Database Support for Efficiently Maintaining Derived Data *

Brad Adelberg† Ben Kao‡ Hector Garcia-Molina§

Abstract

Derived data is maintained in a database system to correlate and summarize base data which record real world facts. As base data changes, derived data needs to be recomputed. A high performance system should execute all these updates and recomputations in a timely fashion so that the data remains fresh and useful, while at the same time executing user transactions quickly. This paper studies the intricate balance between recomputing derived data and transaction execution. Our focus is on efficient *recomputation strategies* — how and when recomputations should be done to reduce their cost without jeopardizing data timeliness. We propose the *Forced Delay* recomputation algorithm and show how it can exploit *update locality* to improve both data freshness and transaction response time.

Keywords: derived data, view maintenance, active database system, transaction scheduling, update locality.

1 Introduction

Active rule-based systems are often employed in dynamic environments to monitor the status of real-world objects and to discover the occurrences of "interesting" events. For instance, a military radar system can track aircraft and signal alerts when a dangerous pattern appears. A program trading application, for example, monitors the prices of stocks and other commodities, looking for good opportunities. Figure 1 illustrates the major components of such systems. The dynamic environment is modeled by a set of *base data* items stored within a database system. The environment is monitored (e.g., through sensors or humans reporting information); any changes are captured by a stream of *updates* to the base data.

In addition to the base data, the system very often contains *derived composite data.* This is data that indirectly reflects the state of the outside environment, and can be computed from the base data. For example, the S&P 500 stock index represents the aggregate price of 500 U.S. stocks (the base data in this case). In a robot arm control application, readings from sensors (base data) may be used to estimate the weight of the object being lifted by the arm. We say that the composite data (such as a financial composite index, the estimated weight of the object held by a robot arm) is *derived* from the base data (such as a stock price, a sensor reading) collected directly from the

*This work was supported by the Telecommunications Center at Stanford University, by Hewlett Packard and by Philips. This work was also supported by an equipment grant from Digital Equipment Corporation.

†Stanford University Department of Computer Science. e-mail: adelberg@cs.stanford.edu

‡Princeton University Department of Computer Science. Current address: Department of Computer Science, Stanford University, Stanford, CA 94305. e-mail: kao@cs.stanford.edu

§Stanford University Department of Computer Science. e-mail: hector@cs.stanford.edu

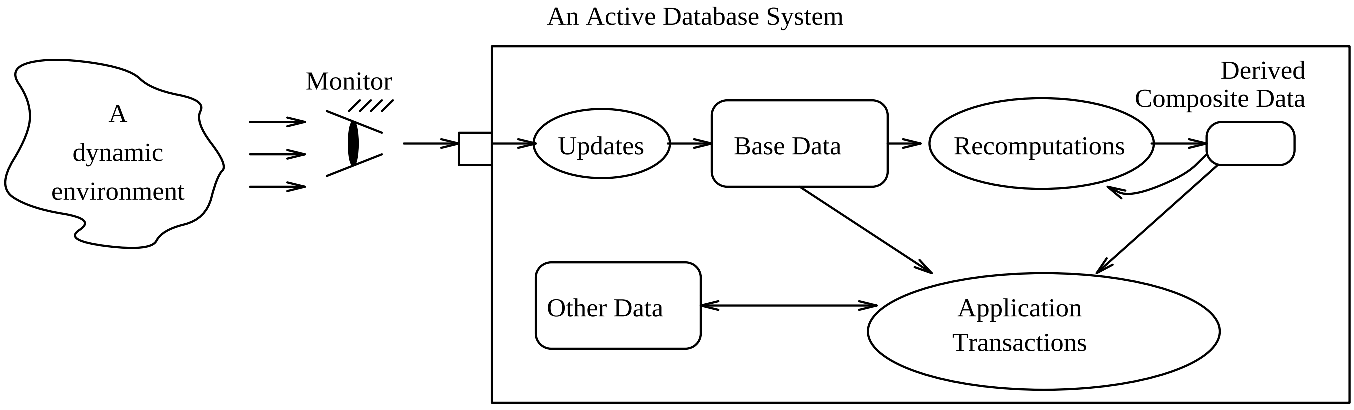

Figure 1: A high level model for the derived data maintenance problem.

environment. The computations for derived data are expressed as *rules* that specify how changes to the base data should be reflected on the derived data. As we will study in this paper, these rules can be triggered and executed in a variety of ways, leading to derived data with varying degrees of "up to dateness."

Both base and derived data are accessed by *application transactions* that generate the ultimate actions taken or decisions made by the system. For instance, application transactions may request the purchase of stock, move a robot arm, or initiate defensive action against enemy aircraft. Application transactions can be driven by user requests, or in other cases by triggers that identify certain conditions in the base or derived data. Notice that updates to base data or recomputations for derived data may also be run as transactions (e.g., with some of the ACID properties). In those cases, we refer to them as update transactions and recomputation transactions. When we use the term transaction alone, we are referring to an application transaction.

Application transactions can be associated with one or two types of timing requirements: transaction timeliness and data timeliness. Transaction timeliness refers to how "fast" the system responds to a transaction request, while data timeliness refers to how "fresh" the data read is, or how closely in time the data read by a transaction models the environment. Satisfying the two timeliness properties poses a major challenge to the design of the underlying database system. This is because the requirements pose conflicting demands on system resources: To keep the data fresh, updates on base data should be applied promptly. Also, whenever the value of a base data item changes, affected derived data has to recomputed accordingly. Furthermore, the computational load of applying base updates and performing recomputations can be extremely high, causing critical delays to transaction, either because there are not enough CPU cycles for them, or because they are delayed waiting for fresh data.

Some of the reasons why update and recomputation loads can be very high are as follows:

- Very large numbers of base data items may be involved. For example, there could be thousands of sensors in a power network, and there are about 300,000 financial instruments in the U.S. market alone.

- Recomputing a derived item may be an expensive operation. To compute the theoretical value of a financial option price, for example, requires computing the cumulative distribution of the standard normal function and the natural log function. In a vision application, computation may involve texture identification which involves computing intensity differences among many pixels.

- The base data update rate can be very high. For example, a financial database that keeps track of the prices of U.S. financial instruments may receive more than 500 updates per second during peak time [CB94].

- Recomputations can have high fan-in and fan-out. A derived data item may depend on a large numbers of base data items i.e., high *fan-in*. For example, the S&P 500 index is derived from a set of 500 stocks. When any one of these base data items changes, the derived data has to be updated, so the recomputation is triggered very frequently. Similarly, a base data item may be "popular" in the sense that it is used to derive a number of derived data items, i.e., high *fan-out*. For example, a pixel value is used to calculate a number of pair-wise co-occurrence probabilities; IBM stock price is being used in a number of composite indices and many other stock options. Whenever a popular item is updated, a number of derived items will have to be updated as well.

This problem of heavy update and recomputation arose in our experience with STRIP. STRIP is a main-memory resident soft real-time database system implemented at Stanford. One of the applications driving our system is program trading. In our experiments, STRIP maintains a database of stock prices and various composite indices on which trading transactions are run. The system is driven by a real-time trace of price quote updates. We found that recomputing the composite indices triggered by the updates represents a major load to the system, and indeed, trading transactions are often significantly delayed due to the heavy update and recomputation activities.

Thus, in these systems we believe that the central problem is the efficient installation of updates and the recomputations they trigger. This problem appears to be more critical than the scheduling of transactions which has traditionally received a lot of attention in the real-time database literature [Ram93]. In a previous study [AGMK95a], we have looked at the scheduling problem of updates and transactions and suggested efficient algorithms that can improve transaction timeliness without sacrificing data timeliness. In that study, our focus was on the update process (how to maintain the base data efficiently and up-to-date), and the semantics of data *staleness* (when data should be considered out-of-date). In this paper, we focus instead on the recomputation process, developing and evaluating algorithms for triggering and scheduling recomputations efficiently. One major difference between the two studies is that, as we have argued, recomputations could be much more expensive than simple updates. They may require more data accesses, more computation cycles, and in some cases, one update could trigger multiple recomputations. Recomputations thus have an even more marked effect on the two timeliness properties than updates do.

Even though recomputations consume substantial system resources, there is one feature, *locality*, in our favor. As we will see later, the recomputation strategies we propose will take advantage of this property whenever possible to significantly reduce loads and improve data timeliness. Intuitively, update locality here means that when a base item is updated, it is very likely that the same item or a *related* one will be updated soon thereafter. For example, a stock price update indicates that there is an interest in its trading. The same stock is therefore likely to be the subject of further trading activities and have its price changed again. In Section 2 we will show that the locality property is very marked in practice. Our recomputation schemes will thus attempt to delay recomputations slightly, so that several related base updates can be combined in a single recomputation. The challenge is to accomplish this without severely affecting the timeliness of the derived data.

There are a number of studies related to the derived data maintenance problem (e.g., [BLT86, BCL86,Han87,AL80,RK86,SF90]). Most of these studies assume the relational model and focus on maintaining *materialized views* expressible by relational algebra and some basic aggregate functions such as "sum" and "average". While materialized views can be considered as a special case of derived data, our study covers the more general case in which recomputing the value of a derived item can be arbitrarily complicated. Where appropriate, we will incorporate techniques from view maintenance to improve performance.

The rest of this paper is organized as follows. In the next section, we discuss the concept of update locality and give a representative example based on stock price quotes for the U.S. financial markets. Section 3 proposes a new definition of temporal correctness to quantify recomputation delays and timeliness of the recomputed data. In Section 4 we identify the general strategies used by traditional view maintenance algorithms to try to reduce maintenance costs. We then present algorithms which use these strategies but are general enough to maintain arbitrarily complex derived data. In Section 5 we define a performance model for an active database system. We drive this model with actual traces from the U.S. stock market, and the results are presented in Section 6. In Section 7, we discuss the implementation issues of the recomputation strategies.

2 Update Locality

Many applications that deal with derived data exhibit update locality: an update to a base item is quickly followed by updates to the same or related items. By related we mean that the later updates trigger the same recomputation(s) that the first one did.

Locality occurs in two forms: time and space. Updates exhibit time locality if related updates occur in bursts. Since base data models the physical environment, an update on a base data item often signals the "movement" of a modelled object. In many cases, this motion generates a burst of updates to the same base item over a certain period of time. For example, when a robot arm moves, the readings of its position sensors change gradually, from an initial value, through a series of intermediate values, to a final value. Updates from sensors would therefore arrive in bursts, each triggering the same recomputations.

As another example, a stock price update usually indicates trading activity on that stock. Further related updates are therefore likely to arrive soon. As a representative example, Figure 2 shows the inter-arrival distribution of the stock price updates of General Electric Co. on January

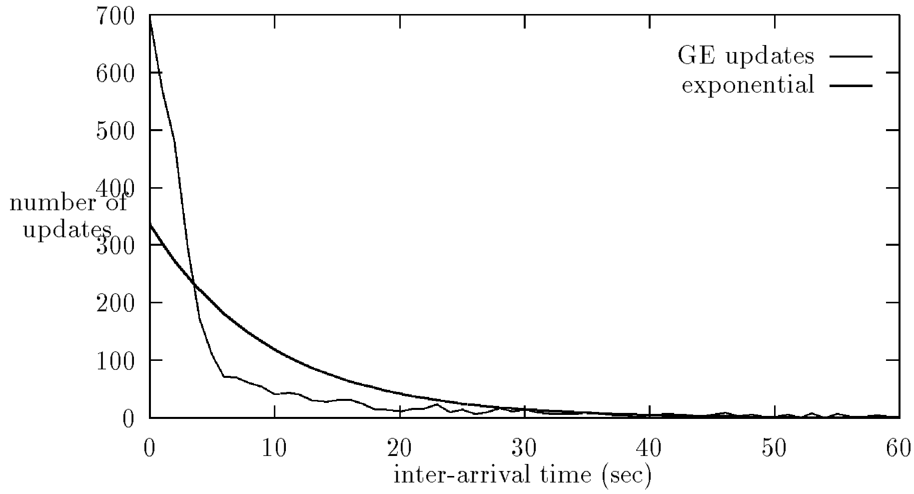

Figure 2: Inter-arrival distribution of the stock price updates of G.E. on 94/1/3.

3rd, 1994. In this trace there were 3240 updates on G.E. stock over a trading period of about 31,000 seconds. The average update arrival rate is therefore about one every 10 seconds. If there were no correlation in update arrivals, the inter-arrival time would be exponentially distributed (memoryless). This hypothetical exponential distribution (with mean 10 seconds) is also shown in Figure 2. We see that the actual distribution is much more "skewed" than the exponential one. For example, about 700 out of the 3240 updates occur within 1 second of a previous update, which is about twice as many as the exponential distribution predicts. Also, 2/3 of the updates occur within 4 seconds of a previous one. This is again twice as many as the number one would expect if there were no correlation among update arrivals. Thus, the graph clearly illustrates the time locality of the updates.

The other form of locality is space locality: when base item b, which affects derived item d, is updated, it is very likely that a related set of base items, affecting d, will be updated soon. Again, using the robot arm example, if the reading of a temperature sensor changes, it indicates that the surrounding temperature is changing. Under normal conditions, readings of other temperature sensors close to the first would change too. Each of these updates could trigger the same recomputation, say for the average room temperature. Texture discrimination in computational vision provides another example of space locality. When an object moves across a scene, the object's pixel intensities change due to the changing illumination. So when a pixel's intensity changes, it is likely that those of neighboring pixels, which are used together to compute various texture features, will change as well.

Update locality implies that recomputations for derived data occur in bursts. Recomputing the affected derived data on every single update is probably very wasteful because the same derived data will be recomputed very soon, often before any application transaction has a chance to read the derived data for any useful work. Instead of recomputing immediately, a better strategy might be to defer a recomputation by a certain amount of time and coalesce the same recomputation requests into a single computation. How to do this effectively will be studied in Section 4.

3 Recomputation and Temporal Correctness

As argued in Section 1, there is a fundamental tradeoff between performance and data correctness. Thus, before we discuss efficient schemes for recomputing derived data, we need a solid definition of what correctness means, both for its consistency and timeliness aspects. In this section we will first review existing notions for consistency and timeliness, and will argue that they are too rigid for the types of applications we are considering. Instead we will suggest a new notion that we believe succinctly captures correctness for our environment, and show how it leads to concise and intuitive metrics for both transaction and data timeliness.

ACID transactions are the traditional way of guaranteeing data consistency, in our case, guaranteeing that the derived data is consistent with the base data. To achieve consistency, recomputations for derived data are folded into the triggering update transactions. Unfortunately, running updates and recomputations as coupled transactions is not desirable in a high performance, real time environment. It makes updates run longer, blocking other transactions that need to access the same data. Indeed, [CJL91] shows that transaction response time is much improved when *events* and *actions* (in our case updates and recomputations) are decoupled into separate transactions.

Thus, we will assume that recomputations are decoupled from update transactions. This flexibility of course means that consistency can be compromised. For example, if a stock price is updated and the transaction to recompute a composite that includes the stock is delayed, a transaction that reads both the stock and composite values will see an inconsistent set. In many of the applications we are considering, application transactions can cope with such inconsistencies. Still, we believe that is desirable to have a handle on how frequently application transactions encounter inconsistencies, and the metric we propose later on provides an indirect measure of this.

Beyond consistency, data *timeliness* is the second critical correctness notion. For instance, say the system chooses to never update base or derived data even as the external world changes. The data could be completely consistent but still incorrect. Thus, we also need a measure that captures the timeliness of the data.

The real-time database community has proposed a scheme (called *temporal correctness*) for quantifying both the consistency and the timeliness of data [SL90,Ram93]. Both works consider hard real-time databases that use periodic tasks to update base data from sensors and to recompute derived data. Because these papers consider sensor data that is continuously changing, they define the timeliness of base data in terms of its *age*, the difference between the current time and the time at which it was last updated. Base data that is older than a user defined threshold is considered incorrect because presumably the value of the external variable has strayed too far from the stored value. Consistency is defined in terms of a maximum age differential in the base data used to define a derived item. In many applications, however, base data represents variables that change at discrete points in time and not continuously (this distinction is discussed in [SS87]). For example, in the program trading example stock prices are updated when trades are made, not periodically. In such a context, age has less meaning since a price quote could be old but still be correct. For this study, we consider only base data that changes at discrete points in time in response to external events, so we believe the hard real-time metrics are not appropriate.

To formulate our new notion of temporal correctness, we define the behavior of an "ideal" system. It is intended only as a yardstick by which to measure the correctness of "real" systems.

Definition 1 (instantaneous system) *An instantaneous system applies base updates and performs all necessary recomputations as soon as an update arrives, taking zero time to do it.*

Definition 2 (temporally correct system) *In a temporally correct system, an application transaction requesting at time t the value of object o (base or derived) is instantaneously given the value that o would have in an instantaneous system.*

The last definition does *not* state that in a temporally correct system data can never be stale or inconsistent. It only states that no transactions can *read* stale or inconsistent data. For base data, this definition is equivalent to the *unapplied update* staleness criteria used in [AGMK95a]: base data is temporally correct iff there are no updates that would change its value that have arrived at the database but have not been applied.

Clearly, it is very hard to build a system that is temporally correct. Instead, our goal is to quantify how much a real system deviates from the ideal, temporally correct system. A real system can deviate in two ways: by sometimes letting transactions read data not in the instantaneous system, or by delaying the completion of the read operation.

Regarding the first type of deviation, a transaction can read three types of data. Let $v_s(o,t)$ be the value of object o read by a transaction at time t in system s, where s is either I for the instantaneous system or R for a real system. At time t_1 a transaction T reads object o_i from R. The read is

correct: iff $v_R(o_i,t_1) = v_I(o_i,t_1)$,

stale: iff the read is not correct but $v_R(o_i,t_1) = v_I(o_i,t_0)$ where $t_0 < t_1$,

erroneous: iff the read is neither correct nor stale.

Figure 3 shows an example using two base data items b_1 and b_2 and a derived item d which is defined as their sum. The times t_1, t_2, t_3 are when an external variable being modelled by b_1 or b_2 changes value. Each $\times$ indicates a change in database state. For this example, both the real and ideal systems are assumed to update their base data as soon as updates arrive, so only one set of base data is shown in the figure. For the derived data, the real system experiences a delay between updates to base data and recomputation. Just before time t_1, a transaction reading d_R would get the value 5, which is identical to the value in the instantaneous database, d_I. If a transactions tries to read d (d_R in the figure) just after time t_1, however, it will still read the value 5. This is a

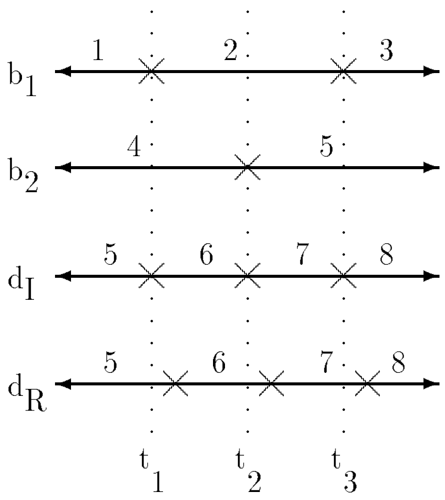

Figure 3: Derived data example

stale read since the same transaction would read the value 6 in the ideal system. Notice that in the terms of our previous discussion of correctness, if a transaction had read both d and either b_1 or b_2, we would have called this an example of inconsistency. Using instantaneity as the correctness definition, it is impossible to read inconsistent data without at least one item being stale. Thus our new definition allows us to measure both the consistency *and* timeliness of the data read by transactions in one metric: the percentage of transaction that read stale data, p_{stale}. This is the metric we will use to evaluate scheduling algorithms in Section 6. (Another useful metric would be the *expected* staleness which we could define as p_{stale} weighted by the staleness of the data read. We do not consider this metric here due to space constraints.)

The second way in which a real system may deviate from a temporally correct one is by delaying reads. In particular, notice that a system can force p_{stale} to zero by delaying transactions that attempt to read stale data until the necessary recomputations have been performed. Thus we need a second metric to quantify the delays experienced by transactions. In this paper we use the average transaction response time, s_q. (We could have used a metric that only looked at read delays but we felt it was not as intuitive.) Notice that s_q measures delays caused by both recomputation delays and excessive system loading. We believe that these two metrics, p_{stale} and s_q, succinctly capture how much a real system deviates from the ideal one. Also, both metrics are intuitive and relevant to the high level behavior that users are interested in.

We stress that p_{stale} is not a metric that can be measured on a running system. It can, however, be measured in a simulation that tracks the database state of both the instantaneous system and the real system. Thus, p_{stale} is used at system design time, to develop algorithms and to select parameter values that yield acceptable p_{stale} and s_q values.

There is a fundamental tradeoff between p_{stale} and s_q, i.e., a system may reduce p_{stale} by delaying transactions that were going to read stale data. However, in order to exploit this tradeoff, the system must be able to determine when a transaction is about to read stale data, or at least when it is *likely* that a read will be stale. In the rest of this section we discuss mechanisms for this.

One approach is to mark relevant derived data as stale whenever an update arrives for base data. In the above example, when the update for b_1 arrives at time t_1, the database marks both b_1 and d stale. When transaction T attempts to read either, the database will see that they are marked as stale and can then restart or delay T as desired. A lazy approach is also possible: When a derived object is read, check the timestamps of all of its base data to see if any is more recent than it. The relative performance of the two approaches will depend on the ratio of reads to writes, as well as the fan-in.

In either case the database needs to know the relation between the base and derived data (i.e., which base data affects which derived data), ideally in the form of a dependency graph. To build a dependency graph we need to know the read and write sets of each recomputation transaction. This can be achieved in two ways: by having the user explicitly define read and write sets for the rules that define derived data, or by having the "rule compiler" automatically extract the read/write sets. The later approach is used in Cactis [HK89], but only works if the rules are all written in a known programming language where access sets can be syntactically determined at compile time.

For our experiments we will assume the existence of an exact dependency graph that gives the precise set of objects read and written by a rule. In some experiments, this graph will be used to block transactions that are about to read stale data (if our main goal is to make $p_{stale} = 0$). In others, the graph is simply used by the simulator to measure the percentage of transactions that read stale data.

4 Algorithms

In this section we describe the algorithms for derived data management. Each algorithm can be characterized by its approach to the following five issues. Our focus in this paper is on the last three issues, but we include the first two for completeness.

1. *Relevancy testing.* One technique for reducing recomputation load is to check in advance if an incoming update affects derived data at all; if it does not, the update can be ignored. Blakely originally proposed this technique in the context of materialized views [BLT86,BCL86]. For example, consider a view V defined as $V(A,B) = \Pi_{A,B}(R_1(A,C,D) \bowtie R_2(C,B))$ where R_1 and R_2 are two base relations and A, B, C, and D are attributes. If a tuple is inserted to R_1 whose values on the A and C attributes are the same as another tuple already in the relation, then the newly inserted tuple will not affect V and is considered *irrelevant* to the view. The difficulty with this technique is choosing tight relevancy tests. Blakely formulated the necessary and sufficient conditions of relevancy for the class of views defined by project-select-join expressions. In non-relational systems with complex derivation of derived data, this technique is hard to apply. Also, relevancy tests involve testing the unsatisfiability of certain Boolean expressions, which could be computationally expensive.

   In summary, recomputation algorithms can be classified by whether they do relevancy testing or not. Since relevancy testing is only possible in a few applications, for our performance studies we assume that relevancy testing is not used.

2. *Decoupled Transactions.* As discussed in Section 3, recomputations may be performed within or outside the context of the transactions performing the base data updates. We believe that decoupled recomputations are best for a high performance system, so in this paper we only consider this type of recomputation.

3. *Incremental Recomputations.* Recomputations can be performed more efficiently if they are done incrementally [Han87]. Instead of fully recomputing a derived item by reading the values of all of the base data items it depends on, it is recomputed using its old value and the set of changes to the base data since its last recomputation. Significant performance gain, due to fewer data accesses and shorter computation, can usually be achieved. This performance improvement is especially marked for derived data with large fan-in. One problem with incremental recomputation though is that not all functions can be computed incrementally. For those that can, additional information may have to be kept. For example, to recompute the variance of a set of values incrementally requires one to know about the average of the square of the values too. In our study, we will consider both incremental (I) and full recomputation (F).

4. *Batching.* As discussed earlier, batching can also reduce the cost of recomputations. The idea is to coalesce several updates into a single recomputation. Batching can be performed in one of the following ways:

   (a) *No Batching (NB):* Recomputation transactions are created when an update transaction commits, one per each derived item that the update affects.

   (b) *On Demand (OD):* Derived data is only recomputed when it is invalid (or stale) and a read is attempted. This approach is also referred to as *lazy* or *deferred* recomputation, and has been studied frequently with regard to view maintenance ([Han87,AL80,RK86, SG90,SJGP90]). On-demand has limited applicability in active systems. In particular, it cannot be used for recomputations that must trigger an application transaction or alerts. For the remaining recomputations, OD requires a dependency graph so that when derived data is read, it can be determined if it is stale and if so, what recomputations must be triggered. In spite of these limitations, we still include it in this study for the cases when it can be used.

   (c) *Periodic (P):* A recomputation transaction is periodically invoked for each derived object.

   (d) *Periodic or On Demand (POD):* This is a combination of periodic recomputation and the on demand approach. Derived data is recomputed periodically unless a transaction tries to access it and finds that it is invalid, in which case it is recomputed immediately.

   (e) *Forced Delay (FD):* When an update u on a base item is applied, a recomputation transaction r is generated for each affected derived data, say d. The recomputation r, however, is not released until a fixed time after update u commits. During the delay, further updates to base data of d will not generate new recomputations.

5. *Block on Stale Data.* Our final issue deals with whether the system tries to minimize p_{stale} by blocking reads when it believes that the data is stale. As discussed in Section 3, a blocking system (BL) keeps a dependency graph to detect possible stale reads. A non-blocking system (NBL) always lets application transactions read data.

We will use the notation X/Y/Z to identify a particular algorithm, where X is either I (incremental) or F (full recomputation), Y is the batching scheme (NB, OD, P, POD, FD), and Z is either BL (blocking) or NBL (non-blocking). Due to space limitations we do not consider further the periodic schemes -/P/- and -/POD/-. We believe that these schemes are dominated by some of our other algorithms. Furthermore, it is usually hard to select a good recomputation period. If it is set too short, the recomputation load is high and s_q (transaction response time) becomes too high. If it is set too long, data becomes stale and p_{stale} grows. Also, we do not consider -/OD/NBL algorithms: if we do a recomputation on demand when a read occurs, it is supposedly so that transaction reads the new value, so a blocking strategy is most natural.

Having outlined the algorithms we will evaluate, in the rest of this section we will briefly discuss some implementation issues related to how these various options interact with each other. The derived data is defined through rules; each rule specifies (a) an update event of a base data object, (b) a condition, and (c) a recomputation procedure. For incremental recomputations (I), the procedure needs the old and the new values of the updated base object. Since the recomputation may be delayed, when the update occurs the system generates a *delta record* that specifies the old and new information. This record is then given to the recomputation procedure when it eventually runs. For full recomputations, delta records are not necessary; the procedure reads whatever data it needs from the database to derive the new values.

When incremental recomputation procedures are batched together, the corresponding delta records are combined into a *delta set.* Thus, a single instance of the recomputation procedure is eventually fired; it takes as input the delta set, and from it computes the new derived values for the entire batch of updates. For this to work, the code in the recompute procedure must know how to handle delta sets, and furthermore, the programming environment in which it was developed should have facilities for accessing the various components in delta sets. In Section 7 we suggest some programming structures to facilitate the writing of recomputations based on delta sets.

Delta records and delta sets need to be saved until the recomputation takes place. In most cases, the storage requirements are small and the information can be kept in main memory. (If a failure occurs and memory is lost, the derived data can be computed from the base data directly.) However, in some cases, the storage requirements could be non-trivial. In particular, with on-demand recomputation, if a derived object is updated frequently but not read for a long period of time, the delta set could become large. Similarly, for a high fan-out base object, its delta record must be included in the delta set of many derived objects until these are read, leading again to large sets. We could deal with this problem with a modified on-demand scheme that delays recomputation until a read occurs or until the delta set becomes too large. Since this modified scheme could deal with growing delta sets without significant performance impact, in our performance evaluations we simply assume that delta sets fit in main memory.

5 Performance Model

In order to compare the different algorithms for scheduling recomputations, we define a performance model that specifies in more detail the structure of the database, its interaction with the external world, and how transactions are processed. The model is developed in the context of a program trading application. The model is driven by an actual trace of stock price changes. However, the recomputations and application transactions are synthetic, generated to approximate a realistic environment. To isolate the key issues for derived data management, without becoming lost in irrelevant details, we have made simplifying assumptions about the application and about the database system as described below.

5.1 Data and Transactions

The relationship between data and transactions in our model mirrors the system shown in Figure 1. We ignore the "other data" from the figure and instead concentrate on the base data, in this case stock data, and the derived data. There is an object for each stock being tracked, which stores many details about the stock (e.g. last price, trade volume, daily high and low). The stock base data is then used to calculate a number of derived objects such as composite averages and theoretical option prices. There are three types of transactions in this system:

updates are write only transactions that change the value of one stock object. For our experiments, the updates that are generated correspond to actual stock trace activity taken from the TAQ database provided by the New York Stock Exchange. Both the arrival time of each update and the stock that is changed are real values from a day of trading.

recomputations are transactions that calculate the value of derived objects using the stock data. Both the fan-in, the number of base objects which are read, and the fan-out, the number of derived objects which are updated, are simulation parameters. The recomputations are actually fired according to the algorithm under consideration (Section 4).

queries are read only transactions that are generated by users of the system. They can read both base and derived data, but modify neither. They are assumed to have Poisson arrival.

We assume that updates to base data are installed using the *updates first* scheme of [AGMK95a]. In this scheme, updates are queued separately from other transactions, and are executed in FIFO order with highest priority. Recomputations and queries are combined into a second class which is also scheduled in FIFO order, but only if no update transactions are queued. If an update arrives while a recomputation or a query is being processed, the running transaction will not be preempted. We believe that this updates-first scheme is the most appropriate for the program trading application because of its data driven nature. (Actually, [AGMK95a] reports that a *split updates* scheme could be superior, but we do not use it here since it requires partitioning the data by importance to the application.)

5.2 Storage Model

The database is composed of N_{stocks} stock objects and $N_{derived}$ derived objects. We assume that the entire database cannot fit in main memory and so must be stored on disk. For simplicity, we do not model the disk caching scheme in detail. Instead, we define the probability that a desired page will already be present in memory to be determined by the parameter p_{cache_hit}. If the page is not present, an I/O will have to be performed which will take io_{lookup} or io_{access} milliseconds for index page reads or data page reads respectively. Log writes will usually be to main memory as well, but when a page fills it requires io_{write_log} milliseconds to write it. We assume that the disk can only service one request at a time, so additional requesters must wait for the current request to finish before starting. The waiting transactions are maintained in two FIFO queues: one for updates and one for other transactions. Updates are always serviced first for I/O, just as in CPU scheduling.

5.3 Fan-in/Fan-out Scenarios

In the introduction, we discussed the fan-in and fan-out of derived data. We have chosen to study the effects of the two properties separately in our simulations. We have therefore defined two scenarios to study, each focusing on only fan-in or fan-out.

In the first scenario, we test the performance of the scheduling algorithms when computing derived data that have high fan-in, such as composite indices. Each of the $N_{derived}$ derived objects is computed from N_{fan_in} stock objects. For each derived object, the required stocks are chosen randomly from all of the stocks but weighted by the number of occurrences of each stock in the trace. For example, a stock that appears 500 times is twice as likely to be used in a derived object as one that appears 250 times. We feel this is a reasonable approach because the most important companies (for predicting general economic and industry specific trends) are also the companies that are most heavily traded.

The second scenario tests the performance of the scheduling algorithms when computing derived data such as theoretical option prices which have moderate fan-out. The fan-in of the derived objects is set to 1 which means that each is computed from only one stock object. The fan-out, defined as $\overline{N_{fan\_out}}$, is the average number of derived objects computed from each stock. The actual number of derived objects computed from each stock depends on the the number of occurrences of the stock in the trace as in scenario 1. For example, a stock that appears 500 times will have twice as many objects derived from it as one that appears 250 times. We feel this is a reasonable approach because the most heavily traded companies are also the ones with the most options (and other derivatives) traded.

Description	Parameter	Value
# of instructions executed per second	ips	100×10^6
# of instructions to find one data object	cpu_{lookup}	500
# of instructions to check condition of rule	$cpu_{check\_condition}$	500
# of instructions to compute derived value (fixed)	$cpu_{compute_f}$	5000
# of instructions to compute derived value (incremental)	$cpu_{compute_i}$	1000
# of instructions to perform query excluding I/O time	cpu_{query}	1000000
# of instructions to begin a transaction	$cpu_{start\_transaction}$	7500
# of instructions to commit a transaction	$cpu_{commit\_transaction}$	7500
# of instructions to begin an update	cpu_{start_update}	1500
# of instructions to commit an update	cpu_{commit_update}	1500
I/O time (ms) to find one data object	io_{lookup}	10
I/O time (ms) to access one data object	io_{access}	10
I/O time (ms) to write log record	io_{write_log}	1
I/O cache hit rate	p_{hit}	0.99

Table 1: Scheduler baseline settings for performance

5.4 Metrics and Parameters

We only use three metrics to evaluate the scheduling algorithms. The two primary metrics, $\overline{s_q}$ and p_{stale}, were motivated and described in Section 3. The third metric, $\overline{N_r}$, is the average number of recomputes per second and is provided to help explain system behavior.

The values of the simulation parameters were chosen as reasonable values for a typical workstation. Where possible, we have performed sensitivity analysis of key parameter values. The simulator is written in **DeNet** [Liv90]. Each simulation experiment (generating one data point) ran for approximately 2 hours of simulated time, driven by the stock price trace data. The simulation parameters are detailed in Tables 1, 2 and 3.

Description	Parameter	Value
stock trace time scale factor	f_{trace}	1.0
query arrival rate	λ_q	50.0
mean # of stock objects read by query	μ_s	50
S.D. of # of stock objects read by query	σ_s	3
mean # of derived objects read by query	μ_d	5
S.D. of # of derived objects read by query	σ_d	1
mean # instructions to perform query	μ_q	1000000
S.D. of # instructions to perform query	σ_q	100000

Table 2: Scheduler baseline settings for transactions

Description	Parameter	Value
# of stock objects	N_{stocks}	10000
# of derived objects	$N_{derived}$	200
fan-in	N_{fan_in}	100
average fan-out	$\overline{N_{fan\_out}}$	2
recompute delay (s)	t_{delay}	0.5

Table 3: Scheduler baseline settings for data and algorithms

6 Results

In this section we present selected results from our simulations. Our goals are to compare the various algorithms with respect to our performance metrics (Section 5.4), and to study how the characteristics of the derived data affect the timeliness of both data and transactions. By evaluating the system under different settings, we are able to address questions such as:

- Which algorithm is best under which conditions?
- What is the best delay value for the Forced Delay strategy to balance data timeliness with recomputation cost?

To answer the second question, we fill experiment with four versions of the Forced Delay scheme. They are denoted as FD(d) where d is the value of the delay (t_{delay}).

We focus first on derived data with high fan-in in Section 6.1, followed by the fan-out case in Section 6.2. All of the experiments described in this section were run with the parameter values described in Tables 1, 2 and 3 unless otherwise stated. Additional results, including various sensitivity analyses, are described in [AKGM95].

6.1 Effects of fan-in

To begin, we examine the performance of algorithms I/-/NBL (incremental recomputation without block on stale reads). This means that recomputations are not expensive and applications are allowed to read stale derived data. To compare the algorithms under different loads we vary the fan-in of the derived data (e.g. the number of stocks used to compute a composite index). By varying fan-in, we can control how many base data items a derived object depends on, and thus the frequency that the derived data is recomputed. Figure 4(a) shows how the different algorithms maintain the timeliness of the data. As expected, under OD $p_{stale} = 0$ because transactions that try to read stale data cause a recompute to be triggered, thereby refreshing the derived data as they read. Also, the no batching scheme (NB), which releases a (decoupled) recomputation transaction immediately after an update, keeps derived data relatively fresh: less than 1 in 10 queries tries to read a derived data object in the small window between update and recompute. The FD algorithms, which force the delay to be larger, exhibit higher rates of p_{stale} in direct relation to the size of t_{delay}.

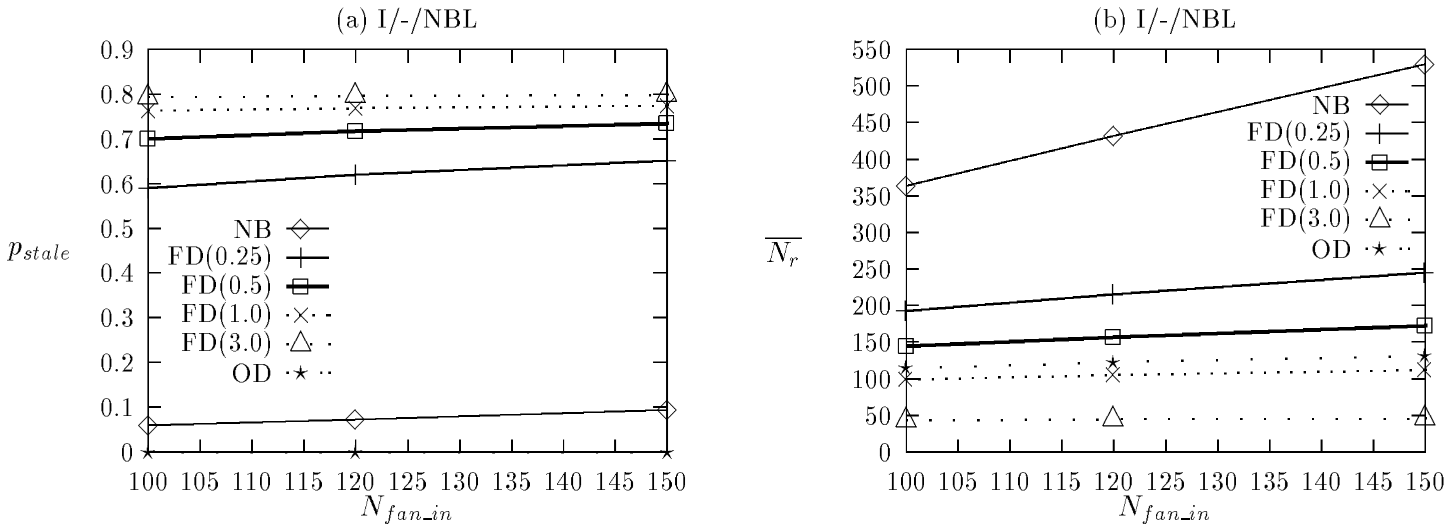

Figure 4: Effects of N_{fan_in} with baseline settings.

Even with a small delay, p_{stale} is around 60%. Even though sometimes queries are reading stale data under FD, we remark that under the non-blocking scenario, transactions are *allowed* to read stale data. Also, the stale data which is read is not very old. Since queries arrive randomly, the average age of a stale item is $\frac{t_{delay}}{2}$ (at most 1.5 seconds in our experiments). This delay is probably dwarfed by the time it takes an update to arrive at the database itself. (Stock price information is entered by hand at terminals on the exchange floor, so the time between a trade and the update for that trade will be many seconds.)

We now look at the number of recomputations the database has to perform per second, $\overline{N_r}$. It was our claim in Section 2 that delaying and batching recomputes that exhibit locality would significantly reduce the number of necessary recomputes. Figure 4(b) supports this. Even a small delay, such as a quarter of a second, reduces N_r by half or more. FD(3.0), which has the longest delay, reduces the number of recomputes by over an order of magnitude when $N_{fan\_in} = 150$. More importantly, notice that NB does not handle large fan_in well. As is seen in the figure, NB requires 50% more recomputations when N_{fan_in} is increased by 50%. The Forced Delay schemes however, effectively absorb the added recomputation load caused by high fan-in situation. FD(3.0), for example, keeps the recomputation rate to 50 per second over the range of fan-in shown.

As seen in Figure 5(a), the savings in recomputation effort by batching directly relate to a reduction in query response time, s_q. More recomputations mean more contention for resources, both CPU and I/O, which in turn means longer delays for queries. When $N_{fan\_in} = 150$, queries take 50% longer under NB than they do under the other algorithms. By allowing queries to read slightly out-of-date data, the FD algorithms significantly improve query response time. This s_q improvement is significant but not as large as the difference in N_r seen in the previous graph. The reason is that with incremental recomputation, most of the work in the system is generated by

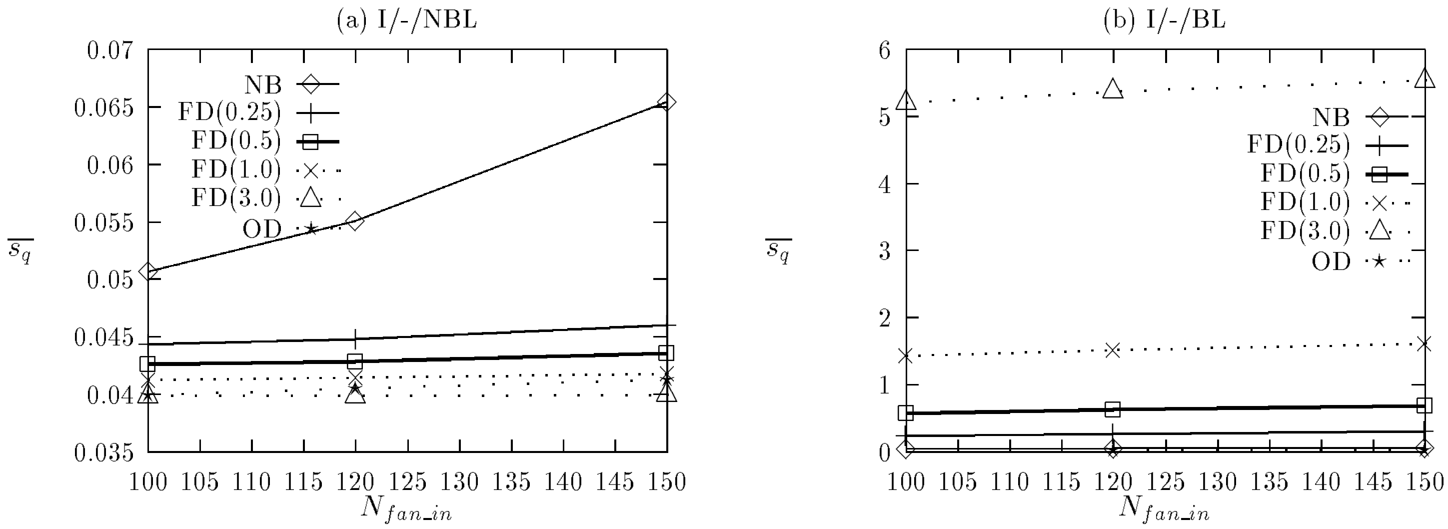

Figure 5: Effects of N_{fan_in} on query response time.

queries so that even a large reduction in the recomputation load does not have a huge effect on the total system load (and hence waiting times). In systems where the recomputation work dominates the system (e.g., high fan-in, large number of derived items, full recomputation) the ratio of s_q between NB and FD would approach the N_r ratio. Such an example is shown in next section on complete recomputation.

In the case where no stale reads are allowed and queries must block if they read stale data (I/-/BL), the conclusions about query response time change dramatically. For FD with a large t_{delay}, we expect that service time will be very long since a query that tries to read stale data will have to wait $\frac{t_{delay}}{2}$ on average. Figure 5(b) illustrates this effect. FD(3.0) has a query response time of over 5 seconds which implies that it is waiting for more than one derived object to be recomputed. Figure 5(b) clearly shows that FD with large delays is inappropriate for applications that do not allow stale reads. (FD still has good throughput though, because it reduces the number of necessary recomputes; hence it lowers the system load. The recompute graph is not shown since it is identical to figure 4(b).)

In conclusion, OD seems to outperform the other algorithms in satisfying both data and transaction timeliness: Stale data is never read, and transaction response time is kept low by recomputing only if necessary. In systems where lazy recomputation is possible, therefore, OD appears to be the best choice. As discussed in Section 4, however, in data-driven systems OD will not work because alerts will not be sent unless a query reads the relevant derived item. In these cases the choice is between NB and FD. If the application can tolerate reads of slightly stale data, FD with a small delay (in this experiment, 0.25 or 0.5 seconds) is a good tradeoff between data and transaction timeliness. Finally, for cases where data timeliness is extremely important, NB should be chosen over FD.

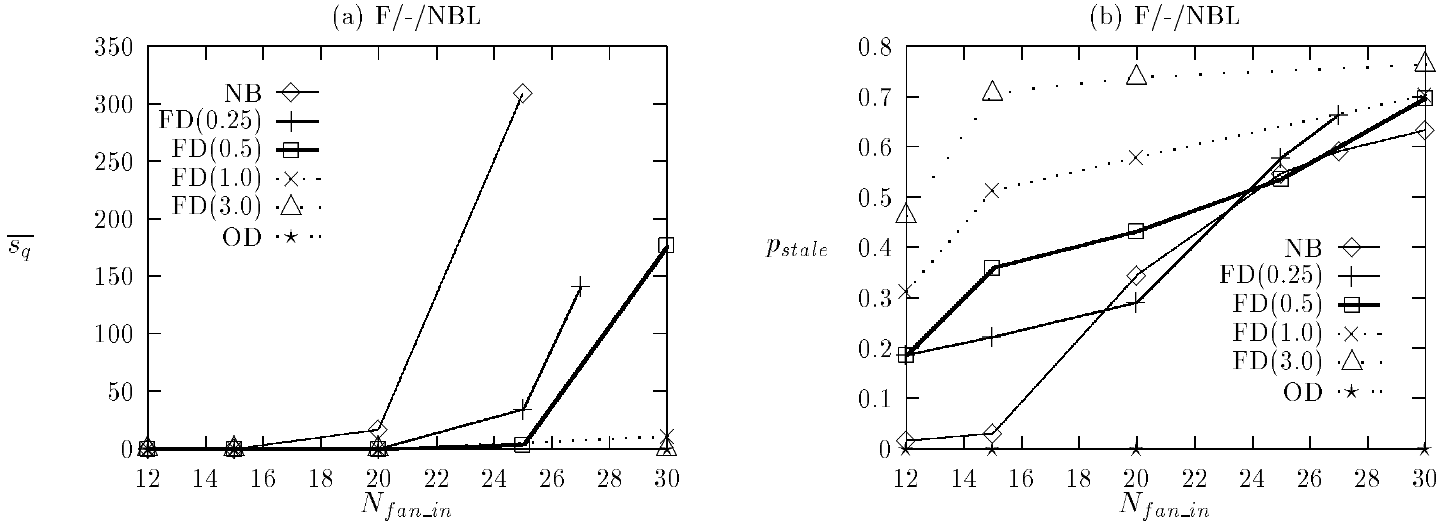

Figure 6: Effects of N_{fan_in} with complete recomputes and high I/O contention.

6.1.1 Effect of Complete Recomputation

When recomputation is full (F/-/NBL), the cost to perform a single recomputation rises in direct proportion to the fan-in. This impacts the performance of the database in two ways. First, since recomputation becomes much more expensive relative to query processing, we would expect that algorithms that reduce N_r will show even greater gains in s_q than in the previous section. Second, since both the number of recomputes and the cost per recompute increase with N_{fan_in}, we would expect that the system load (CPU and I/O) will increase drastically. Figure 6 supports this. The database using NB starts to overload with $N_{fan\_in} < 20$, far lower than the 100-150 range used in the incremental experiments. Due to the high loading of the system, reducing the amount of recomputation work yields large benefits in query response time: Switching from NB to FD(0.25) allows the system to handle 30% more fan-in (which also translates to 30% more recomputation load). The heavy load also causes NB, which under moderate loading maintains data timeliness well, to degrade below the performance of the FD algorithms. For example, in Figure 6(b), p_{stale} under NB goes up from 1% to 60% as fan-in increases. This is due to long queueing time that causes long delay to recomputation transactions. As in the incremental case, OD performs very well in both metrics and across the entire range of fan-in. In conclusion, in systems with complete recomputation or with heavy loading, OD should be used if possible. If OD is not appropriate, FD is a clear winner over NB.

6.2 Effects of fan-out

In this section we discuss the performance of the algorithms when maintaining derived data with fan-out instead of fan-in. Due to space constraints, we only report a few results and direct the reader to the technical report for more details [AKGM95]. The relative performance of the algorithms is

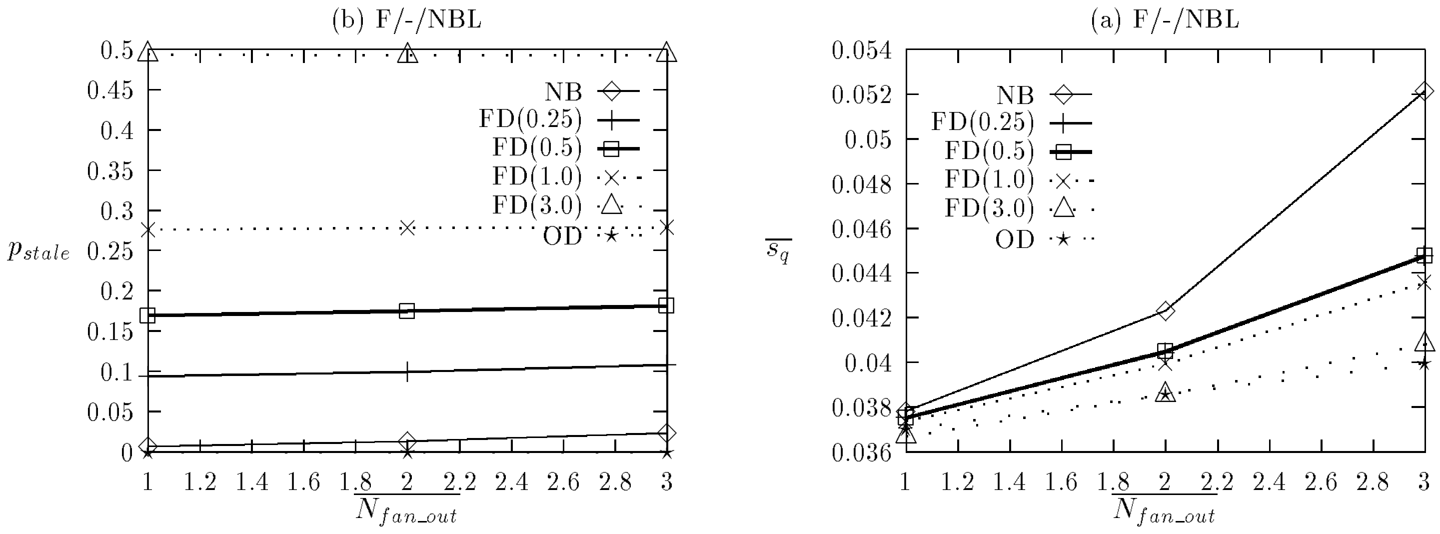

Figure 7: Effects of $\overline{N_{fan\_out}}$.

similar to that of the fan-in experiments: OD and NB do the best job maintaining data timeliness, whereas FD and OD best maintain transaction timeliness. However, notice in Figure 7(a) that in this experiment the FD algorithms keep the derived data far fresher. Each derived object only becomes stale when the value of a particular base object changes, so the relative rate of invalidation is much lower. FD(0.25) maintains p_{stale} near 10% compared to 65% in Figure 4(a). The algorithms are also closer in their query response times: NB is only 20% worse than FD(0.25) when $\overline{N_{fan\_out}} = 3$ compared to 50% worse in Figure 5(a). Still, it is clear that s_q is growing fastest for NB and that at larger values of $\overline{N_{fan\_out}}$ the difference will widen. In conclusion, for maintaining fan-out data, OD is the best choice if it is applicable. If not, FD with a low delay is again a good compromise choice.

7 System support for batching

The simulation results from the last section reinforce the intuitive benefits of batching recomputations. In this section, we will examine to what extent current systems support batching and how they can be extended to provide more support. In particular, we focus on supporting the *forced delay* algorithm for both its good performance and its high suitability to data-driven systems where a lazy approach is inappropriate. The *forced delay* algorithm imposes four requirements on a system:

1. **Delayed Execution -** It must be possible to specify that a triggered transaction be released only after a fixed delay.

2. **Unique Execution -** If a particular transaction is in its delay period, no other transaction of the same type should be triggered.

3. **Storage of Delta Sets-** Since the recomputation is decoupled from the transaction that triggered it, the database must maintain the set of changes to base data and pass them to the triggered transaction so that it can perform incremental recomputation.

4. **Control Over the Unit of Batching -** In the two cases studied by simulation, fan-in and fan-out, recomputations were batched differently. For high fan-in derived data, the fixed delay algorithm combined updates to many *different* base data into one recomputation. For the fan-out case, only updates to the *same* base data object were batched.

These four requirements are supported to different degrees by currently proposed systems. Requirement 1 is supported by systems which have temporal rules such as HiPAC [DHL90]. Requirement 2 is not supported directly by any proposed system, but can be emulated by using the rule deactivation feature of HiPAC and others: When a rule triggers a recompute, it can deactivate all rules (including itself) that could trigger another recompute of the same type. When the recompute finally runs, it can reactivate all of the rules that trigger it. Building and maintaining such a system is fraught with danger. For example, if the recompute transaction is aborted some other transaction must be run to re-enable all of the rules. Also, the addition of a new rule requires updating all of the other related rules and the recompute transaction. This scheme is even more difficult in combination with requirement 4, controlling the unit of batching. Using deactivation to guarantee uniqueness fixes the granularity of batching at the rule level: If we desire finer granularity, we must create many specific rules to replace one general rule. This can lead to an unacceptable increase in the total number of rules in the system. We illustrate a practical example of this problem below.

To our knowledge, no proposed system provides direct support for requirement 3. To simulate this aspect of unique transactions, application programmers will have to create and maintain the information explicitly using custom data objects. This necessitates more changes to the rule and transaction definitions, more complexity, and hence more opportunities for errors. Similarly, we know of no system with support for requirement 4.

We believe that all four requirements could be met with a simple mechanism we call *unique transactions*. The rest of the section is devoted to describing a unique transaction facility and demonstrating how it can be used to implement the forced delay algorithm easily. The semantics are very similar to those described for FD in Section 4. When a unique transaction is triggered, the system must check if the same transaction is already queued. If it is, the delta structure of the current triggering transaction is appended to the delta set of the queued transaction. Thus when the queued transaction finally runs, it has access to the all of the delta structures of the rules which fired it.

We illustrate unique transactions in the context of a program trading application implemented in a generic active database (we do not presuppose a data model or any particular rule implementation). To simplify the example, we will assume the rule system follows a simplified version of the the standard Event-Condition-Action (ECA) model in which the condition checks are performed within the action and rules are only triggered by updates to data items. A richer rule system designed for STRIP is described in [AGMK95b].

Assume that the stock information is stored in Stocks (either a table or a class) and that each stock object (or row) has two attributes, symbol and price. We will expand the schema of the database as the examples require. First, we consider the computation of a weighted composite average such as the S&P 500 index (SP500). Recomputation will be triggered by updates to any of the 500 base stocks. We need to store the weighting of each stock in the composite, so we create a new class SP500_Weights whose members have the two attributes symbol and weight. [1] A straight forward way to maintain SP500 is the following:

```
DEFINE TRANSACTION compute_sp500_1
    BEGIN CODE
        FIND index IN Weights WHERE index.symbol = old.symbol
        IF (index ≠ NIL) THEN
            sp500.value = sp500.value + (new.price - old.price) × index.weight
        END
    END CODE
END DEFINE
```

This transaction first tries to find an entry for the stock in Weights. If found, the value of the composite is recomputed incrementally using the before and after image of the stock object changed which are provided by the delta structure as 'new' and 'old'. (Our view of delta structures is modelled on the NAOS system described in [CCS94].) Rules of this form can be written in existing databases, both relational and object-oriented. Unfortunately, we have no way of specifying that recomputations should be batched. Let us now rewrite this rule using unique transactions:

```
DEFINE TRANSACTION compute_sp500_2
    UNIQUE
    BEGIN CODE
        REAL composite_change = 0.0;
        FOREACH DELTA DO
            FIND index IN Weights WHERE index.symbol = old.symbol
            IF (index ≠ NIL) THEN
                composite_change += (new.price - old.price) × index.weight
            END
        END
        sp500.value += composite_change;
    END CODE
END DEFINE
```

When a stock price changes, compute_sp500_2 can be triggered with a fixed delay. Every time compute_sp500_2 is triggered within the delay, the delta structure of the triggering transaction will be appended to the delta set of the current instance of compute_sp500_2. When compute_sp500_2

[1]Alternatively, another attribute could be added to the stock objects to store their weightings for the composite but this approach would require one new attribute per composite for every object in Stocks. Membership and weighting information could also be hard-coded into the rules but that may be too rigid in a system where the weightings change.

finally runs, it is able to incrementally compute the new composite value using all of the deltas. The actual statements to manipulate delta sets are not described here in detail, although this example demonstrates how 'FOREACH' can be used to iterate over them. Generally, we have tried to keep the interface simple and leave sophisticated processing to the recompute transaction in order to ease system implementation. For example, there is no attempt to compose events to provide the triggered transaction with only a net result.

Unique transactions as defined so far are very useful for high fan-in derived data such as composite indices, but what about for high fan-out objects such as theoretical option prices? With unique transactions as defined, we can only combine recomputations performed by transactions with the same name, so we have limited control over the unit of batching (requirement 4). To illustrate the problem concretely, we extend our schema to define a new class of objects Options, each object having the attributes stock_symbol, exercise_price, expiration, and theor_price. The attribute stock_symbol corresponds to the underlying stock the option is based on, theor_price is the theoretical price of the option which the database must maintain, and the other attributes are parameters used in that calculation (see Section 1). Using the same approach as we used for compute_sp500_2, the rule for recomputing theoretical option prices can be written as:

```
DEFINE TRANSACTION compute_option_1
    UNIQUE
    BEGIN CODE
        FOREACH DELTA DO
            FOREACH option IN Options WHERE option.symbol = old.symbol DO
                option.theor_price = f(new.price,option.exercise_price,...)
            END
        END
    END CODE
END DEFINE
```

Recomputation defined this way will correctly maintain the theoretical option prices, but the unit of batching is too coarse: Only one recomputation transaction will be permitted for all of the theoretical option prices that need to be recomputed, regardless of which stock they are derived from. In this instance, there is little gain in computing the theoretical prices for options based on totally different stocks at the same time. In addition, grouping so much diverse work into one transactions would limit the system's flexibility in scheduling. For example, the recomputations would have to be serialized, forcing all unstarted recomputations to wait for the I/O of the current one.

Intuitively, the recomputations should be batched for the options of a single stock, which was the behavior simulated in the last section. In that case, all of the changes to the underlying stock but the last can be discarded, saving CPU cycles and I/O but without reducing the scheduler's flexibility. To allow finer control of the unit of batching, we further extend unique transactions to qualify how the system determines uniqueness. The transaction compute_option_1 can be rewritten as:

```
DEFINE TRANSACTION compute_option_2
    UNIQUE ON new.symbol
    BEGIN CODE
        discard all but newest delta structure
        FOREACH option IN Options WHERE option.symbol = old.symbol DO
            option.theor_price = f(new.price,option.exercise_price,...)
        END
    END CODE
END DEFINE
```

The definition of compute_option_2 shows the extension to the UNIQUE statement. By further qualifying uniqueness, which was previously based only on the name of the transaction, with the value new.symbol, we are specifying that there can be multiple copies of the transaction enqueued as long as the value of new.symbol is different in each. This means that there will be only one version of compute_option_1 enqueued for each base stock. By qualifying the uniqueness definition this way, we can control the unit of recomputation which is batched, thereby reducing the amount of work required for derived data maintenance without sacrificing system flexibility.

8 Conclusions

Management of derived data is a critical task in real-time database systems. In this paper we have presented and analyzed a variety of schemes for managing this derived data, balancing the needs for up-to-date data with those for processing cycles for application transactions. Our results indicate that a good general-purpose strategy is to delay recomputations slightly (Forced Delay), to allow batching of related recomputations. A lazy, On-Demand strategy can perform even better for those applications where derived data does not trigger application transactions. Incremental recomputations are best (with either of the above schemes), but again, are only feasible in some cases.

Since batching of incremental recomputations is so beneficial (reduces processing load while still keeping data relatively fresh), we believe that systems need to provide good support for this. Thus, we also proposed some basic transaction constructs that let the application programmer specify which recomputations can be batched together, and how the recomputations on a set of updates should be performed. The STRIP real-time database system we are currently implementing is incorporating such facilities for incremental and batched recomputations.

References

[AGMK95a] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, 1995.

[AGMK95b] B. Adelberg, H. Garcia-Molina, and B. Kao. Rules in strip. Technical report, Stanford University, 1995. Available by anonymous ftp from db.stanford.edu in /pub/adelberg/1995.

[AKGM95] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for maintaining derived data. Technical report, Stanford University, 1995. Available by anonymous ftp from db.stanford.edu in /pub/adelberg/1995.

[AL80] M. Adiba and B. Lindsay. Database snapshots. In *Proceedings of the 6th VLDB Conference*, pages 86–91, 1980.

[BCL86] J. A. Blakely, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *Proceedings of the 12th VLDB Conference*, pages 457–66, 1986.

[BLT86] J. A. Blakely, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, pages 61–71, 1986.

[CB94] M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. In *Proceedings of the 20th VLDB Conference*, pages 714–721, 1994.

[CCS94] C. Collet, T. Coupaye, and T. Svensen. Naos - efficient and modular reactive capabilities in an object-oriented database system. In *Proceedings of the 20th VLDB Conference*, pages 132–43, 1994.

[CJL91] M. Carey, R. Jauhari, and M. Livny. On transaction boundaries in active databases: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):320–36, 1991.

[DHL90] U. Dayal, M. Hsu, and R. Ledin. Organizing long-running activities with triggers and transactions. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, pages 204–14, 1990.

[Han87] E. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, pages 440–453, 1987.

[HK89] S.E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems*, 14(3):291–321, 1989.

[Liv90] M. Livny. **DeNet** user's guide. Technical report, University of Wisconsin-Madison, 1990.

[Ram93] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[RK86] N. Roussopoulos and H. Kang. Perliminary design of ADMS $\pm$: A workstation-mainframe integrated architecture for database management systems. In *Proceedings of the 12th VLDB Conference*, pages 355–364, 1986.

[SF90] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 512–20, 1990.

[SG90] A. Segev and H. Gunadhi. Temporal query optimization in scientific databases. *IEEE Data Engineering*, 13(3), sep 1990.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, 1990.

[SL90] X. Song and J. Liu. Performance of multiversion concurrency control algorithms in maintaining temporal consistency. In *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference*, pages 132–139, 1990.

[SS87] A. Segev and A. Shoshani. Logical modeling of temporal data. In U. Dayal and I. Traiger, editors, *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, pages 454–466, San Francisco, CA, may 1987. acm, ACM Press.