Handling Sensor Data in Rapidly Changing Environments to Support Soft Real-Time Requirements

Anindya Datta, Igor R. Viguier
University of Arizona
Tucson, AZ 85721
{adatta,igor}@loochi.bpa.arizona.edu

May 6, 1996

Abstract

The objective of this paper is to perform an in-depth study on the problem of handling sensor data in rapidly changing environments. Due to the dynamic nature of this data, as well as specific reporting patterns, the transactions that manipulate this data have particular requirements. Specifically, these requirements are *recency* and *temporal consistency.* We study strategies to satisfy these requirements, while maintaining predetermined levels of *Quality of Service* (QOS) in the system. We identify a number of algorithms and perform a detailed performance study. This study enables us to make certain general recommendations regarding the properties of algorithms designed to handle rapidly changing sensor data.

Keywords: Sensor Data, Recency, Temporal Consistency, Soft Real-Time, QOS, Performance Evaluation

1 Introduction

Database systems that attempt to model and control external environments have attracted the attention of researchers in recent times. The application domains for such databases are numerous – network management, manufacturing process control, air traffic control, intelligent highway systems to name a few [4, 35, 39]. The general system model that has been proposed for such systems include the following features:

- The database is the repository of all system information that needs to be accessed or manipulated;
- Monitoring tools (commonly known as sensors) are distributed throughout the real system being modeled. These sensors monitor the state of the system and report to the database. Such state reports arrive at the database at a high frequency

We call such systems ARCS (Active, Rapidly Changing data Systems). The motivation for designing database management systems (DBMSs) for ARCS arises from the fact that ARCS scenarios tend to be extremely data intensive (e.g., a typical network management center handles several gigabytes of data per day [8]). Also, the control decisions in ARCS scenarios are typically data driven, e.g., a control decision in network management may be to compute retransmission rates at specific nodes in response to existing and past state information. In general it is often necessary to access and manipulate current as well as historical data in order to make control decisions. System operators submit queries to ARCS databases regarding current or past state information. Such queries typically access sensor reported information and have certain special requirements. Briefly, these requirements are:

- There are usually soft-real time requirements on these queries often specified as *Quality of Service* (QOS) measures. Such QOS requirements place desired bounds on the response times of these queries.
- These queries typically ask for data values that were valid at particular times. Thus the returned values need to have occurred at or around that time. We refer to this requirement as *recency*
- These queries need data values that are *temporally consistent*, i.e., these values need to have been valid around the same time

Thus ARCS databases are expected to require an amalgamation of *real-time* and *temporal* characteristics. While substantial work has been performed on these systems in isolation (see section 2 below for many references), not much is reported on their synthesis. In our research group at the University of Arizona, we have embarked upon the study of transaction processing in this important class of systems, i.e., ARCS databases. This paper is a report on part of this work.

1.1 Objective of This Paper

As mentioned previously, ARCS databases need to amalgamate *real-time* and *temporal* characteristics. Given this context, the objective of this paper is to perform an in-depth study on the problem of handling the rapidly changing information reported by sensors, termed sensor data. Due to the dynamic nature of this data, as well as specific reporting patterns, the transactions that manipulate this data have particular requirements. Specifically, these requirements are *recency* and *temporal consistency*. We study strategies to satisfy these requirements, while maintaining predetermined levels of *Quality of Service* (QOS) in the system. We identify a number of algorithms and perform a detailed performance study. This study enables us to make certain general recommendations regarding the properties of algorithms designed to handle rapidly changing sensor data.

The rest of the paper is organized as follows. In section 2, we identify research related to that reported in this paper. Section 3 explores the data and transaction characteristics of ARCS databases. In section 4 we discuss strategies to handle sensor data to satisfy the special requirements of transactions that manipulate this information. Section 5 describes the simulation model designed to evaluate the performance of the different algorithms proposed in the previous section, followed by a detailed performance study of the algorithms in section 6. Finally we provide a discussion and conclude in section 7.

2 Related Work

ARCS databases need to amalgamate characteristics of *Real-Time Database Systems* (RTDBSs) *Temporal Database Systems* (TDBSs). While individually these areas are fairly well explored, there is not much work at all on the synthesis of these fields, which is what this paper proposes.

There has been substantial research efforts devoted to *Real-Time database Systems* (RTDBSs) in recent years. Much of this effort has been focussed towards developing high performance scheduling algorithms [1, 2, 22, 25, 31, 12, 34] as well as concurrency control algorithms [21, 26, 27, 24, 23, 32]. For a nice discussion on the requirements of RTDBSs see [20] and for a nice survey of recent work see [48]. Typically, performance has been characterized as the ability to reduce transaction tardiness. None of this work has been performed with temporal consistency in mind. Very recently, a paper has appeared on issues concerning algorithms for temporal consistency preservation [47] – this is the first such paper we have encountered. Note that this paper simply suggests some modeling assumptions and simple algorithms – no detailed analysis or performance evaluation is presented. In other words, the problem addressed in this paper, namely satisfying real-time constraints while simultaneously being cognizant of recency and temporal consistency requirements, is a very open problem which has been recognized as being important.

Initiated by the pioneering work in [7, 44], *temporal databases* have been the focus of substantial work by researchers attempting to enrich conventional database semantics with those of time [9, 10, 5, 19, 17, 18, 28, 40, 41, 43, 46]. A large majority of these papers have explored either extensions to conventional data models such as relational or object-oriented, or have concerned themselves with the enrichment of conventional languages such as relational algebra and SQL with temporal augmentations. Note that none of this work has considered a real-time context along with the temporal dimension. Also, while notions of *temporal consistency* have been recognized [42], not much is reported on efficient means of enforcing such consistency requirements with regard to transactions. In general there is not a whole lot reported on transaction processing in temporal databases. A few notable exceptions are [33, 38], which are nice papers on temporal query processing, and [14] which is one of the very few papers that we are aware of, on temporal transaction processing.

Clearly, as evidenced by the aforementioned discussion, much work has been done on RTDBSs and TDBSs. However, very little work exists in synthesizing these areas. Yet an attempt to combine these areas results in new and unsolved problems. The need for synthesis has also been recognized by researchers (e.g., see [37]). A recent paper [3] looks at scheduling updates and transactions in a soft real-time database system, but they do not look at the issue of recency or temporal consistency maintenance. In fact they consider a case where only the most recent value of a data item is stored, thus the issue of providing temporal consistency does not arise at all. They also assume that updates are singleton writes and that job execution times are known. We do not.

3 Data and Workload Characteristics

3.1 Data Characteristics

In this paper we are primarily concerned with *sensor data.* Sensor (or measurement) data is the raw information that is received from the system monitoring processes. Since ARCS scenarios tend to be extremely dynamic, the values of sensor data items usually change very rapidly. An example of ARCS databases is *network management databases*, where the sensor data may include variables such as node queue lengths, retransmission rates, link status and call statistics. Typically sensor data represents dynamic performance characteristics of the system. One assumption we make is that sensors are distributed and work in their own private data partition – thus a particular instance variable is always reported by one particular sensor. Usually each sensor's data arrives in the database at a regular frequency under normal system operation (though aperiodic reporting may work as well). In typical ARCS databases (such as one for network management) several gigabytes of sensor data need to be processed per day [8]. Furthermore, sensor data would need to be versioned, in order to maintain a history of "environmental evolution" for post-mortem analysis as well as for future planning. In this paper we will assume that whenever sensor data item D_i is written, a new version of the data item is created (we shall use the notation D_i^v to denote version v of sensor data item D_i).Thus sensor data in ARCS databases, in addition to being somewhat similar to data in real-time systems, is of a temporal nature as well, because of the requirement that a "history" of data needs to be maintained. Therefore, with each version of a data item, there exists an associated *data timestamp.*

Definition 1 *The* **data timestamp** *of version v of data item D_i, i.e., D_i^v, denoted by $DTS(D_i^v)$, is the time at which the state of D_i in the outside environment was measured to be the value of D_i^v*

Clearly, the data timestamp value corresponds to the notion of *valid time* used in temporal database terminology [28]. There exists an important, and subtle, difference between sensor data in ARCS databases and the notion of data as assumed in traditional temporal databases. In classical temporal databases, researchers make the *value persistence between updates* or the *step-wise change* assumption [30] with regard to the mode of change in data values. In ARCS databases on the other hand, the data value change mode may be classified as *continuously changing.* This has certain implications with regard to which data items are provided to queries. The step-wise change assumption mandates that if the value of data item X is required at time t, then the system retrieves the version of X with the largest timestamp *prior to* t, even though another version of X may exist with a timestamp greater than, but closer (in terms of absolute difference) to t. In ARCS databases such a strategy would not work because of the "continuous" nature of sensor data. Rather, given two versions of X with timestamps on either side of t in the time axis, we would prefer to retrieve the version with the smallest absolute temporal distance from t. Thus the retrieved value may have occurred *after* the requested time, but was provided as a result of the *temporal proximity* criterion.

This has an impact on our model explained in section 3.3, in particular with respect to definitions 4 and 5.

3.2 Workload Characteristics

Examination of transaction characteristics yields three different types of transactions in ARCS databases [11, 45]: sensor reporting transactions (SRT), querying transactions (QT) and control-action transactions (CAT). In the current study, we only examine the first two types since only they manipulate sensor data.

Before examining the specific characteristics of SRTs and QTs, we note that all transactions have an associated *transaction timestamp.*

Definition 2 *The* **transaction timestamp** *of T_i, denoted by TTS(T_i), where T_i may be a SRT or a QT, denotes the time at which T_i was generated.*

We assume all sensor clocks and user clocks are synchronized.

3.2.1 State Reporting Transactions (SRT)

SRTs are transactions that are generated by sensors dispersed throughout the entity being modeled (e.g., a heterogeneous communications network). In most dynamic systems, these transactions are generated at a high frequency (e.g., every 15 seconds). They report the current states of real world objects and thus consist purely of `write`(D_i) operations. There also exists a unique mapping between a sensor S_i and a set of data items D, such that the write-set (WS) of every SRT generated by sensor S_i is D. Clearly, therefore, the SRT workload in an ARCS system monitored by s sensors is comprised of s SRT types, where each SRT type is reported by a particular sensor. We use the terminology $T^{srt,a}$ to denote SRT type a and $T_i^{srt,a}$ to denote an instance of a SRT of type a. Also, each sensor operates in its private data partition. In other words, all instances of a particular SRT type have identical write sets (i.e., $\forall i, j, WS(T_i^{srt,a}) = WS(T_j^{srt,a})$) and each SRT operates in its own data partition (i.e., $\forall a, b, WS(T_i^{srt,a}) \cap WS(T_j^{srt,b}) = \emptyset$).

In addition, the value updated by a particular SRT instance are independent of the current value of the data object; SRTs are thus *blind writes* [6]. There can thus be no write-write conflicts between SRTs, which renders concurrency control mechanisms unnecessary. However, since it is important to ensure that the database project a coherent state of the real world, SRTs need to be atomic, i.e., we cannot allow these transactions to commit their operations on some objects and not on others. As indicated section 3.1, each successful execution of an SRT results in the creation of new versions of the data items that it writes. We refer to this process as an *installation* of the SRT. During an installation, the data timestamp of the new version is set to the transaction timestamp of the SRT. For instance, when a SRT $T_i^{srt,a}$ with associated transaction timestamp $TTS(T_i^{srt,a}) = t$ updates a data item D_j, the new version D_j^v is assigned a data timestamp $DTS(D_j^v) = t$. Note that a SRT will typically arrive at the system at a time later than its transaction timestamp. This phenomenon has been termed *retroactive update* [29] and reflects the fact that an update ages between the time it is generated by a sensor and the time when it is received by the system. Such aging is often the result of network delays.

In this paper we study the execution of SRTs in detail. Two primary modes of SRT arrival are recognized, *periodic* and *aperiodic* (or sporadic). In the periodic SRT arrival mode, state reports arrive in a periodic fashion, without regard to whether the system state has actually changed or not. In the aperiodic case, sensors report values unpredictably, perhaps only when values change. We make the assumption that sensors report *periodically.*

3.2.2 Querying Transactions (QT)

QTs are transactions which want to read ARCS data values. They are generated by readers of ARCS data - typically humans users. While in a broader context, QTs might read from different data classes, in this study, we only look at their access of sensor data. The read-set (RS) of a QT may span data items reported by multiple sensors, i.e., a QT may want to read data items D_i and D_j where $D_i \in WS(T^{srt,a})$ and $D_j \in WS(T^{srt,b})$. A special feature of QTs is that instead of simply requesting values of data items, such requests also need to retrieve data values that were valid at particular instants in time. For example, a QT may be of the following form: `read(`D_i`)`, `read(`D_j`)` `at time = 100`. The reader can easily understand the need for such valid-time attachments to QTs by considering an example from the network management domain: if an alarm was logged at time t_1, and subsequently a QT was submitted as part of a post-mortem analysis of the problem, the QT would very likely wish to view values at t_1. We thus associate with each QT a *request timestamp.*

Definition 3 *The* **request timestamp** *of* T_j*, denoted by* $RQTS(T_j)$*, where* T_j *is a QT, denotes the time at which* T_j *requires the values of the items in its read set.*

We recognize that QTs may want to read the most recent values or older values. Therefore, the request timestamp may assume any value subject to the condition $RQTS(T^{qt}) \leq TTS(T^{qt})$.

Two requirements are associated with the successful execution of QTs:

1. **Recency:** The values read by a QT must be *recent* with respect to the time at which those values were requested. This is required as QTs are often used by operators for decision making and analysis purposes and the quality of analysis and the resultant decision is directly related to the quality of data the QT returns. One of the indicators of data quality is the degree of recency of the returned results with respect to the request timestamp.

2. **Temporal Consistency:** The values read by the QT must be temporally consistent, i.e., they must have occurred around the same time. Perfect temporal consistency, clearly, would mandate that these values *co-occur* in the system. As we shall see later, while perfect consistency is unachievable, some degree of temporal consistency must be provided in the system.

Aside from the two requirements above, we also impose a *Quality of Service* (QOS) requirement on QT execution. We quantify the QOS of QT processing by a soft real-time parameter called *Desired Response Time* (DRT). The name DRT is self explanatory – it is the response time that the system would like to process QTs in. It should be clear from the discussion thus far that an important component of sensor data processing in ARCS databases is the judicious allocation of resources to QTs and SRTs. Since SRTs typically arrive with higher frequencies than QTs, it would be very easy for the system to devote a lot of resources to SRT processing and let QTs wait. This situation however, is undesirable as operators expect to receive answers to their queries in a reasonable amount of time.

3.3 Recency and Temporal Consistency

As previously mentioned, the basic motivation in enforcing recency and temporal consistency is that the system must report values close to the time at which such values are requested, and when some decision needs several values as input, such values must be temporally consistent, i.e., they must have co-occurred in the system. Ideally, we would like to provide *perfect recency* (i.e., if the value of D_i is desired at time t_i, we would like to provide the version that represents that the value which occurred precisely at t_i) and *perfect consistency* (i.e., if values of D_i and D_j are desired at t_j, we would like to provide values that co-occurred precisely at t_j). To satisfy these ideal conditions it is necessary for our system to be able to guarantee two features:

1. Virtually every SRT needs to be installed – this will ensure that recency can be satisfied very closely, and

2. Every sensor must synchronize their respective SRT generation instants, i.e., if one sensor reports values at instant t_i, then every sensor must report their SRTs with *TTS* of t_i. Otherwise, if a QT arrives with a *RQTS* of t_i and its read-set spans the output of more than one sensor, temporal consistency cannot be guaranteed.

Clearly, given current processing resources the above requirements, especially requirement (1), are unsatisfiable, particularly when sensors report updates very frequently. Moreover, the system needs to provide reasonable response times for QTs as well, signifying that resources need to be awarded towards processing of QTs. This makes the task of rapid SRT processing even harder.

Thus, while recognizing that perfect recency and consistency are unachievable, it is our objective to provide *reasonable* recency and temporal consistency. Simply stated, reasonableness means: (a) if a value of D_i is desired at t_1, and it is not possible to perfectly satisfy this request, we would provide a value at a time t_2, such that t_2 was *close* to t_1, and (b) if values of D_i and D_j were desired at t_1, and perfect temporal consistency was unsatisfiable, we provide a value of D_i at t_2 and a value of D_j at t_3, such that t_2 and t_3 were reasonably close to each other, while, at the same time, both t_2 and t_3 were reasonably close to t_1. Note the preceding discussion was deliberately left rather imprecise to communicate our basic idea.

We use a measure called *Maximum Tolerable Delay* (MTD) to quantify the concept of *reasonable closeness* referred to in the previous paragraph. Using this parameter allows different application developers to set their own level of tolerance. Utilizing this measure, we define the notions of *recency* and *temporal consistency* used in this paper.

Definition 4 *Version v of data item D_i, i.e., D_i^v, is considered to be* **recent** *with respect to time t_j if and only if* $|\mathrm{DTS}(D_i^v) - t_j| \leq \mathrm{MTD}$.

Definition 5 *D_i^v and D_j^w are considered to be* **temporally consistent** *if and only if* $|DTS(D_i^v) - DTS(D_j^w)| \leq \mathrm{MTD}$.

Note that we use absolute values in these definitions. The reason is that the retrieved data versions could have timestamps which may be *smaller* or *larger* than the request timestamp. This is a result of the "continuous" value assumption discussed earlier in section 3.1. This is illustrated through the following example. Consider a QT T_i that wishes to read D_j. Further assume that two versions of D_j, D_j^v and D_j^w exist such that $DTS(D_j^v) \leq RQTS(T_i) \leq DTS(D_j^w)$. Moreover, $RQTS(T_i) - DTS(D_j^v) > MTD$ and $DTS(D_j^w) - RQTS(T_i) < MTD$. In such a situation, we contend that the transaction should be provided the value of D_j^w, even though this version occurs *after* the request timestamp of T_i. Our rationale is that by virtue of being within *MTD* time units, D_j^w offers a truer view of D_i with respect to *RQTS(T_i)* than does D_j^v. These assumptions could be changed without affecting in any way the subsequent work presented in this paper.

One other fact needs to be mentioned at this juncture. While the terms *state recency*, and *MTD* are introduced by us, definitions 4 and 5 above are by no means entirely our creations. The inspiration for these definitions are derived from the often cited paper by Ramamritham [36], where the terms *absolute* and *relative* consistency are used. A fairly similar model was also used in [3], where the authors use a parameter called *maximum Age*. Thus, our basic model has a lot of support in the literature. We have altered the notions in [36] in two ways: (a) we have taken the liberty of using the terms *state recency* and *temporal consistency* as they appear more semantically rich, and (b) instead of using two validity intervals, namely *AVI* (Absolute Validity Interval) and *RVI* (Relative Validity Interval), we use a single delay measure, *MTD* to characterize our notions. This leads to some additional simplicity in the model without losing information. Aside from these differences our definitions are the same. We hope that this similarity lends increased credence to our model, and in turn, lends increased justification to the subsequent analyses and proposed algorithms which rely on this model. Some readers may still question the practical usefulness of such a heuristic definition of recency and temporal consistency. We were convinced of the utility of this model based on extensive interactions with potential users of such a system (e.g., [8]) where it was revealed that the validity of an observed value is often (if not always) judged by the interval of time by which its observation instant differs from a particular time instant (for recency) or the observation instant of other values (temporal consistency). Thus our model not only builds on established notions in the research literature, it agrees with practitioners as well.

Another interesting fact to note is that definitions 4 and 5 mandate that data item values be updated at least once every *MTD* units. Otherwise, recency and temporal consistency may be unsatisfiable. In a periodic update arrival scenario, which we consider in this paper, this is easily achieved by requiring that all sensors have periods less than *MTD*. Thus, recency is always satisfiable in our system.

When a QT wishes to read a number of data items, it is necessary to ensure that the values returned are *both* recent and temporally consistent. Satisfying both properties simultaneously requires the concurrent satisfaction of the conditions specified in definitions 4 and 5 above.

Definition 6 *A set of data item versions $\mathcal{D}$, is considered to be* **recent** *with respect to a time t_i and* **temporally consistent** *if and only if the following two conditions are simultaneously satisfied:*

1. $\forall D_i^v \in \mathcal{D}, |\mathrm{DTS}(D_i^v) - t_i| \leq \mathrm{MTD}$;
2. $\forall D_i^v \in \mathcal{D}, \forall D_j^w \in \mathcal{D}, |\mathrm{DTS}(D_i^v) - \mathrm{DTS}(D_j^w)| \leq \mathrm{MTD}$.

Next, we turn our attention to designing mechanisms to attempt the satisfaction of above rules. The key factor that will influence the degree to which we satisfy recency and temporal consistency is how well we are able to install SRTs. If we can install SRTs well, transactions will read "good" values. We consider the "goodness" of the SRT installation procedure to be dependent on two interrelated factors.

1. How *efficiently* are SRTs installed? Efficient SRT installation signifies that SRTs are installed at a rate commensurate with their rate of arrival into the system. Given that it may be impractical to install every arriving SRT (because, e.g., resources may have to be devoted to QT processing) an efficient SRT installation procedure would end up installing a larger fraction of transactions than an inefficient one. If the SRT installation process is inefficient (i.e., the system is not able to install a large number of SRTs), transactions will be more likely to read non-recent values.

2. How *fairly* are SRTs installed? Given that a large number of SRTs are arriving rapidly and that different SRT types are arriving with potentially different frequencies, the system must ensure that it installs values of *all* data items and not *some* data items. If the SRT installation process is unfair, transactions will be more likely to retrieve temporally inconsistent values.

Essentially, the point being made above was that *efficiency* contributes to recency while *fairness* contributes to temporal consistency. Note however, that the two notions are related and impacting one will usually impact the other as well.

In systems such as those being considered in this paper, SRT patterns could be highly random and completely unpredictable, especially because of delays introduced by the intermediate network over which transactions must travel. The notion of fairness (item (2) above) is an important one in such scenarios. To see this, consider a database with two data items D_i and D_j. Moreover, assume that D_i and D_j are updated by different SRT types. An efficient but unfair installation procedure may be biased towards a particular SRT type and consequently update D_i very frequently but D_j very seldom. A fair installation policy on the other hand, would ensure that both data items are updated on a regular basis by being unbiased in its treatment of the different SRT types. Clearly, transactions that read updates of the second installation procedure would be expected to perform better control actions than those that read the output of the first policy.

Below we describe a number of algorithms that attempt to regulate the processing of QTs and SRTs in the system.

4 Algorithms

In this section we describe four algorithms which apply different strategies with the goal of fulfilling the requirements identified in the previous section.

4.1 Sensor Data Handler (SDH)

The SDH algorithm takes into account the special characteristics of sensor data and the transactions that access sensor data and attempts to fulfill the following objectives.

1. **Preservation of Recency** This property manifests itself through the read operations issued by QTs – if a large fraction of QT reads access stale values, recency is not being preserved well. SDH attempts to enforce this property by the *efficient* installation of SRTs, i.e., installing as large a fraction of the offered SRT load as possible.
2. **Offering Temporally Consistent Views** In addition to reducing overall staleness, SDH attempts to make sure that if the system absolutely has to suffer staleness (perhaps because of high transient load on the system), particular data items are not discriminated against. In other words, SDH will attempt to update data items in a way such that recency is preserved uniformly across all sensor data items. The key idea behind achieving this objective is *fairness* in SRT installation, i.e., making sure that all SRT classes get uniformly installed.
3. **Providing required QOS for QTs** As mentioned previously, ARCS applications are usually characterized by certain quality of service requirements, such as average response time requirements for user queries. This is captured in our model with the DRT parameter. SDH attempts to satisfy this objective through *priority based adaptive scheduling*, i.e., re-prioritizing SRTs and QTs in response to changing system states.

From the above discussion, two distinct components of SDH emerge – (a) *efficient and fair installation of SRTs*, and (b) a *scheduling* policy. These are described in the next two subsections.

4.1.1 SRT Installation

We first state the intuition behind our SRT installation strategy and then provide the details of our installation algorithm. As mentioned previously, the goals of our installation strategy are to minimize data staleness and preserve temporal consistency. Also, these objectives need to be achieved without usurping resources, such that QTs can get processed in a manner that satisfies the QOS requirements.

Since recency is clearly a primary goal, and a lot of SRTs are going to arrive at the system, we start out by asking the following question: "what is the minimum number of SRTs of a particular type that need to be installed in order to ensure that recency is preserved?" We now turn our attention to answering this question.

Consider an ARCS environment with *NumSensor* sensors, where sensor S_i has period P_i. We consider a general case, where all sensors need not have the same period (because some parts of the environment need closer monitoring than other parts), i.e., P_i is not necessarily equal to P_j. To preserve recency in the database (recall definition 4) it is only necessary to ensure that at least one instance of each SRT type is installed every *MTD* time units. If this were to be achieved, the system could guarantee that whenever a QT, say T_i^{qt}, requests the value of any data item D_j, a version D_j^v can always be found such that it is recent with respect to $RQTS(T_i^{qt})$, i.e.,

$$RQTS(T_i^{qt}) - MTD \leq DTS(D_j^v) \leq RQTS(T_i^{qt}) + MTD$$

Having considered the minimum number of SRT installations required to maintain recency, let us now consider the *maximum* number of SRTs that can possibly arrive at the system. This may be easily estimated by considering the period of sensors. Clearly, a sensor S_i with period P_i will generate at most $\frac{MTD}{P_i}$ SRTs every *MTD* units of time. Thus, given that there are *NumSensor* number of sensors in the system, the maximum SRT load $MAXLOAD_{srt}$ will be

$$MAXLOAD_{srt} = MTD \times \sum_{i=1}^{NumSensor} \frac{1}{P_i} \text{ per MTD time units}$$

Having discussed recency, we next turn our attention to *fairness*. To be fair among all SRT types, our goal is to install an identical fraction of SRTs of each type. This fraction should be such that it also preserves recency. Let $NUMSRT_i$ instances of SRT type i per *MTD* units of time denote this fraction. Our solution is as follows.

$$NUMSRT_i = \left\lceil \alpha \times \frac{MTD}{P_i} \right\rceil \tag{1}$$

The variable α is a dynamic control variable defined in the range $\frac{P_{min}}{MTD} \leq \alpha \leq 1.0$, where P_{min} is the value of the period of the sensor with the minimum period. It is easily seen when $\alpha = \frac{P_{min}}{MTD}$, the system will attempt to install the minimum number of SRTs required to maintain recency, i.e., 1 SRT per type. Conversely when $\alpha = 1.0$ the system will attempt to install all SRTs that arrive. The variable α is also dynamic, i.e., it adapts to the current load based on system feedback through a process described below in procedure SRT-ADMIT. Equation 1 expresses our basic strategy of SRT installation. However, we do not implement the equation directly. By simple algebraic manipulation we restate equation 1 as follows: we will attempt to install an instance of SRT type i every τ_i time units, where:

$$\tau_i = \frac{P_i}{\alpha} \tag{2}$$

Equation 2 expresses the actual strategy used in our SRT installation algorithm. This is shown below though procedures UPDATE-ALPHA and SRT-ADMIT.

Before stating the actual procedures we describe a structure called *ST_ARRAY* (Sensor Timestamp Array) that is used by the procedures. The *ST_ARRAY* is a memory resident array of dimension *NumSensor* where the i^{th} element denotes the transaction timestamp of the latest SRT generated by sensor S_i, i.e., the transaction timestamp of the most recent instance of SRT of type i.

```
                    Procedure UPDATE-ALPHA
    This procedure shows how the variable $\alpha$ is updated based on system feedback
on the completion of every $SampleBatch$ QTs do
    if Percentage of QTs that did not complete within DRT $> ThresholdMR$ then
        $\alpha = max(\frac{P_{min}}{MTD}, \alpha \times 0.95)$;
    else
        $\alpha = min(1.0, \alpha \times 1.05)$;
```

Procedure UPDATE-ALPHA dynamically updates the value of the variable α, based on system load. α is initialized to 1.0, such that the system initially attempts to install every arriving SRT. However if the system load is too heavy, this policy results in over-committing resources for SRTs, as a result of which QTs suffer. This is manifested in a lower QOS for QTs, resulting in high response times. This is measured by keeping track of what fraction of QTs are failing to meet their QOS requirement. When this fraction exceeds a predetermined threshold (indicated by the parameter *ThresholdMR*), the system reduces α until enough resources are devoted to QTs such as to bring response times down to acceptable levels. However, if system load should decrease at some point (because, e.g., QT arrival rates decrease), QTs exhibit better and better response times, at which time α is increased again to increase the efficiency of SRT processing. Procedure SRT-ADMIT below, outlines how SRTs are processed.

```
                    Procedure SRT-ADMIT
on arrival of SRT $T_i^{srt,j}$ the following is performed
  if $TTS(T_i^{srt,j}) - ST\_ARRAY[j] \leq \frac{P_j}{\alpha}$ then
      discard $T_i^{srt,j}$;
  else
      admit $T_i^{srt,j}$;
      $ST\_ARRAY[j]$ = $TTS(T_i^{srt,j})$;
  endif
```

Procedure SRT-ADMIT attempts to make sure that the SRT installation procedure is efficient by admitting SRTs selectively. The TRUE part of the **if** condition is the heart of this procedure. Recall that at any given moment, the system only wishes to install instances of SRTs of type j once every τ_j (i.e., $\frac{P_j}{\alpha}$) time units. Essentially, if an instance of an SRT of the same type as the current SRT (type j) was installed in less than $\frac{P_j}{\alpha}$ time, then the current SRT need not be installed. Procedure SRT-ADMIT then discards the SRT, thereby freeing up resources for use by transactions in the system. Also note that, if an SRT arrives out of order – due to network delays, this condition ensures that it will be discarded (since in that case, $TTS(T_i^{srt,j})$ - $ST\_ARRAY[j] \leq 0 \leq \frac{P_j}{\alpha}$). If a SRT instance is admitted, the appropriate element of the *ST_ARRAY* is updated, as shown in the procedure.

4.1.2 Scheduling

Having described how SRTs are handled, we finally describe how our algorithm performs on line scheduling across both transaction classes, i.e., we attempt to answer the question: "given that SRTs and QTs are going to be present in the system simultaneously, how does the scheduler decide which actions to perform at any moment"? Below we state a simple strategy to answer the above question.

The intuition behind the idea is as follows. At the outset, we award identical priorities to SRTs and QTs, i.e., the system does not discriminate among the two transaction classes. However, we monitor the system by checking whether QTs are reading recent values as well as temporally consistent values. In other words, we consider the results returned by QTs as indicators of how well the system is doing. Based on the results of this feedback, we follow a *greedy* strategy. If QT results are good (i.e., if recency and temporal consistency are mostly satisfied), we attempt to provide better QT response times by giving QTs higher priority. However, if QT results are below acceptable standards, we try to give SRTs higher priority. Details are given below.

First we describe a few notions that will be useful in detailing our scheduling strategy. Any read that accesses a non-recent value is termed a *stale read.* We use two feedback variables called *Fraction of Stale Reads* (FSR), and *Fraction of Temporal Consistency Violations* (FTCV) to characterize the system. These are defined below.

Definition 7 Fraction of Stale Reads *(FSR) is the fraction of read operations issued by QTs that return stale values, i.e.,*

$$FSR = \frac{\text{Number of reads returning stale values}}{\text{Total number of reads issued by QTs}}$$

Definition 8 Fraction of Temporal Consistency Violations *(FTCV) is the fraction of QTs aborted due to temporal inconsistency, i.e.,*

$$FTCV = \frac{\text{Number of QTs reading temporally inconsistent information}}{\text{Total number of QTs}}$$

Now we are in a position to describe the algorithm:

Procedure FLIP-FLOP

This procedure attempts to prioritize incoming SRTs and QTs with respect to each other. The system is initialized as $Priority_{srt} = Priority_{qt} = 1$.

```
after processing $SampleBatch$ QTs do
    if ($FSR_{batch} > ThresholdFSR$) or ($FTCV_{batch} > ThresholdFTCV$) then
        $Priority_{srt} \leftarrow 1$ and $Priority_{qt} \leftarrow 0$
    else
        $Priority_{qt} \leftarrow 1$ and $Priority_{srt} \leftarrow 0$
```

Procedure FLIP-FLOP is very simple – it monitors the quality of QT processing in the system and correspondingly alters the relative priorities of SRTs and QTs. The monitoring is achieved by keeping track of the FSR_{batch} and $FTCV_{batch}$ characteristics. The values of FSR_{batch} and $FTCV_{batch}$ being higher than acceptable levels signifies that QTs are reading "bad" values, which is the result of inadequate quality of SRT processing. Recall from previous discussion that if SRTs are processed well, "good" values should be available in the system. In this case therefore, the priority of arriving SRTs is increased, prompting the system to devote more resources for SRT processing. However, this strategy will restore "good" data values in the database, when the priorities of incoming QTs are increased.

4.2 Other Scheduling Algorithms

To evaluate the performance of SDH, we compared it to three other algorithms.

SRT First (SRTF): SRTF gives all SRTs higher priorities than all QTs. Within each class, transactions are scheduled on a FCFS basis.

QT First (QTF): QTF gives all QTs higher priorities than all SRTs. Within each class, transactions are scheduled on a FCFS basis.

No Priority(NP): NP gives identical priorities to both transaction classes. All transactions are scheduled on a FCFS basis.

4.3 The Correct Execution of QTs: Ensuring Temporal Consistency

Having described the different algorithms, we now turn our attention to a problem that we tacitly ignored in the preceding discussion, i.e., how do QTs retrieve data? Basically, QTs can only retrieve what has been installed. However, the reader can very well imagine that from the several data item versions that are potentially available, only certain particular versions need to be retrieved. More specifically, versions need to be retrieved in such a way that recency and temporal consistency are satisfied (if possible). In this section we elaborate on the retrieval mechanism to fetch versions that best satisfy QT requirements. Note that the described procedure is independent of the algorithms described above. It is simply a retrieval mechanism that is applied to all four algorithms.

We will illustrate the retrieval procedure with an example: Consider a QT T that wants to retrieve data items D_i, D_j, D_k in that order. Further assume $RQTS(T) = t_r$. The first operation to be executed would be $read(D_i)$. This operation would select version v of data item D_i, i.e., D_i^v, such that there exists no other version of D_i with a timestamp value closer to t_r. In other words, D_i^v is the *most recent* value of D_i with respect to t_r. Having selected D_i^v, we then select the appropriate versions of D_j and D_k, such that temporal consistency conditions are satisfied between D_i, D_j and D_k. There are two subtle points to note here:

1. Assume that the final result of retrieval for the above QT is D_i^v, D_j^w, D_k^z. D_i^v was chosen by virtue of being the version of D_i most recent to t_r. Now, when D_j^w and D_k^z are chosen, the overriding criteria is that they should be temporally consistent with D_i^v and with each other. Thus, these versions may not be the closest versions of data items D_j and D_k to t_r.

2. In order to satisfy recency the following condition needs to be satisfied as well: (a) $|D_i^v - t_r| \leq MTD$, (b) $|D_j^w - t_r| \leq MTD$, and (c) $|D_k^z - t_r| \leq MTD$, i.e., all the retrieved versions should satisfy recency with respect to t_r. However, if, for some reason, no updates were installed in that period, this condition may be unsatisfiable.

The above high level discussion outlines the basic requirements of a retrieval policy that seeks to ensure temporal consistency among retrieved data items. A straightforward implementation of such a policy (pairwise comparison between data items) is highly inefficient, requiring $O(N^2)$ time, where N is the number of data items. In real-time scenarios such an implementation will be unacceptable. Below we describe a procedure called *Interval Adjustment Technique* (IAT) that implements our retrieval requirements efficiently.

The intuitive basis of IAT is as follows. In processing the retrieval requirements of a QT T, we first define an acceptable timestamp interval based on *RQTS(T)*. Subsequently, based on each read access by T, we modify this timestamp interval by factoring in the data timestamp of the data item that was just read. Also, each data access is performed such that the data timestamp of the accessed data item is within the current acceptable timestamp interval. If such data items can be found for the entire read set of T, then temporal consistency is ensured. A procedural description of IAT follows. We denote the *Acceptable Timestamp Interval* of T by the notation ATI_T.

Procedure IAT

This procedure attempts to ensure temporal consistency across the read set of each QT. The following pseudo-code is written with respect to a QT T, having request timestamp *RQTS(T)*

```
$ATI_T = [RQTS(T) - ATIFactor \times MTD,\ RQTS(T) + ATIFactor \times MTD]$; /* Initialize */
foreach read access request for data item $D_i$
  if no version of $D_i$ exists with timestamp in the range given by $ATI_T$ then
      mark $T$; /* Temporal Consistency Violation */
  endif
  retrieve the version of $D_i$, $D_i^v$ with $DTS(D_i^v)$ closest to the midpoint of $ATI_T$;
  $ATI_T \leftarrow ATI_T \cap [DTS(D_i^v) - MTD,\ DTS(D_i^v) + MTD]$; /*Interval Adjustment*/
```

The IAT procedure is simple: it first initializes the *ATI* of T to a large interval (characterized by the parameter *ATIFactor* centered on the request timestamp of T. Then, with each data access request, if it is not possible to find a version of the data item in the ATI_T interval, then T is marked as temporal consistency cannot be preserved. If such versions are found, an appropriate version is retrieved and ATI_T is adjusted to factor in the data timestamp of the recently accessed version. The version closest to midpoint is retrieved to ensure that the largest possible ATI_T results after the adjustment. Other retrieval rules (such as version closest to *RQTS(T)*) will work as well, at

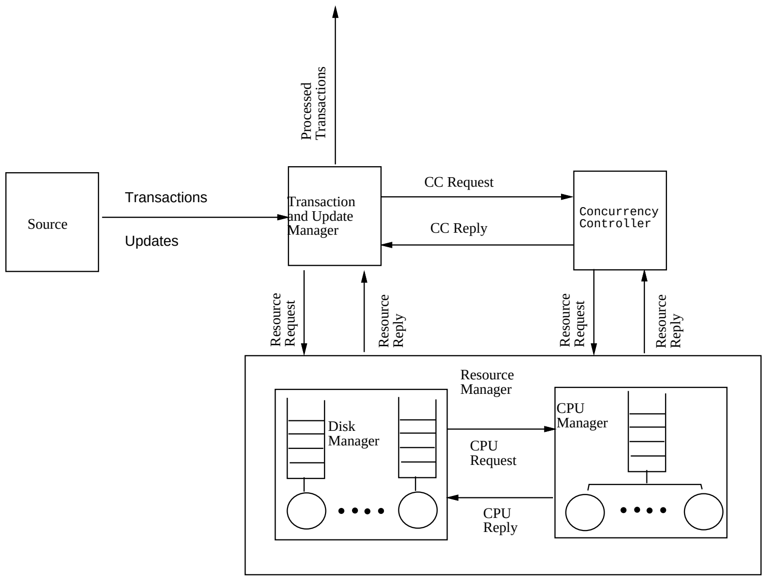

Figure 1: The RTDBS Model

the cost of perhaps overly restricting the acceptable interval to a point where feasible versions may exist, but will not be found.

5 Simulation Model and Performance Metrics

To evaluate the performance of the SDH, SRTF, QTF and NP algorithms outlined above, we conducted extensive simulation experiments. Our simulation programs were written in SIMPACK, a C/C++ based simulation toolkit [15, 16].

The simulated database system consists of a shared-memory multiprocessor that operates on disk-resident data. We model the database as a collection of pages. A QT is a sequence of read page accesses, while a SRT is a sequence of write page accesses. A read request submits a data access request to the concurrency control (CC) manager, on the approval of which, a disk I/O is performed to fetch the page into memory followed by CPU usage to process the page. Write requests are handled similarly, with the exception a second I/O performed to write the page – note that this second I/O is deferred till commit time. In other words a write submits a data access request to the concurrency control (CC) manager, on the approval of which, a disk I/O is performed to fetch the page into memory followed by CPU usage to process the page. If the transaction successfully commits, all pages are written back to disk – thus the write I/O is deferred till commit time.

Our database model is shown in figure 1. There are four major components:

1. An *arrival generator*, that generates the workload, i.e., SRTs and QTs
2. A *transaction manager* that models SRT and QT execution and implements the algorithms
3. A *concurrency controller*, that implements the CC algorithm, in our case *multiversion timestamping* [13].

4. A *resource manager* that models system resources, i.e., CPUs and disks and the associated queues

5.1 Resource Model

Our resource model considers multiple CPUs and disks as the physical resources. For our simulations we assumed the data items are uniformly distributed across all disks. The *NumCPU* and *NumDisk* parameters specify the system composition, i.e., the number of each type of system resource. The parameter *DBSize* denotes the number of data pages, while a parameter *SensorFrac* is used to denote the fraction of data pages that correspond to sensor data, the implicit assumption being that a data page either contains all sensor data or all non-sensor data. There is a single queue for the CPUs and the service discipline is assumed to be preemptive-resume based on the task (i.e., transactions or updates) priorities. Each individual disk has its own queue with a non-preemptive, transaction priority based service discipline. The parameters *ProcCPU* and *ProcDisk* denote the CPU and disk processing time per page respectively. These parameters are summarized in table 1.

5.2 Workload Model

Our workload model consists of modeling the characteristics of SRTs and QTs as well as their arrival rate.

5.2.1 SRTs

As mentioned before, our system assumes periodic SRT arrival, with the guarantee that no two SRTs of the same type arrive more than MTD units apart. In other words, no sensor has a period greater than MTD. This restriction is imposed to make recency always satisfiable in the system. Also recall that the working sets of SRT types are pairwise disjoint, i.e., each sensor works in its private data partition. Also recall that corresponding to each sensor there is a particular SRT type. Thus, the number of different SRT types is equal to the number of sensors in the system, given by the parameter *NumSensor*. In the operational mode, each sensor periodically dispatches an instance of its corresponding SRT type. Clearly, to generate the SRT workload, all we need to figure out are (a) the period of each sensor, and (b) the members of the data partition of each sensor, i.e., the write set of its corresponding SRT type. Consequently, we simulate the SRT workload through the following steps:

1. For each sensor S_i, $i = 1, 2, \ldots, NumSensor$, we first generate a period P_i, such that $P_i \leq MTD$
2. For each sensor we generate the working set of its corresponding SRT type, i.e., the data items that the sensor reports

Step (1) is performed as follows: the period for each sensor is determined by generating an exponentially distributed random variable with mean β, such that $\beta = SRTLoadFac \times MTD$, where *SRTLoadFac* is a parameter such that $0 < SRTLoadFac < 1.0$. Note that with a low value of *SRTLoadFac*, sensors on an average have smaller periods, thereby increasing the SRT arrival rate. By varying *SRTLoadFac* we control the SRT load into the system. Note that simply generating an exponentially distributed period is not adequate, as the generated value may be greater than MTD. This would violate the requirement that all period values must be less than *MTD*. We thus use a *truncated* exponential distribution, i.e., if a generated value is greater than *MTD* we keep on generating values until we get one in the desired range, i.e., less than *MTD*.

Step (2) is performed in two sub-steps: we first generate a *size* for each SRT type (i.e., the cardinality of the working sets), and subsequently generate the disjoint working sets corresponding to the sizes. The size for each SRT type is chosen from a Poisson distribution with mean λ, such that

$$\lambda = \frac{DBSize \times SensorFrac \times NumItemsPerPage}{NumSensor}$$

where *NumItemsPerPage* denotes the number of data items per page. We ensure that the sizes sum up to the number of sensor data items by utilizing a truncated Poisson distribution with redistribution. Once the sizes are generated we generate the actual working sets by randomly drawing (without replacement) a number of sensor data items equal to the pre-determined size of that SRT class.

Having generated the SRT types and their periods, the process of workload arrival into the system is very simple. An instance of SRT type SRT_i, with period P_i, is generated every P_i time units, and arrives at the system after an exponential delay with a mean of δ. The delay simulates unpredictable communication delays that would very likely be experienced in real systems.

5.2.2 QTs

QTs arrive at the system as a Poisson stream with a mean rate of *ArrivalRate*. We consider two broad classes of QT characteristics: (a.) $Size_T$, the number of pages accessed by T ; and (b.) $RQTS_T$, the request time stamp of T. Transaction sizes belong uniformly in the range *SizeInterval*. $RQTS_T$ is generated to be uniformly distributed between $[(Current\_Time - ATIFactor \times MTD), Current\_Time]$. Our workload parameters are summarized in table 1.

Parameter Type	**Notation**	**Description**
Resource	*NumCPU*	Number of CPUs
	NumDisk	Number of disks
	ProcCPU	CPU time /data page
	ProcDisk	Disk time/data page
Workload	*ArrivalRate*	Transaction Arrival Rate
	DBSize	Number of pages in the database
	NumItemsPerPage	Number of data items per page
	SensorFrac	Fraction of sensor data pages
	SRTLoadFac	Factor controlling SRT load on the system
	SizeInterval	Range of the number of pages accessed per transaction
	RQTSInterval	Range from which request timestamps are drawn
	ATIFactor	Request Timestamp interval for QTs

Table 1: Input Parameters to our RTDBS Model

5.3 Performance Metrics

The following performance metrics are used to evaluate the different algorithms:

Fraction of QOS Failures (FQF): This is the fraction of QTs that completed after the system-wide response time requirement and is computed as:

$$\text{FQF} = \frac{\textit{number of QTs processed with response times greater than DRT}}{\textit{total number of QTs processed}}$$

It is an indicator of the QT processing efficiency of the system and of how well the system can meet the desired QOS. In other words, FQF measures the real-time performance of the system.

Clean QT Throughput (CQT): This measures "clean" or "correct" QT throughput, i.e., the average number of QTs processed per second without any stale reads or temporal consistency violations. It is thus, as well, an indirect measure of how well SRTs are installed in the system, i.e., a measure of *temporal* efficiency of the system.

6 Performance Results

6.1 Baseline Results

We first report a baseline experiment and subsequently report results of other experiments conducted by varying one parameter at a time. Note that in the ensuing discussion, for clarity, we use a **bold** font to denote acronyms for algorithm names (i.e., **SDH, QTF, SRTF, NP**) and an *italicized* font for the acronyms corresponding to performance metrics (i.e., *FQF* and *CQT*).

The baseline parameter settings for our experiments are shown in Table 2 below. Based on these parameter settings it is easily seen that the average SRT load on the system is 16.66 SRTs per second and each SRT, on an average touches 4 pages. These numbers were chosen with the goal of having a "reasonable" SRT load on the system. Reasonableness was defined as follows – the SRT load should not be high enough to cause overload in the system as this would render the system unable to effectively process QTs. Also, the SRT load should not be so low as to cause negligible resource contention in processing QTs. At the above load it was found that the system had to work hard to allocate resources smartly to the transaction classes, creating a nice test-bed for our algorithms. Also, resource availability levels were kept at a constrained level by setting *NumCPU* and *NumDisk* to 2 and 4 respectively.

Figures 2A and B plot the *FQF* and *CQT* metrics for each algorithm in the baseline case. We first consider QOS results, shown in figure 2A. All the curves show an 'S' shape, which indicates that as QT arrival rates increase, the ability of the different algorithms to meet the QOS requirements decreases, until no QTs can be processed within DRT time units. Two features of these curves are noteworthy – (a) the respective overload points[1], and (b) the relative slopes after the overload point. Regarding the overload point, it is clear that **SRTF**, which uniformly favors SRTs, becomes unable to efficiently process QTs earlier than all other algorithms, at an arrival rate of about 2 QTs per second. It is closely followed by **NP** (at about 3 QTs per second), which does not discriminate between transaction classes. **QTF**, on the other hand, being strongly biased towards QTs achieves the highest overload point (about 11 QTs per second). Finally, **SDH**, which attempts

[1] In these experiments, the *overload point* corresponds to the highest QT arrival rate at which a given algorithm can still process all QTs within DRT time units.

Parameter	**Value**
MTD	60 sec
ProcCPU	10ms
ProcDisk	20ms
NumCPU	2
NumDisk	4
NumSensor	250
DBSize	1000 pages
SensorFrac	0.9
ItemsPerPage	1
NumSDItems	900
SRTLoadFac	.25
SizeInterval	[1,15]
SRInterval	[2,6]
RQTSInterval	$[Current\_time - 5 \times MTD,\ Current\_time]$
ATIFactor	5
ThresholdMR	5%
δ	1
SampleBatch	100
DRT	2.5 sec

Table 2: Parameters Settings for Our Experiments

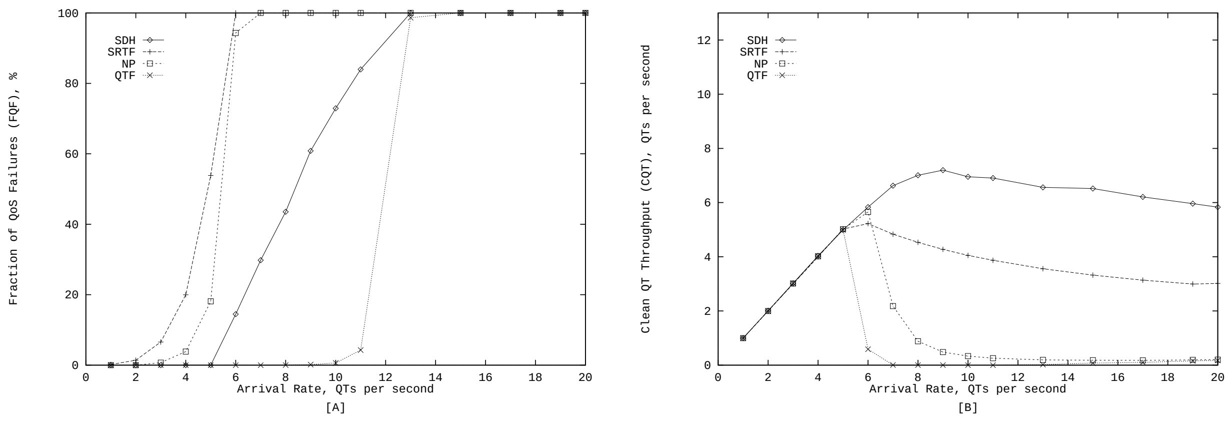

Figure 2: Baseline Results

to intelligently balance QT and SRT processing achieves better results than both **SRTF** and **NP**, yet does not reach the *FQF* performance levels afforded by *QTF*. A more interesting aspect of these curves, however, lies in their slopes after overload. This feature is an indicator of how well the different algorithms are able to cope with overload situations in terms of QT processing. **NP** appears to be the worst performer: between QT arrival rates of 5 and 6 transactions per second, it jumps from being unable to process less than 20% to more than 90% of all QTs within DRT time units. This low performance is closely followed by that of **SRTF** and **QTF**. Both algorithms suffer from the same inability to intelligently balance their SRT and QT loads. As a result, both are equally unable to adapt to an increased QT load past their overload point; the performance of **QTF** is constrained by resource availability, and that of **SRTF** by SRT processing. **SDH**, however, sees its performance degrade significantly more gracefully than the other three algorithms, which is clearly evidenced by the proximity of its curve to that of **NP** at low arrival rates (*FQF* of 0 at 5 QTs per second) and to that of **QTF** under extreme overload (*FQF* of 100% at 13 QTs per second). This remarkably low rate of performance degradation is due to the fact that **SDH**, by virtue of discarding certain SRTs (recall procedure SRT-ADMIT) lessens system load, thereby ameliorating to some extent the effects of overload.

Having examined the QOS curves, we now turn our attention to the *CQT* metric, which, in a sense, measures the "true" performance of the algorithms. As mentioned on several occasions in this paper, a query truly succeeds when it reads both recent *and* temporally consistent values. The *CQT* metric measures the throughput of such "correctly" executed queries. The *CQT* curves are plotted in figure 2B, and as is immediately apparent, **SDH** substantially outperforms all the other algorithms. The basic shape of the curves is explained as follows. At low QT loads, the system is able to install a large fraction of SRTs, which leads to the correct execution of virtually the entire offered QT load. This fact manifests itself in all the curves following a $\angle 45°$ through the origin. After overload however, fewer and fewer SRTs are installed. *CQT* decreases for all algorithms. The noteworthy feature regarding the curves is the difference in the degree of descent after overload. **QTF** and **NP** exhibit a sharp downward trend, whereas **SDH** and **SRTF** display a markedly gentler descent pattern, with **SDH** showing the flattest descent. This difference is significant. In **QTF**, once QT load reaches a certain level (in this case $\approx$ 5 transactions per second) the system is unable to install all SRTs. Subsequently, any further increase in QT arrival rate results in sharp drops in the SRT installation capability of the algorithm, and soon, virtually no SRTs can be installed. At this stage, *CQT* reaches 0, as all data values become stale. **NP** closely follows the same trend: even though it has no inherent bias towards a given class of transaction, it rapidly becomes unable to process SRTs efficiently enough to ensure that the necessary version of a given data item will be available when requested. At high loads, almost no SRTs are processed in time to ensure recency and temporal consistency. **SRTF** and **SDH** on the other hand, always process enough SRTs in time, leading to the less steep descent. This descent is due to the fact that, as QT load increases, QTs spend longer periods of time waiting to access resources (i.e., CPUs and disks). As a result throughput falls off. Finally **SDH** exhibits significantly superior performance to all the other algorithms. It is able to follow the upward trend longer than the other algorithms (in our case until $\approx$ 8 QTs per second) as a result of judiciously discarding some SRTs without sacrificing fairness. The reader may enquire as to why it is able to keep its upward trend longer than **SRTF**. The reason is as follows. **SRTF** installs all SRTs, whereas **SDH** may discard some when freshness will not be violated. Thus, **SDH** is able to create more opportunities for QTs to be processed, *without sacrificing freshness.* After this point performance does drop, but less than the other algorithms, as the QOS feedback mechanism kicks in, and makes sure some SRTs always get processed while simultaneously attempting to ensure that QT response times remain below a threshold.

6.2 Effect of Varying Resource Contention

In this experiment we studied how sensitive the algorithms were to resource contention levels in the system. Since the baseline case studies a fairly tightly resource constrained systems (a dual processor scenario), in this experiment we reduced the level of resource contention[2] by setting *NumCPU* = 3 and *NumDisk* = 6. The curves plotting *FQF* and *CQT* are shown in figure 3A and B. Figure 3 reconfirms the results of figure 2, displaying fairly identical trends. As a result of

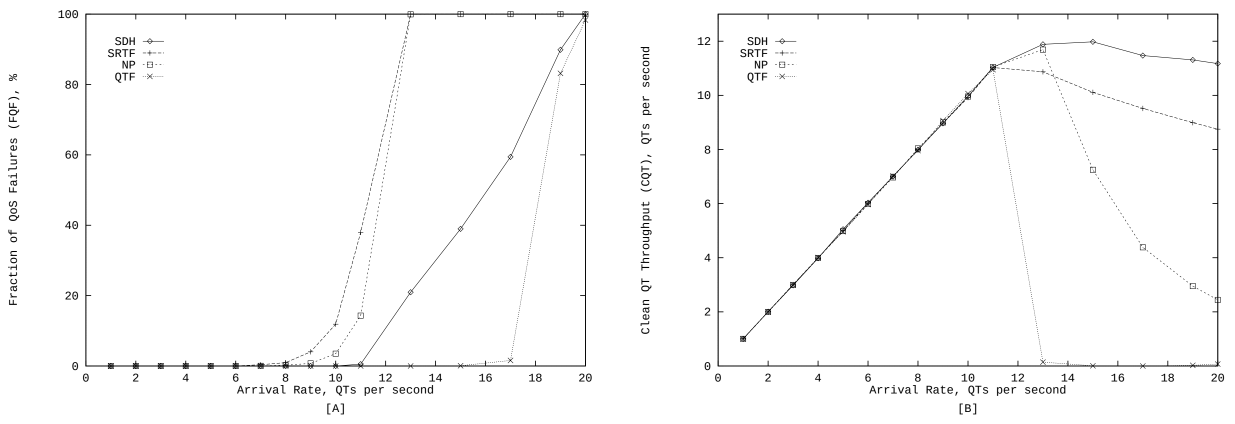

Figure 3: Sensitivity to Resource Availability Levels

increased resource availability, the overload points are shifted rightward, since all algorithms are able to process more transactions per time unit. Of particular interest is how well **SRTF** performs with respect to *FQF*. Whereas the other algorithms see their rate of performance degradation after overload essentially unchanged, **SRTF** is able to take better advantage of the increased resource availability as is evidenced by its lower *FQF* slope after overload.

6.3 Effect of Varying SRT Load

We studied two cases with regard to varying SRT loads. In one experiment SRT load was increased to the point of overload, by setting *SRTLoadFac* = .10. In this experiment, because of its superior SRT installation procedure, **SDH** performed much better than the other algorithms. Thus this experiment did not show anything unexpected, and we do not report the results here. More interesting was the experiment when SRT load was decreased by setting *SRTLoadFac* = 0.50. This basically halved the average SRT load on the system in comparison to the baseline case. For these experiments *NumCPU* and *NumDisk* were again reset to their baseline values. Figure 4A and B show the relevant graphs. While the basic facts are similar to the baseline results, there is one interesting difference. To appreciate this fact, compare figure 4B and figure 2B. In particular, note the changes in performance displayed by **SRTF** and **SDH**. **SRTF**, which suffered much in the baseline case, is now able to take advantage of the reduced load on the system to process more QTs. At high arrival rates, its separation from the **SDH** curve is almost halved. On the other hand, the performance of **SDH** remains almost exactly the same. This is a clear demonstration that **SDH** is almost entirely immune to changes in SRT loads which are effectively handled by the SRT-ADMIT procedure. It is also a strong indicator of the scalability of the algorithm.

[2]We also studied a uniprocessor case, i.e., even more resource constrained than the baseline case. In this experiment **SDH** performed even better than the other algorithms than in the baseline case, and is thus not as interesting as the current case, where the other algorithms are given more advantage compared to **SDH**. For keeping this paper reasonably sized, we do not report the results of the uniprocessor experiment

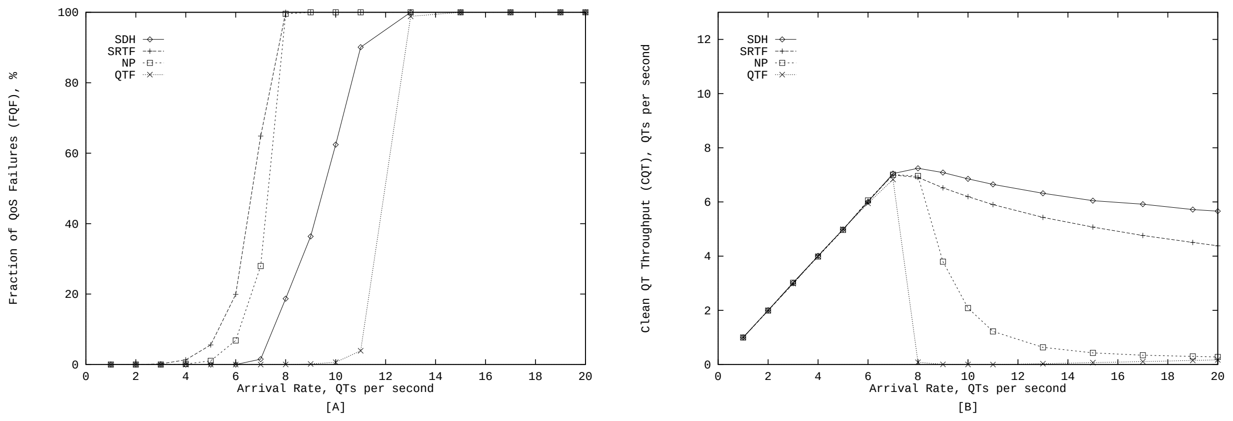

Figure 4: Sensitivity to SRT Load

7 Discussion and Conclusion

In this paper, we studied how to effectively handle the update and access of sensor data elements. Sensor data are primarily handled by two classes of transactions: SRTs, which write sensor data and QTs, which read them. To ensure sensor data correctness, it is necessary to enforce two criteria, namely *recency* and *temporal consistency*, which we defined based on an interval measure termed *Maximum Tolerable Delay* (MTD). We showed that in order to enforce these criteria, three actions need to be performed.

- SRTs must be installed *efficiently*, i.e., as much as possible of the offered load should be installed to prevent the data items from becoming stale
- The SRT installation process must be *fair*, i.e., all SRT classes should be installed similarly often. This would aid in preserving temporal consistency in the system
- QTs need a procedure to retrieve temporally consistent data

With these three goals in mind, we postulated **SDH**, an algorithm that attempts to preserve recency and temporal consistency across sensor data elements. **SDH** attempts to attain this goal by awarding priorities to QTs and SRTs depending on system load and performance, and judiciously discarding some SRTs.

We also postulated three other simple algorithms, **SRTF**, **QTF** and **NP** and proceeded to evaluate the performance of these four algorithms. Some important findings from this study are summarized below. We first summarize specific findings and then state some general lessons learned from this study.

1. **SDH** emerges as the best performing among all four algorithms. It outperforms the others across a wide range of system loading and resource availability levels. While **QTF** exhibits better real-time performance than **SDH**, **SDH** allows the maximum number of QTs to execute cleanly, i.e., without recency and temporal consistency violations. One nice feature of **SDH** is that unlike many other high performance algorithms which are designed to handle overload, **SDH** demonstrates good performance regardless of whether the system is underloaded or overloaded. Another remarkable feature is that while the other algorithms suffer dramatic performance degradations under overload

(e.g., response time for **SRTF**, and clean throughput for **NP** and **QTF**) **SDH**, in almost all cases degrades gracefully, demonstrating the power of its load sensitivity. In the case of the *CQT* metric, **SDH** holds its performance almost constant under overload, which is remarkable.

2. Of the four algorithms described, **NP** has the least computational overhead. Its performance was studied with the objective of understanding the effects of prioritization, which occurs, in one form or another, in all other algorithms. In other words, we were interested in learning whether anything could be gained by prioritizing reads and writes to sensor data items. Some interesting results emerge in this regard. It appears that the benefits of prioritization depend on the "rapidity" of change of the state of the outside world. In applications where the state changes very rapidly (e.g., air traffic control or telecommunications applications where SRT frequencies of 10-15 per second are the norm [8]), some sort of prioritizing scheme between the readers and writers of sensor data is desirable. This recommendation is borne out by the results of our first two reported experiments, i.e., the baseline experiment and the experiment with varying resource contention. In both these experiments, where the outside environment was changing rapidly (high SRT load), **NP** performed consistently worse than **SDH** and was outperformed by **SRTF** with respect to *CQT* and **QTF** with respect to *FQF*. The other notable fact in this respect is that the level of resource contention does not appear to have a substantial impact on the attractiveness of prioritization. This appears to indicate that before spending a lot of money to purchase a better system, it may make sense to tinker around with different prioritizing schemes to see if performance improves.

3. A general lesson is learned regarding the relative importance of readers and updaters of sensor data. It is intuitive that **QTF** is optimal with respect to QT lateness. However, it is observed from the experiments that **SRTF** resoundingly outperforms **QTF** with respect to *CQT*, which, arguably, is more accurately representative of performance than the *FQF*. It may be accepted as a general rule of thumb that in the absence of any other sophisticated strategies, writers to sensor data should be given more importance than readers of sensor data. The only exception to this rule of course is when QT load is high and response time is of great concern.

In the future we plan to extend this work by looking at other problems concerning ARCS databases. It is encouraging to us that the notion of ARCS databases is receiving some recognition among other researchers and practitioners (e.g., see [45]).

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of the 15th VLDB*, 1989.

[2] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: Performance Evaluation. *ACM Transactions on Database Systems*, 1992.

[3] B. Adelberg, H. Garcia-Molina, and B. kao. Applying Update Streams in a Soft Real-Time Database System. In *Proceedings of the 1995 ACM SIGMOD*, pages 245–256, 1995.

[4] I. Ahn. Database Issues in Telecommunications Network Management. In *ACM SIGMOD*, pages 37–43, May 1994.

[5] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, 1986.

[6] P. Bernstein, V. Hadzilakos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[7] R. L. Blum and G. Wiederhold. Studying hypotheses on a time-oriented clinical database: An overview of the rx project. In *Proceedings of the Symposium on Computer Applications in Medical Care*, pages 725–735, Washington, DC, 1982. IEEE Computer Society.

[8] Sprint Network Management Center. Site Visit, April 1992.

[9] S. Chakravarthy and S-K. Kim. Semantics of Time Varying Information and Resolution of Time Concepts in Temporal Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages G1–G13, 1993.

[10] J. Clifford and A. Tansel. On An Algebra for Historical Relational Databases. In *Proceedings of the International Conference on Management of Data* , pages 247–265, 1985.

[11] A. Datta. Research Issues in Databases for Active Rapidly Changing data Systems (ARCS). *ACM SIGMOD RECORD*, 23(3):8–13, September 1994.

[12] A. Datta, S. Mukherjee, P. Konana, I. Viguier, and A. Bajaj. Multiclass Transaction Scheduling and Overload Management in Firm Real-Time Database Systems. *Information Systems*, 21(1):29–54, March 1996.

[13] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, 1994.

[14] O. Etzion, A. Gal, and A. Segev. Temporal Active Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, June 1993.

[15] P.A. Fishwick. Simpack: Getting started with simulation programming in C and C++. Technical Report TR92-022, Computer and Information Sciences, University Of Florida, Gainesville, Florida, 1992.

[16] P.A. Fishwick. *Simulation Model Design And Execution: Building Digital Worlds.* Prentice Hall, 1995.

[17] S.K. Gadia. A homogeneous relational model and query languages for temporal databases. *tods*, 13(4):418–448, dec 1988.

[18] S.K. Gadia and G. Bhargava. Sql-like seamless query of temporal data. In R. T. Snodgrass, editor, *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, jun 1993.

[19] S.K. Gadia and J.H. Vaishnav. A query language for a homogeneous temporal database. In *pods*, pages 51–56, mar 1985.

[20] M.H. Graham. Issues in Real-Time Data Management. *Journal of Real-Time Systems*, 4(3):185–202, September 1992.

[21] M.H. Graham. How to Get Serializability for Real-Time Transactions Without Having to Pay For It. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 56–65, 1993.

[22] J. Haritsa, M. Livny, and M. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1991.

[23] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. On being optimistic about real-time constraints. *ACM PODS*, 1990.

[24] J.R. Haritsa, M.J. Carey, and M. Livny. Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1990.

[25] J.R. Haritsa, M.J. Carey, and M. Livny. Data Access Scheduling in Firm Real-Time Database Systems. *The Journal of Real-Time Systems*, 4, 1992.

[26] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In *Proceedings of the 17th Intl. Conf. on Very Large Data Bases*, 1991.

[27] J. Huang, J. Stankovic, D. Towsley, and K. Ramamrithnam. Experimental Evaluation of Realtime transaction processing. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[28] C. Jensen, J. Clifford, S.K. Gadia, A. Segev, and R.T. Snodgrass. A Glossary of temporal Database Concepts. *ACM SIGMOD RECORD*, 21(3):35–43, 1992.

[29] C.S. Jensen and R. Snodgrass. Temporal Specialization and Generalization. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):954–974, 1994.

[30] C.S. Jensen and R. Snodgrass. Semantics of Time-Varying Attributes and Their Use for Temporal Database Design. In *Proceedings of the Object-Oriented and Entity Relationship Conference*, Gold Coast, Australia, December 1995.

[31] G. Koren and D. Shasha. D^{over}: An optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 290–299, August 1992.

[32] J. Lee and S.H. Son. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1993.

[33] T.Y.C. Leung and R. Muntz. *Stream Processing: Temporal Query Processing and Optimization*, chapter 14, pages 329–355. Benjamin/Cummings, 1993.

[34] H. Pang, M. Livny, and M.J. Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[35] B. Purimetla, R.M. Sivasankaran, and J. Stankovic. A Study of Distributed Real-Time Active Database Applications. In *IEEE Workshop on Parallel and Distributed Real-time Systems*, April 1993.

[36] K. Ramamritham. Real-Time Databases. *Distributed and Parallel Databases: An International Journal*, 1(2):199–226, 1993.

[37] K. Ramamritham. Time for Real-Time Temporal Databases. In *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, pages DD1–DD5, 1993.

[38] A. Segev and H. Gunadhi. Query Processing in Temporal Databases. In *Proceedings of the Workshop on Query Optimization*, pages 159–164, Portland, Oregon, may 1989.

[39] R.M. Sivasankaran, B. Purimetla, J. Stankovic, and K. Ramamritham. Network Services Databases - A Distributed Real-Time active Database Applications. In *IEEE Workshop on Real-time Applications*, May 1993.

[40] R. Snodgrass. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.

[41] R. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of ACM SIGMOD*, June 1985.

[42] X. Song and J.W.S. Liu. How Well Can Data Temporal Consistency be Maintained? In *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design*, 1992.

[43] M.D. Soo. Bibliography on Temporal Databases. *ACM SIGMOD RECORD*, 20(1):14–23, 1991.

[44] G. Wiederhold. How to write a schema for a time oriented medical record data bank. Technical report, Stanford, 1973.

[45] A. Wolski, J. Karvonen, and A. Puolakka. The RAPID Case Study: Requirements for and the Design of a Fast-Response Database System. In *Proceedings of the First International Workshop on Real-Time Databases*, pages 8–14, March 1996.

[46] G.T.J. Wuu and U. Dayal. A Uniform Model for Temporal Object-Oriented Databases. In *Proceedings of the International Conference on Data Engineering*, pages 584–593, 1992.

[47] M. Xiong, J. Stankovic, K. Ramamritham, D. Towsley, and R. Sivasankaran. Maintaining Temporal Consistency: Issues and Algorithms. In *Proceedings of the First International Workshop on Real-Time Databases*, pages 2–7, March 1996.

[48] P.S. Yu, K-L. Wu, K-J. Lin, and S.H. Son. On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE, Special Issue on Real-Time Systems*, 82(1):140–157, 1994.