves data conflicts in
favor of transactions with higher priority. The favored
transaction regarded as the winner of data conflict, is
allowed to lock the requested data object.

B. Delayed Restart with Temporal Consistency
Check (DR-TC)

In traditional database applications, data access
pattern of transactions is usually assumed unknown
in prior. A transaction has to process an operation
before it knows the next data object to be accessed.
During execution of a transaction, if it conflicts with
other active transactions in the system; one of the
conflicting sides must be aborted. Transaction restart
is notorious for its deterioration in the system
performance. Thus, the objective of the Delayed
Restart mechanism [7] is to reduce the number of
transaction restarts. This Delayed Restart mechanism
is applied on aborted transactions and is invoked
when the aborted transaction is about to restart. By
making use of data access pattern and execution time
recorded in the last run before being aborted, we can
avoid repeating the same data conflicts and therefore
reduce the possibility of being aborted and restarted
again. In DR-TC, we will also perform temporal
consistency check in every operation. In this way,
we can identify stale transactions and abort them
earlier. This saves resources by not executing useless
operations. In Fig. 1, the one-time transaction, T
starts at t_0, reads temporal data item x at t_1 and y at t_3
and is going to read z at t_5. In every operation, T will
check if any temporal data it has accessed got stale.
In this example, x expires at t_4. Hence, T will be
aborted and restarted. Before T actually restarts,
delayed restart mechanism is triggered to determine

(1) the possibility of further data conflict and (2) if it
has enough time to re-execute. In this example (Fig.
1), the remaining time (i.e. deadline - current time) is
not enough for T to re-execute all the previous
operations. In this case, T should be terminated
immediately to save resources because it has no
chance to commit before its deadline. In order to
predict possible data conflicts for making decision on
delaying restart, we need to have answers to the
following two questions. (1) Which data items will a
restared transaction access in its restart run? (2) How
long will it take for a restarted transaction to repeat its
previous execution?

Changing of data access path between first run and
restart run for a transaction is rare. This property is
known as access invariance) [6]. When a transaction is
aborted, it may have performed several operations.
Data access information up to the point of abort can
be captured by tracking the data access path. Since the
time lap between the aborted execution and the restart
run is relatively short. Therefore, it is reasonable to
assume that the transaction will access the same set of
data items during its restart run. If we can ensure the
availability and freshness of those temporal data items
to be accessed by the restarted transaction, we can
lower the possibility of aborting the restarted
transaction again. This is possible because we have
information about the data access pattern of CT and
thus any conflict between the restarted transaction and
CT can be avoided. A transaction should be delayed
to restart if any temporal data item to be accessed will
expire soon.

In addition to determining which data items will be
accessed again, estimating the time needed for a
restared transaction to re-execute some or all of its
operations is equally important. This is because we
hope to provide some “assurance” to restarted
transactions that they can get all the data locks that
have been accessed in its previous execution and there
will be enough time for it to re-execute at least up to
the point before being restarted in its previous
execution. However, due to the unpredictability,
estimation of transaction execution time is infeasible
in traditional RTDBS. This is especially true when the
transaction is executed for the first time. However for
a restarted transaction, it is possible to estimate the re-
exection time needed by tracking its previous
execution path. As our system do not assume any

![Fig. 1 Transaction execution under DR-TC](image-url)
prior knowledge of data access pattern of one time transactions, thus the restart time estimation is solely based on the operations already performed by the aborted transaction in its previous execution. We assume that the system loading remains more or less the same in a transaction’s restarted run since its previous execution. Therefore, the time needed for the restarted transaction to repeat the previous operations should be similar to the amount of time elapsed in its aborted run. Thus the execution time for a restarted transaction can be estimated in the following way: Re-execution Time = Abort Time – Start Time.

C. Forced Wait with Priority Abort – FW-PA

In the previous protocols, the assignment of transaction priority is based on transaction deadline. Data deadline is not taken into consideration. Dynamically increasing the transaction priority based on the tightness of effective deadline can help to maintain temporal consistency. Effective deadline of a transaction is the minimum value among transaction deadline and deadlines (expiry times) of all its accessed data.

Fig 2 shows an example of transaction execution under Forced Wait with Priority Abort (FW-PA). The one time transaction T will check if the temporal data item going to access (i.e. z) will expire before T can possibly commit at t8. If z expires before t8, T may be blocked until z is updated or otherwise T will read stale data and be aborted. However, as we mentioned above, estimating the commit time of a one time transaction is difficult for data intensive applications. Thus our FW-PA algorithm uses an alternative way to estimate the commit time of T: current time + \alpha \times slack time of T. If there exists any data item that has been accessed or going to be accessed will be updated within \alpha \times slack time (we called this checked period afterward), then T will be blocked if the following two conditions are fulfilled. 1. Priority (T) < Priority (CTz) 2. Expiry Time (z) + Estimated Execution Time (CTz) + Remaining Execution Time (T) <= Deadline (T), where CTz is the CT for updating z. Otherwise, T will be aborted. Condition 1 prevents a high priority OT to be blocked by a low priority CT. This avoids priority inversion and deadlock formation. Condition 2 aims to prevent unnecessary blocking. Transactions fail Condition 2 will not have enough time to execute after blocking.

D. Forced Wait with Priority Inheritance – FW-PI

In FW-PA, in order to prevent a low priority transaction wait a high priority transaction (i.e. priority inversion), T will be blocked if its priority is lower than the conflicting CT. In most time, the deadline of CT, which is equal to its period, is later than that of OT. This makes the priority of CT lower than that of OT. Due to the relatively low priority of CT, CT may be difficult to commit even if conflicting OT is aborted. This introduces a lose-lose situation. On the one hand, the conflicting OT kills itself and releases the lock for CT to execute and on the other hand, the relatively low priority CT also fails to commit. FW-PI relieves the problem by having CT inherit a higher priority from the conflicting OT. In this way, the CT can get a higher priority and execute in a shorter time. The conflicting OT can also resume earlier after the CT is committed.

V. SIMULATION MODEL

A detail simulation model of our system is shown in Fig. 3. The transaction manager (TM) generates and handles different classes of transactions, namely continuous transactions (CT) and one-time transactions (OT). CTs are registered in the continuous transaction store and placed together with OTs into the transaction queue for scheduling.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart_delay=adaptive</td>
<td>The delay time before a transaction restart</td>
</tr>
<tr>
<td>Numof CPU=1</td>
<td>Number of CPU</td>
</tr>
<tr>
<td>CT_load= 60%</td>
<td>CT loading</td>
</tr>
<tr>
<td>cpu_time= 0.015 s</td>
<td>CPU Processing Time</td>
</tr>
<tr>
<td>memory_time=0.001s</td>
<td>Memory Accessing Time</td>
</tr>
<tr>
<td>update_rate=0.0001s</td>
<td>Stream Update Rate</td>
</tr>
<tr>
<td>buffer_time=0.000025s</td>
<td>Buffer Services Time</td>
</tr>
<tr>
<td>sensor_data = 200</td>
<td># of Sensor Objects</td>
</tr>
<tr>
<td>temporal_data= 300</td>
<td># of Temporal Objects</td>
</tr>
<tr>
<td>general_data=500</td>
<td># of General Objects</td>
</tr>
<tr>
<td>avg_txn_size = 10</td>
<td>Average Transaction Length for both OT and CT</td>
</tr>
<tr>
<td>%of_Sensor_Operation = 30</td>
<td>% of operations accessing sensor data for CT</td>
</tr>
<tr>
<td>%of_temporal_Operation = 30% (50%)</td>
<td>% of operations accessing temporal data for CT (OT)</td>
</tr>
<tr>
<td>%_of_General_Operation = 40% (50%)</td>
<td>% of operations accessing general data for CT (OT)</td>
</tr>
<tr>
<td>write_probability= 0.9</td>
<td>Write Probability</td>
</tr>
<tr>
<td>lsf = 2.0 , hsf = 8.0</td>
<td>Slack Factor Range</td>
</tr>
</tbody>
</table>

Table I gives values of the simulation parameters that are used in our experiments. Our simulation model consists of one CPU and the entire database is
kept in main memory. Transactions are queued in the order of priority while the priority of transactions is based on their deadline (Earliest Deadline First). CT is fired according to its own period while OT is submitted to system “sporadically”. Each OT and CT is associated with a firm deadline. Transactions that miss their deadlines will be discarded as they produce no or little value. The deadline of a one time transaction $T$ is set using the following formula: 
$$d(T) = a(T) + U(lsf, hsf) * \text{avgtxn size} * (\text{cpu time} + \text{memory time})$$
where $a(T)$ is the arrival time of $T$ and $U(lsf, hsf)$ denotes a uniformly distributed value in the range $[lsf, hsf]$. The deadline of a CT is equal to the end of its declared period. The restart delay parameter, is used to determine the blocking time before a transaction should restart. This approach was applied over all tested protocols except the base line (HP-2PL). The way used in estimating the delay time has been explained before. For HP-2PL, the restart delay is set to zero. The parameter CT_load is used to determine the system loading contributed by pre-delay. The parameter CT_load is used to determine the system loading before a transaction should restart. This approach was applied over all tested protocols except the base line (HP-2PL). The way used in estimating the delay time has been explained before. For HP-2PL, the restart delay is set to zero. The parameter CT_load is used to determine the system loading contributed by pre-deregistered CTS in the system. In our simulation, the loading of CTS is fixed throughout the experiment and is defined as: 
$$\text{CT load} = \sum_{i} C_i / T_i$$
where $C_i$ and $T_i$ is the estimated execution time and the period of continuous transaction $i$, respectively.

In choosing parameter values for our simulation experiments, we want to create a situation with high data contention. Our experiments are conducted with an average transaction size of ten and a database size of 500 data objects only. Due to the small database size and relatively long transactions, a large number of data conflicts can be created in these experiments. Also, our experiments are conducted in a resource-limited environment (with only one CPU). We choose such environment because most applications are with lots of data conflict in real world situation and resources is insufficient.

VI. RESULTS

In this section, we will have an evaluation on the performance of the four studied algorithms, namely: (1) HP-2PL (2) Delayed Restart with Temporal Consistency check, (DR-TC) (3) Forced Wait with Priority Abort (FW-PA) and (4) Forced Wait with Priority Inheritance (FW-PI). We have evaluated the algorithms with different values of $\alpha$ and found that $\alpha = 0.3$ can best demonstrate the performance difference of the algorithms. In fact, $\alpha$ can be a tunable system parameter to be determined by system loading.

A. Performance of Continuous Transactions (CT)

Fig 4 shows the miss rate of continuous transactions (CT) when $\alpha = 0.3$. Among the four protocols, HP-2PL performs the worst. DR-TC has better performance than HP-2PL due to the resources saved by the – (1) Restart check policies and (2) temporal consistency checks employed in DR-TC. Before a transaction restarts, DR-TC checks the availability of the “required locks” and determines if it has enough time to re-execute all its previously aborted operations. This lowers the chance of a restarted transaction to be aborted again due to same data conflicts. Another reason for performance improvement in DR-TC is that a transaction will check the freshness of all its accessed data in every operation while HP-2PL performs this checking only at the end of a transaction execution. If any accessed data becomes stale, the transaction will be aborted immediately and resources can be saved from completing this useless transaction. FW-PA and FW-PI have even better performance compared with DR-TC. This is because both FW-PA and FW-PI favor CT and let CT update temporal data first such that OT can access fresh data. With FW-PA, when a temporal data item to be accessed by an OT expired before it can possibly commit, the OT will either be (1) blocked (if $P(CT) > P(OT)$) and waits for the new version of data item, or (2) aborted (if $P(OT) > P(CT)$) and releases all the locks for CT. CT is given an opportunity to execute no matter its priority is higher or lower than that of OT. This increases the possibility for CT to commit and thus reduces the miss rate of CT. However, in FW-PA, the relatively low priority of CT may make it difficult to commit even all conflicting OTs have been aborted and release the locks, leading to a lose-lose situation. FW-PI relieves the problem by having CT inherits a higher priority from the blocked OT. CT can be executed faster and commits earlier with a higher priority.

B. Performance of One Time Transactions (OT)

Fig 5 shows the miss rate of one time transactions (OT) when $\alpha = 0.3$. Similar to the performance of CT, HP-2PL has the worst performance. DR-TC has a better performance than HP-2PL due to (1) early abort the ‘already stale’ transactions and (2) feasible check before transaction actually restart. These two policies save resources from doing useless operations. With DR-TC, it will not do any forecasting about whether the data going to access will expire within the remaining execution time of the transaction. In other words, a transaction that accesses valid data may not be able to commit eventually because the validity interval of data object it reads expires before it can commit. With FW-PA, before an OT accesses the data item, it will predict whether that data item will expire soon. If yes, OT will then decide whether to wait for the coming update or abort. If it finds that it will not have enough time to commit after waiting, OT will abort immediately. On the other hand, if time is enough, the priority of that OT and the conflicting CT will be compared. In order to prevent the lower priority transaction blocked by a higher priority transaction (priority inversion), OT will not block if $P(OT) > P(CT)$, instead it will abort immediately. On the contrary, if $P(CT) > P(OT)$, OT will be blocked and resume until the data item is updated by the corresponding CT. This blocking approach saves the
resources from aborting and restarting the OT. Resources can be devoted to other transactions, thus lowering the miss rate of OT. In other word, if OT is going to read a data item that will expire soon, there are two possible outcomes: 1. Abort – If it is impossible for OT to commit after blocking or P(OT) > P(CT) 2. Block – If it is possible for OT to commit after blocking AND P(CT) > P(OT), OT will be blocked and resume after the data item is updated. With FW-PA if current version of the data item going to access expires before OT can commit, OT will decide either to wait for the coming data version OR to abort itself if (a) it is infeasible to commit after the wait or (b) priority inversion occur. In the latter abort case, its intention is to make OT releases all locks it hold and let the low priority CT to update the corresponding data item first. However this intention is not always sound, due to the relatively low priority of CT, this CT may still be blocked by other high priority transactions. The relatively low priority of CT lengthens the execution time of the CT and thus updates time of the data items which the OT is waiting for. With FW-PI, CT inherit the priority of OT plus a small value (α) which makes CT to be executed faster and in a higher priority. In this way, not only CT can update the data item earlier but OT needs not to be aborted as well. This is because there is no priority inversion now (Priority of CT is slightly higher than the waiting OT after the inheritance). This helps OT to access the updated data earlier and avoids OT to be aborted which in turn improve the miss rate of OT.

VII. CONCLUSION

We observed that one main bottleneck to the performance of traditional concurrency control protocols in real-time database systems is the ineffective use of resources on restarted transactions that ultimately miss their deadlines. In this work, we extend a traditional algorithm, HP-2PL to another three algorithms called DR-TC, FW–PA and FW–PI. In DR-TC, aborted transactions with forecast able data conflict will be blocked until the conflicting transaction is committed. In FW–PA, and FW–PI, transactions will be blocked / aborted if it is found the data item going to access is to be expire soon. A simulation model is built to evaluate the performance of DR-TC, FW–PA and FW–PI as compared to HP–2PL. The results show that all proposed protocols can successfully salvage resources by delay executing those transactions with forecast able data conflict item. Benefit from these salvaged resources, more transactions can meet their deadlines.

REFERENCES


Fig. 4 CT Miss Rate VS OT Arrival Rate

Fig. 5 OT Miss Rate VS OT Arrival Rate

CT Miss Rate VS OT Arrival Rate

OT Miss Rate VS OT Arrival Rate

CT Miss Rate (α = 0.3)

OT Miss Rate (α = 0.3)