Backlog Estimation and Management for Real-Time Data Services[*]

Kyoung-Don Kang, Jisu Oh, Yan Zhou
Department of Computer Science
State University of New York at Binghamton
{*kang,joh,yzhou*}*@cs.binghamton.edu*

Abstract

Real-time data services can benefit data-intensive real-time applications, e.g., e-commerce, via timely transaction processing using fresh data, e.g., the current stock prices. To enhance the real-time data service quality, we present several novel techniques for (1) database backlog estimation, (2) fine-grained closed-loop admission control based on the backlog model, and (3) hint-based incoming load smoothing. Our backlog estimation and feedback control aim to support the desired service delay bound without degrading the data freshness critical for real-time data services. Workload smoothing, under overload, help the database admit and process more transactions in a timely manner by probabilistically reducing the burstiness of incoming data service requests. In terms of the data service delay and throughput, our feedback-based admission control and probabilistic load smoothing considerably outperform the baselines, which represent the current state of the art, in the experiments performed in a stock trading database testbed.

1 Introduction

Real-time data services can benefit data-intensive real-time applications such as e-commerce, agile manufacturing, and traffic control by supporting timely transaction processing using fresh data, reflecting the current market or traffic status. If the service delay is longer than a few seconds, most users may leave [3]. Transaction processing based on stale data may adversely affect decision making. Without RTDB (real-time database) support, timing and freshness constraints as well as the ACID (atomicity, consistency, integrity, and durability) properties of transactions such as stock trades have to be supported manually.

[*]This work was supported, in part, by a NSF grant CNS-0614771.

Supporting the desired timeliness and freshness is challenging, because database workloads may vary significantly, for example, due to the varying market or traffic status. To enhance the quality of *soft real-time* data services, in this paper, we present several novel techniques: (1) a database backlog estimation technique, (2) a fine-grained feedback-based admission control scheme designed based on the relation between the estimated backlog and service delay, and (3) a hint-based scheme for smoothing incoming workloads. For real-time data services, it is important for a database to be able to estimate the current data service workload and its impact on performance. Especially, to quantify the workload, we define the notion of the database backlog that indicates the *amount of data for the database to process.* Due to potential data/resource contention, it is hard to precisely analyze the backlog. Thus, we estimate the backlog, if any. More specifically, our backlog estimator predicts the amount of data for the database server to actually process by looking up the data service requests in the queue, using the meta data extracted from the database schema and transaction semantics for workload estimation.

Backlog estimation could be optimistic, since a transaction expected to access a certain number of data may get aborted and restarted due to data conflicts. To address this problem, we apply control theoretic techniques [6] by modeling database dynamics in terms of the *relation between the backlog and data service delay.* Intuitively, if there are more data to process, the service delay increases and vice versa. For data-intensive applications, this is a natural way to estimate backlogs. We statistically identify the relationship between the backlog and delay via system identification [6], since a database and its workloads are too complex to deterministically model. This model is used for feedback-based admission control in this paper. The controller computes how much backlog, i.e., the amount of data to process, needs to be reduced when the data service delay exceeds the desired delay bound, e.g., $2s$, and vice versa. The desired average and transient service delays are specified as the service level agreement (SLA) between the database and clients. Feedback control has previously been applied to manage the RTDB performance [2, 10]; however, most work on feedback control of RTDB performance is based on simulations. It is applied to manage a real database performance [8]; however, only a gross-grained control model, which is based on the relation between the *queue length* and service delay, is employed in [8]. Notably, this model cannot closely capture database dynamics when the transaction size−the amount of data for a transaction to process−substantially varies from transaction to transaction. Compared to them, our approach is based on a novel fine-grained model for feedback control in a real database. Via database-specific backlog estimation and feedback-based admission control, we can support the desired data service delay without degrading the data freshness unlike most existing work on feedback control of the RTDB performance including [2, 10, 8].

In addition, we provide an optional *load smoothing knob* to clients. If clients accept the smoothing option through the SLA, individual clients *probabilistically reduce the service request rate* under overload by small predefined values included in the SLA. The basic idea is that, under overload, an individual client may accept a relatively small additional delay for load smoothing rather than getting rejected by the admission controller. For example, e-commerce clients may accept $0.1s$ delay due to load smoothing predefined in the SLA to increase the chance for their requests to get admitted when the database delay exceeds a few seconds due to overloads. Especially, individual clients proactively take action rather than waiting until the feedback controller computes a new admission control signal at the next sampling instant.[1]

From the system's perspective, flash aggregate workloads can be considerably relieved, if a fraction of clients delay submitting their requests by even a small amount of time. We take a simple yet effective approach to load smoothing. The admission controller keeps track of the *request rejection rate* and the database piggybacks the rate to the responses to service requests to minimize the communication overhead. Given the hint, i.e., the current rejection rate, a client delays its next request submission with the probability proportional to the rejection rate. As a result, not only resource but also data contention can be reduced due to the reduced burstiness of the incoming request traffic. The feedback-based admission controller can in turn accept more incoming data service requests. Very little prior work has considered request traffic smoothing for real-time data services. Further, little work has been done to apply both proactive and feedback control approaches to real-time data services.

For performance comparisons, we have conducted extensive experiments in a stock trading database testbed. In our experiments, feedback-based admission control and probabilistic traffic shaping closely support the desired average and transient service delay for bursty workloads. Also, they significantly outperform the tested baselines, which represent the current state of the art, in terms of the average/transient service delay and database throughput. More specifically, our feedback-based admission control reduces the average delay by 109%, 75%, 61% compared to the underlying Berkeley DB [4]−an open source database from Oracle−and the other two baselines, respectively. The traffic shaping scheme integrated with the feedback-based admission controller enhances the average delay by 156%, 115%, and 98% compared to the same baselines. Our approaches increase the number of the data processed by timely transactions, which finish within the desired delay bound, by $14\% - 53\%$ compared to the baselines. Moreover, our approaches support reliable transient service delays in the presence of dynamic workloads, whereas the baselines show largely fluctuating transient delays. Notably, our feedback control and traffic smoothing only take approximately 1.5% CPU utilization and less than 5KB of memory.

The remainder of this paper is organized as follows. The structure of our database and service requirements are described in Section 2. A description of backlog estimation and traffic smoothing is given in Section 3. Our feedback control scheme is discussed in Section 4. Performance evaluation results are described in Section 5. Section 6 discusses related work. Finally, Section 7 concludes the paper and discusses future work.

2 Real-Time Data Service Structure

Figure 1 shows the architecture of our database system Chronos built on top of Berkeley DB [4]. It consists of the database backlog estimator, admission controller, traffic smoother, feedback controller, and a database server. The database server processes data service requests and periodically updates stock prices received from the stock quote server to support the freshness of stock prices. Chronos periodically updates 3000 stock prices. A fixed update period selected in a range $[0.2s, 5s]$ is associated with each stock price. The database server schedules accepted requests in a

[1]Note that the delay due to voluntary load smoothing via clients is included to measure the service delay in our system design and performance evaluation. Thus, comparisons of data service delays between our approach and the tested baselines in Section 5 are fair.

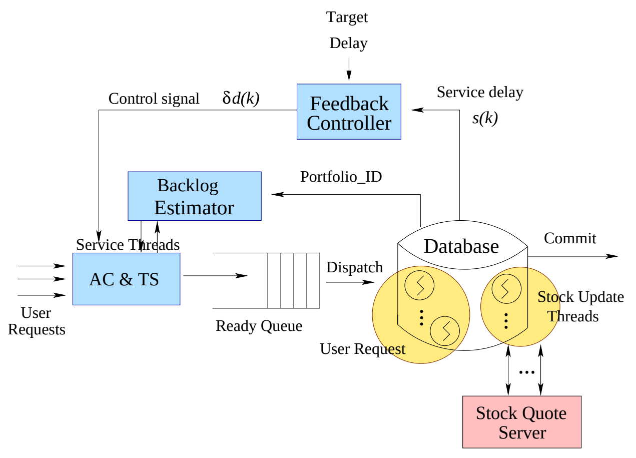

Figure 1. Chronos Real-Time Database

Table 1. Desired Performance

Notation	Description	Desired Value
S_t	Target Service Delay	$2s$
S_v	Service Delay Overshoot	$2.5s$
T_v	Settling Time	$10s$

FCFS manner, while applying 2PL (two phase locking) for concurrency control. As most databases support FCFS and 2PL, our approach is easy to deploy.

A SLA can be specified by an administrator of a real-time data service application, such as e-commerce, and agreed with clients. An exemplar SLA considered in this paper consists of the desired data freshness, average/transient service delay, and a predefined range of delays for traffic smoothing (discussed in Section 3). Data freshness, i.e., data temporal validity, is supported by periodically updating temporal data such as stock prices [20]. If the rejection rate is greater than zero, a client probabilistically picks a certain amount of a delay for smoothing in the range predefined in the SLA. Thus, the SLA can be satisfied by supporting the desired average/transient delay specified in Table 1.

As described in Table 1, the average delay needs to be shorter than or equal to $S_t = 2s$. This is realistic, because, for example, E*Trade (www.etrade.com) guarantees $2s$ response time for a specific subset of orders. We also specify transient performance requirements. An overshoot S_v, if any, is a transient service delay longer than S_t. It is desired that $S_v \le 2.5s$ and S_v decays within the settling time $T_v = 10s$. Given bursty workloads in Section 5, our feedback-based admission control scheme closely supports the desired average/transient delay due to the systematic backlog estimation and controller design performed in Sections 3

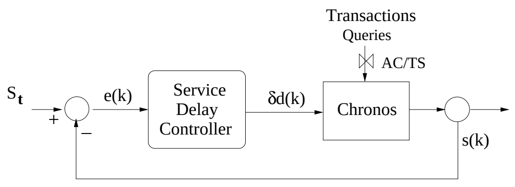

Figure 2. Data Service Delay Control Loop

and 4, while load smoothing further improves the delay and throughput. For feedback control, we set the sampling period $P = 1s$ during which several hundreds to thousands of transactions arrive at the database in our testbed.

In this paper, the service delay s_i of the i^{th} data service request is the sum of the TCP connection delay c_i, queuing delay q_i in the ready queue in Figure 1, and processing delay e_i inside the database. The $k^{th}(\ge 1)$ sampling period is the time interval $[(k-1)P, kP)$. The k^{th} sampling instant is equal to kP. Let $n(k)$ be the number of the data service requests, i.e., transactions and queries, processed in the k^{th} sampling period. Our feedback-based admission controller works as follows:

- At the k^{th} sampling instant, the feedback controller in Figure 2 (discussed in Section 4) computes the service delay $s(k) = \sum_{i=1}^{n(k)} s_i/n(k)$ and delay error $e(k) = S_t - s(k)$. Based on $e(k)$, it computes the required backlog bound adjustment $\delta d(k)$.

- Given $\delta d(k)$, the admission controller updates the database backlog bound to be used in the $(k+1)^{th}$ sampling period:

$$d_t(k) = d_t(k-1) + \delta d(k) \qquad (1)$$

  where the initial control signal $d_t(0)$ is a small positive integer when the system starts. To manage the backlog bound, we also apply the anti-windup technique [23]. If $d_t(k) < 0$ we set $d_t(k) = 0$. To avoid excessive backlogs, we also make $d_t(k) = \textit{max-data}$ if $d_t(k) > \textit{max-data}$ where $\textit{max-data} = \textit{max-queue-size} \cdot \textit{max-transaction-size}$ defined in terms of the amount of data for one transaction to process.

- In the $(k+1)^{th}$ sampling period, the database accepts a newly incoming transaction as long as the estimated backlog in the ready queue in Figure 1 does not exceed $d_t(k)$ after accepting the new request. Otherwise, the database drops requests at the tail of the queue and returns busy messages to the corresponding clients.

During the k^{th} sampling period, the database returns the current rejection rate computed by the admission controller to the clients together with data service results or busy messages. Given the information, clients proactively reduce the workload burstiness before the feedback controller computes the next control signal at the k^{th} sampling instant.

3 Backlog Estimation and Smoothing

This section describes how to estimate the database backlog by examining the ready queue in Figure 1 and smooth the incoming request traffic under overload.

3.1 Estimating Database Backlog

Chronos provides four types of transactions: view-stock, view-portfolio, purchase, and sale for seven tables [9]. Our database schema and transactions are similar to TPC-W [22], which models a small number of tables and well-defined queries and transactions such as catalog browsing, purchase, and sale in an emulated online bookstore. Further, we add periodic stock price updates for data freshness management critical for real-time data services. For backlog estimation, we leverage our stock trade database schema and semantics of transactions/queries, which largely affect the data access pattern and amount of data to process. Especially, we focus on a subset of the tables directly related to the database backlog estimation. Our approach is generally applicable to database workload estimation, since the information about the schema and transaction types in a database application is usually available. For a transaction (or query) T_i from a client, we estimate the amount of data to access as follows.

- **view-stock**: This query is about a set of companies' information and their associated stock prices. To process this request, Chronos needs to access STOCKS and QUOTES tables that hold <stock symbol, full company name, company ID> and <company ID, current stock price> for each company. After parsing the query, Chronos finds the number of companies n_c specified in the query. Chronos then calculates the amount of data to access for T_i: $n_i = n_c \cdot \{r(\text{STOCKS}) + r(\text{QUOTES})\}$ where $r(x)$ is the average size of a row (i.e., the average number of bytes in a row) in table x.

- **view-portfolio**: In this query, a client wants to see certain stock prices in its portfolio. For each stock item in the portfolio, Chronos needs to look up the PORTFOLIOS table that holds <client ID, company ID, purchase price, shares> to find the company IDs used to look up the QUOTES table. Thus, the estimated amount of data to access for this query is: $n_i = |portfolio(id)| \cdot \{r(\text{PORTFOLIOS}) + r(\text{QUOTES})\}$ where $|portfolio(id)|$ is the number of stock items in the portfolio owned by the client whose ID is id.

- **purchase**: When a client places a purchase order for a stock item, Chronos first gets the current stock price from the QUOTES table. If the purchased stock was not in the portfolio before the purchase, the stock item and its purchase price are added to the portfolio. If it is already in the portfolio, Chronos updates the corresponding shares. Hence, the estimated amount of data accesses for a PURCHASE transaction is: $n_i = r(\text{QUOTES}) + (|portfolio(id)| + 1) \cdot r(\text{PORTFOLIOS})$.

- **sale**: A client can sell a subset of its stocks in the portfolio via a sale transaction. In our current implementation, a client simply specifies the number of stocks $n_{i,sell}$ that it wants to sell where $n_{i,sell} \leq |portfolio(id)|$. To process a sale transaction, Chronos scans the PORTFOLIOS table to find the stock items belonging to this client's portfolio. Using the stock IDs found in the PORTFOLIOS table, Chronos searches the QUOTES table to find the corresponding stock prices. After that, Chronos updates the client's portfolio in the PORTFOLIOS table to indicate the sale. Thus, the estimated amount of data to process for this transaction is: $n_i = |portfolio(id)| \cdot r(\text{PORTFOLIOS}) + n_{sell} \cdot r(\text{QUOTES}) + n_{sell} \cdot r(\text{PORTFOLIOS})$.

The database backlog at time t is defined in a fine-grained way:

$$d(t) = \sum_{i=1}^{q(t)} n_i \tag{2}$$

where $q(t)$ is the number of the requests in the ready queue at time t. Suppose a transaction T_i arrives at time t during the $(k+1)^{th}$ sampling period; that is, $kP \leq t < (k+1)P$. T_i is admitted if $d(t) + n_i \leq d_t(k)$ where $d_t(k)$ is computed in Eq 1. For database backlog estimation, the transaction size is measured in terms of the number of data accesses, e.g., the number of the stock prices to read or the number of portfolio updates, needed to process the transaction. This approach is similar to techniques for evaluating database operators [19]. We take this approach, because our stock trading testbed mainly handles textual data whose size does not significantly vary from row to row (or table to table). Thus, in this paper, the row size is considered uniform. For different types of data such as images, backlog estimation needs to consider the size of individual data items in addition to the number of data to access. In that sense, our backlog estimation method described in this section is general purpose and stock trading is its specific instance.

Two kinds of meta data are used for estimation: system statistics and materialized portfolio information. System statistics are used to obtain general information of every table such as the number of the rows, row sizes, and number of distinct keys in the table. We periodically update the statistics at every sampling period. For more accurate backlog estimation, we keep track of the number of the stock items in each user's portfolio via view materialization. A view is a virtual table that can be materialized to quickly answer user queries with no database look up [19]. A database can materialize a view by extracting related data from the underlying tables consisting the view. In this paper, portfolio information is materialized for backlog estimation; that is, we pre-count the number of stock items in each user's portfolio and incrementally update the number whenever it is changed. By doing it, we can maintain the consistency between the meta data and the underlying table in the database. When a client wants to look up its portfolio, our backlog estimator can immediately find how many stocks need to be quoted to process this request using the meta data.

Overall, our approach is lightweight. We have measured the CPU consumption through the `/proc` interface in Linux. Our backlog estimation procedure described in this section only consumes approximately 0.3% CPU utilization and our feedback control scheme consumes less than 1.2% CPU utilization. Further, they all together consume less than 5KB of memory mainly to store the meta data needed to estimate database backlogs.

3.2 Request Traffic Smoothing

In our approach, traffic smoothing intends to increase the chance for an individual request to get admitted to the system under overload. The admission controller continuously updates the rejection rate $p(k)$ ($0 \leq p(k) \leq 1$) for transactions arriving in the k^{th} sampling period. The database returns the hint $p(k)$ to the clients as discussed before. Given $p(k)$, a client delays submitting its next service request, if any, with probability $p(k)$. Specifically, a client picks a uniform random number x in [0,1]. If $x < p(k)$, it delays the next request by the time uniformly selected in a predefined range $[t_a, t_b]$. For analysis, let us assume that the inter-request time between two consecutive data service requests from an arbitrary client uniformly ranges in an interval $[t_1, t_2]$; that is, a client submits $1/0.5(t_1 + t_2)$ requests/s in average. If $p(k) > 0$, a request is delayed by the mean time of $0.5(t_a + t_b)$ with probability $p(k)$ due to smoothing. Thus, the client submits $1/0.5(t_1 + t_2 + p(k)(t_a + t_b))$ requests/s in average. If m clients concurrently submit data service requests, the expected mean reduction of the total arrival rate via smoothing is:

$$E[\gamma] = \frac{2mp(k)(t_a + t_b)}{(t_1 + t_2)[t_1 + t_2 + p(k)(t_a + t_b)]} \text{requests/s.} \tag{3}$$

In the SLA considered in this paper, we set $t_a = 0.1s$ and $t_b = 0.3s$. Hence, under overload, a client's request suffers maximum $0.3s$ extra delay to increase its chance for admission via smoothing. For example, suppose that $t_1 = 0.1s$, $t_2 = 0.5s$, $t_a = 0.1s$, and $t_b = 0.3s$. From Eq 3, it can be computed that, for arbitrary m, the total arrival rate is expected to be reduced by 40% when the rejection rate $p(k) = 1$, while it is expected to be reduced by 25% when $p(k) = 0.5$. This approach has several advantages:

- If $m \gg t_a, t_b, t_1$, and t_2 in Eq 3, our approach can considerably reduce the total arrival rate when $p(k) > 0$. When overloaded, many clients may submit service requests with short inter-request times. Thus, $E[\gamma]$ in Eq 3 could be large, reducing the burstiness of workloads. As a result, the database server can accept a larger fraction of incoming requests without severely increasing the service delay for individual clients. In Section 5, we show that our smoothing scheme significantly improves the average/transient delay and throughput of real-time data services;

- Our smoothing technique can enhance the stability of the overall system. By proactively reducing the burstiness, it makes the job of feedback-based admission control easier. In Section 5, the approach integrating traffic smoothing and feedback-based admission control shows the best average/transient performance among the tested approaches; and

- This approach is distributed requiring no centralized control. It is undertaken by clients based on the hint, $p(k)$, provided by the database. It imposes minimal overheads on the database and clients.

Our traffic smoothing scheme is analogous to network traffic shaping [12]. However, network traffic shaping is implemented by a router, while our approach is voluntarily undertaken by clients based on the hint about the database status. As a result, the burstiness of workloads can be reduced under overload, enhancing the quality of real-time data services in a cost-effective way. Our smoothing approach is driven by the behavior of a data service application supported by Chronos. In contrast, network traffic shaping is oblivious to real-time data service semantics and, therefore, it is not directly applicable to RTDBs.

4 Service Delay Control via Backlog Adaptation

We apply feedback control to support the desired delay bound in the presence of dynamic workloads. As the backlog in Eq 2 increases, the service delay increases and vice versa. To model this relation, we express the data service delay based on the n previous service delays and database backlogs in a difference equation:

$$s(k) = \sum_{i=1}^{n} \{a_i s(k-i) + b_i d(k-i)\} \qquad (4)$$

where $n(\geq 1)$ is the system order [6]. Generally, a higher order model is more accurate but more complex. Using this difference equation, we can model RTDB dynamics considering individual transactions, potentially having different amounts of data to process. Thus, our control model is fine grained.

The unknown model coefficients a_i's and b_i's in Eq 4 are statistically derived via SYSID (system identification) [6] to minimize the sum of the squared errors of data service delay predictions based on database backlogs estimated using Eq 2. As SYSID aims to identify the behavior of the controlled database system, Chronos accepts all incoming data service requests. Neither feedback-based admission control nor traffic smoothing is applied during SYSID. Stock prices are periodically updated as discussed before.

In the current version of our testbed, 1200 client threads concurrently send data service requests to the database for one hour. Each client sends a service request and waits for the response from the database server. After receiving the transaction or query processing result, it waits for an inter-request time uniformly selected in a range before sending the next data service request. For SYSID, we choose the inter-request time range [$0.5s$, $2.5s$], since the service delay shows a near linear pattern around $S_t = 2s$ in Figure 3 depicting the inter-request time sub-ranges vs. service delays. Due to our system modeling and controller tuning, the feedback-based admission controller can support the

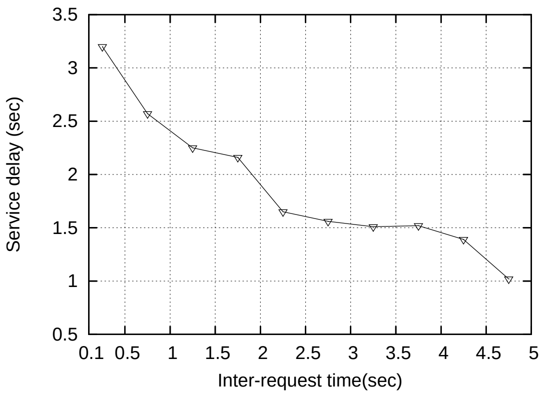

Figure 3. Inter-request time vs. service delay

desired delay bound within this operating range. Under overload, smoothing may help the system to reenter the operating range by reducing the burstiness of incoming data service requests as discussed before.

For accurate SYSID of statistical database dynamics, we use the square root values of the performance data, similar to [5]. To analyze the accuracy of SYSID, we use the R^2 metric [6]. For the second order model, $R^2 = 1-$ variance(service delay prediction error)/variance(actual service delay) = 0.86. A control model is acceptable if its $R^2 \geq 0.8$ [6]. We have found that the first order model is not accurate enough. The third and fourth order models show almost the same R^2. Thus, to model the relation between the database backlog and delay, we favor the second order model, which is less complicated:

$$\begin{aligned} s(k) = &-0.003s(k-1) - 0.104s(k-2) + \\ &0.015d(k-1) + 0.006d(k-2). \end{aligned} \qquad (5)$$

We derive the transfer function of the open-loop database by taking the z-transform [6] of Eq 5 to algebraically represent the relationship between the database backlog and service delay in the frequency domain:

$$\tau(z) = \frac{S(z)}{D(z)} = \frac{0.015z + 0.006}{z^2 + 0.003z + 0.104} \qquad (6)$$

where $S(z)$ is the z-transform of $s(k)$ and $D(z)$ is the z-transform of $d(k)$ in Eq 5.

To support the average and transient performance in Table 1, we apply an efficient PI (proportional and integral) control law, which can support the stability via I control in addition to P control. We do not use a derivative controller sensitive to noise. At the k^{th} sampling instant, the PI controller computes the control signal $\delta d(k)$, i.e., the database backlog adjustment needed to support S_t:

$$\delta d(k) = \delta d(k-1) + K_P[(K_I+1)e(k) - e(k-1)] \quad (7)$$

where the error $e(k) = S_t - s(k)$ at the k^{th} sampling instant as shown in Figure 2. The z-transform of Eq 7 is:

$$\psi(z) = \frac{\Delta D(z)}{E(z)} = \frac{K_P(K_I+1)[z-1/(K_I+1)]}{z-1} \quad (8)$$

where $\Delta D(z)$ and $E(z)$ are the z-transform of $\delta d(k)$ and $e(k)$, respectively. Given the open loop transfer function in Eq 6 and the transfer function of the PI controller in Eq 8, the transfer function of the closed loop in Figure 2 is: $\eta(z) = \frac{\tau(z)\psi(z)}{1+\tau(z)\psi(z)}$ [6]. We locate the closed loop poles$-$the roots of the denominator of $\eta(z)-$inside the unit circle to support the stability and performance requirements specified in Table 1. The selected closed loop poles are -0.308 and $0.52 \pm 0.106i$. The corresponding $K_P = 2.77$ and $K_I = 5.28$. $\delta d(k)$ computed in the closed-loop is used to compute the backlog bound in Eq 1.

5 Performance Evaluation

In this section, we evaluate our approach and several baselines to observe whether or not they can support the desired performance in Table 1. A description of experimental settings is followed by the average and transient performance results.

5.1 Experimental Settings

A Chronos server runs in a laptop with the 1.66 GHz dual core CPU and 1 GB memory. Clients run in a desktop PC that has the 3 GHz CPU and 2GB memory. In total, 1200 client threads continuously send service requests to the Chronos server. The stock quote server runs in a desktop PC that has the same CPU and 4 GB memory. Every machine runs the 2.6.15 Linux kernel. The clients, stock quote server, and database sever are connected through a 1 Gbps Ethernet switch using the TCP protocol.

For 60% of time, a client thread issues a query about stock prices, because a large fraction of requests can be stock quotes in stock trading. For the remaining 40% of time, a client uniformly requests a portfolio browsing, purchase, or sale transaction at a time. One experiment runs for 900s. At the beginning of an experiment, the inter-request time is uniformly distributed in [3.5s, 4s]. At 300s, the range of the inter-request time is suddenly reduced to [0.1s, 0.5s] to model bursty workload changes; that is, the total arrival rate abruptly increases by $7-40$ times at 300s. The reduced inter-request time range is maintained until the end of an experiment at 900s. This is considerably more bursty than our previous testbed workloads [8]. Also, most existing RTDB work is not evaluated in a real database system using bursty workloads [20, 8].

Table 2. Tested Approaches

Open	Pure Berkeley DB [4]
AC	Ad-hoc admission control
FC-Q	Feedback control (FC)$-$queue size vs. delay
FC-D	FC$-$database backlog vs. delay
FC-TS	FC of database backlog & traffic smoothing

For performance comparisons, we consider the five approaches shown in Table 2:

- **Open** is the basic Berkeley DB [4] without any special performance control facility. Hence, it represents a state-of-the-art database system. For performance comparisons, all the other tested approaches are implemented atop Berkeley DB.

- **AC** applies admission control to incoming transactions in proportion to the service delay error under overload.

- **FC-Q** models the relation between the queue size and response time, similar to [8]. The controller in FC-Q computes the required *queue length* adaptation to support the desired response time. The control signal for queue size adaptation is enforced by applying admission control to incoming transactions. Since the workloads used in this paper are different from$-$more bursty than$-$the workloads used in [8], we re-tuned FC-Q's controller based on the relation between the queue length and delay to support the performance requirements described in Table 1. Open, AC, and FC-Q are the baselines to which the performance of our approaches is compared.

- **FC-D** is the feedback-based admission control scheme presented in this paper.

- **FC-TS** extends FC-D by supporting traffic smoothing in addition to closed-loop admission control under overload. For fair comparisons, we add the delay due to traffic smoothing, if any, to the service delay of FC-TS. Specifically, we set the range of the delay for traffic smoothing to [0.1s, 0.3s] as discussed in Section 3.2

Each performance data is the average of 10 runs using different random seeds. 90% confidence intervals are also derived. We show the performance results observed between $100s$ and $900s$ to exclude the database initialization phase, involving initial housekeeping chores such as database schema and data structure initialization.

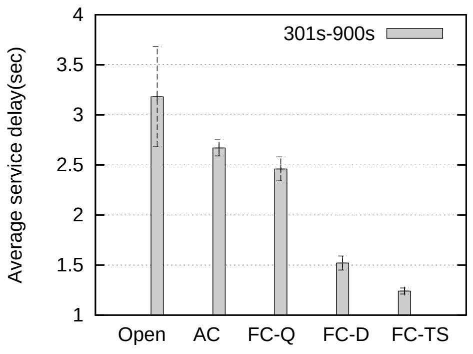

Figure 4. Average service delay

5.2 Average Data Service Delay

Figure 4 shows the average service delay for each tested approach. In the interval [$100s$, $300s$], all the tested approaches can support the desired delay bound $S_t = 2s$ (Table 1) due to the relatively low workload. However, only FC-D and FC-TS can support the delay bound in [$301s$, $900s$] as shown in the figure. FC-D achieves the $1.52 \pm 0.07s$ average service delay via closed-loop admission control based on the database backlog model. FC-TS further reduces the average delay to $1.24 \pm 0.03s$ via traffic smoothing in addition to feedback control. Open fails to support S_t by accepting all incoming requests, causing system overloads. AC fails to support S_t due to its ad hoc admission control. From this, we observe that systematic feedback control is necessary. The average delay of FC-Q is shorter than Open and AC. However, FC-Q's control model is only based on the queue length vs. delay, which is coarse grained. Thus, it fails to support the delay bound under overload. FC-Q was only able to support the delay bound for dynamic workloads, when the data freshness was degraded too [8].

5.3 Transient Service Delay

Figure 5 plots the transient service delay measured at every $1s$ sampling period. FC-D and FC-TS do not suffer any delay overshoot in the [$100s$, $300s$] interval. In contrast, Open, AC, and FC-Q suffer overshoots even in this range. This implies that the database can be transiently overloaded due to dynamic data/resource contention for even moderate workloads. Hence, fine-grained, database-specific workload estimation, feedback control, and traffic smoothing are desirable. In Figure 5, after the bursty workload increase at $300s$, FC-D significantly outperforms the baselines in terms of the transient service delay. FC-D supports S_t for most of the time despite bursty workloads. There are a few overshoots after $300s$; however, most of them are smaller than $S_v = 2.5s$ and they decay within the desired settling time $T_v = 10s$ specified in Table 1. The largest overshoot is $4.13s$ at $542s$, but it decay in $3s$. FC-TS supports even better transient performance: Its highest delay overshoot is $2.88s$ at $620s$ and it decays in only $3s$. FC-Q performs better than AC and Open, but it largely violates the desired transient performance

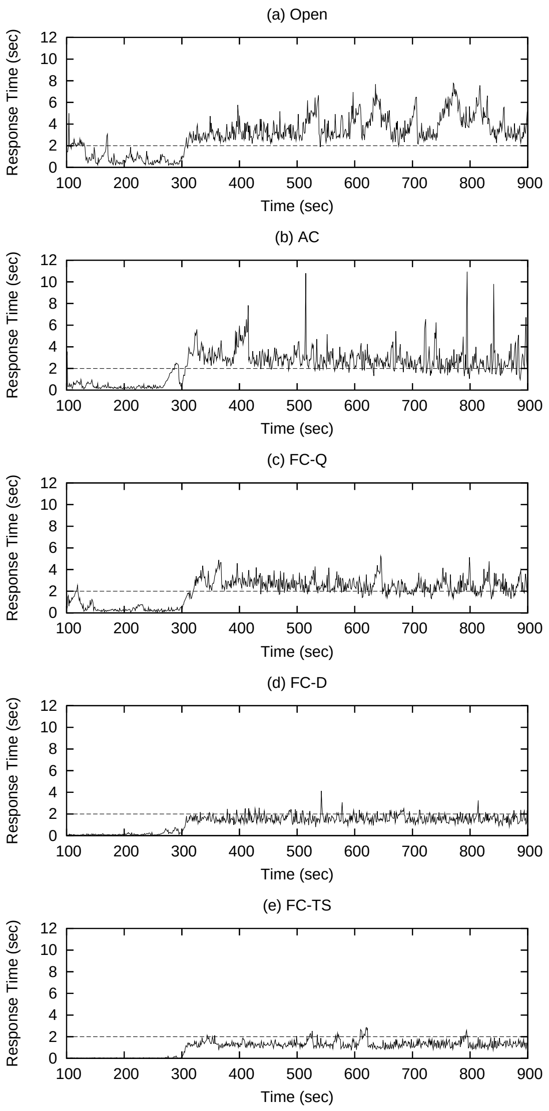

Figure 5. Transient service delay

specified in Table 1. Moreover, FC-D and FC-TS process much more data than the baselines do as discussed next.

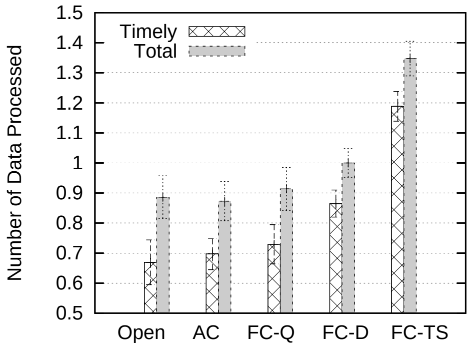

Figure 6. Normalized average data accesses

5.4 Data Service Throughput

Figure 6 shows the average number of the data processed by the committed transactions, which is *normalized* to the corresponding number for FC-D. There are two bars for each tested approach. The right bar is the total number of data processed during $900s$, while the left bar indicates the number of data processed by the timely transactions completed within S_t. We call it the number of *timely* data. As shown in the figure, FC-D and FC-TS increase the total number of processed data and the number of timely data by approximately $9\% - 48\%$ and $14\% - 53\%$ compared to FC-Q, AC, and Open.

FC-D has processed more than 18,680,000 data accesses in $900s$. Out of the total number of the data processed by the committed transactions, more than 16,150,000 data accesses are served by the transactions committed within S_t, which is approximately 86% of the total data processed. By managing the database performance based on the amount of data to process, FC-D effectively avoids excessive service delays. As a result, it processes more data in a timely fashion than the baseline approaches. By applying smoothing in addition to feedback-based admission control, FC-TS processes 25,175,939 data in $900s$. Approximately 88% of them are processed in a timely manner.

We have also observed that our approach significantly improves the transient database throughput compared to the baselines. However, we do not include the performance evaluation results due to space limitations. In summary, FC-D closely supports the desired delay specified in Table 1, while considerably enhancing the throughput. FC-TS further improves the average and transient service delay. Its throughput is significantly higher than the other approaches. Since the burstiness of workloads and resulting data/resource contention can be considerably relieved by traffic smoothing, more transactions can be admitted and processed within the desired delay bound. On the other hand, individual clients do not suffer excessive extra delays due to traffic smoothing, because the delay due to traffic smoothing, if any, is bounded by the predefined maximum as discussed in Section 3. Overall, our database-specific backlog estimation and service quality management schemes are very effective.

6 Related Work

Feedback control has recently been applied to RTDB performance management; however, most these approaches including [2, 10] are based on simulations. Also, their control models are not specific to databases but similar to [15]. Unlike prior work on feedback control of RTDB performance, our previous work [8] applies feedback control theory to manage the performance of a real database system. In this paper, we present a new feedback control model based on the database backlog vs. delay relation as well as a workload smoothing scheme to support the desired delay bound without degrading the data freshness. In general, most RTDB work−not limited to the work on feedback control of RTDB performance−is based on simulations. There have been several RTDB systems [1, 11, 18, 13]; however, they are either proprietary or discontinued. Thus, our work can provide valuable insights to system dynamics that can be leveraged for further RTDB research.

In [17], feedback control is applied to manage the database throughput by reducing background utility work in a database, e.g., database backup or restore, under overload. Schroeder et. al. [21] determine an appropriate number of concurrent transactions to execute in a database by applying queuing and control theoretic techniques. However, no formal, detailed discussion of the control model is given. Neither is the stability analyzed. Thus, we cannot replicate their approach for comparisons. Unfortunately, none of these approaches considers to support the desired data service delay, which is critical for real-time data services.

Feedback control has been applied to real-time scheduling [15], a real-time middleware[16], and a web server[14]. However, they do not consider database-specific issues such as data service backlogs. In addition to feedback, feed-forward has been applied to support the desired performance in a web server [7] and real-time middleware [16]. Analogously, we predict the estimated database backlog before the database actually processes the submitted transactions using meta data, while adapting the burstiness of workloads, if necessary, to reduce the rejection rate before the next feedback control signal becomes available. Little prior work has been done to apply both feedback control and feed-forward techniques to real-time data services.

7 Conclusions and Future Work

In this paper, we present several novel techniques for (1) database backlog estimation, (2) closed-loop admission control based on the estimated backlog vs. service delay relation, and (3) hint-based load smoothing to enhance the quality of real-time data services. In a stock trading testbed, our feedback control and traffic smoothing schemes closely support the desired average/transient data service delay. They substantially outperform the baselines representing the current state of the art in terms of the service delay and database throughput. In the future, we will continue to enhance feedback control, workload adaptation, and transaction scheduling schemes.

References

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying Update Streams in a Soft Real-Time Database System. In *ACM SIGMOD*, 1995.

[2] M. Amirijoo, J. Hansson, and S. H. Son. Specification and Management of QoS in Real-Time Databases Supporting Imprecise Computations. *IEEE Transactions on Computers*, 55(3):304–319, 2006.

[3] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-Perceived Quality into Web Server Design. In *9th International World Wide Web Conference*, 2000.

[4] Oracle Berkeley DB Product Family, High Performance, Embeddable Database Engines. Available at http://www.oracle.com/database/berkeley-db/index.html.

[5] N. R. Draper and H. Smith. *Applied Regression Analysis*. Wiley, 1968.

[6] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. A John Wiley and Sons, Inc., Publication, 2004.

[7] D. Henriksson, Y. Lu, and T. Abdelzaher. Improved Prediction for Web Server Delay Control. In *Euromicro Conference on Real-Time Systems*, 2004.

[8] K. D. Kang, J. Oh, and S. H. Son. Chronos: Feedback Control of a Real Database System Performance. In *the 28th IEEE Real-Time Systems Symposium*, 2007.

[9] K. D. Kang, P. H. Sin, J. Oh, and S. H. Son. A Real-Time Database Testbed and Performance Evaluation. In *the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[10] K. D. Kang, S. H. Son, and J. A. Stankovic. Managing Deadline Miss Ratio and Sensor Data Freshness in Real-Time Databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1200–1216, 2004.

[11] S. Kim, S. H. Son, and J. A. Stankovic. Performance Evaluation on a Real-Time Database. In *IEEE Real-Time Technology and Applications Symposium*, 2002.

[12] J. F. Kurose and K. W. Ross. *Computer Networking*. Addison Wesley, 2007.

[13] Lockheed Martin. EagleSpeed Real-Time Database Manager.

[14] C. Lu, Y. Lu, T. Abdelzaher, J. A. Stankovic, and S. Son. Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, September 2006.

[15] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 23(1/2), May 2002.

[16] C. Lu, X. Wang, and X. Koutsoukos. End-to-End Utilization Control in Distributed Real-Time Systems. In *International Conference on Distributed Computing Systems*, 2004.

[17] S. Parekh, K. Rose, Y. Diao, V. Chang, J. Hellerstein, S. Lightstone, and M. Huras. Throttling Utilities in the IBM DB2 Universal Database Server. In *Proceedings of the 2004 American Control Conference*, volume 3, pages 1986–1991, June 2004.

[18] C.-S. Peng, K.-J. Lin, and C. Boettcher. Real-Time Database Benchmark Design for Avionics Systems. In *the First International Workshop on Real-Time Databases: Issues and Applications*, 1996.

[19] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2003.

[20] K. Ramamritham, S. H. Son, and L. C. Dipippo. Real-Time Databases and Data Services. In *Real-Time Systems*, volume 28, Nov.-Dec. 2004.

[21] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *Proceedings of the 22nd International Conference on Data Engineering*, page 60, 2006.

[22] Transaction processing performance council. http://www.tpc.org/.

[23] D. Vrancic. *Design of Anti-Windup and Bumpless Transfer Protection*. PhD thesis, University of Ljubljana, Slovenia, 1997.