The STRIP Rule System For Efficiently Maintaining Derived Data *

Brad Adelberg† Hector Garcia-Molina‡ Jennifer Widom§

Abstract

Derived data is maintained in a database system to correlate and summarize base data which records real world facts. As base data changes, derived data needs to be recomputed. This is often implemented by writing active rules that are triggered by changes to base data. In a system with rapidly changing base data, a database with a standard rule system may consume most of its resources running rules to recompute data. This paper presents the rule system implemented as part of the STanford Real-time Information Processor (STRIP). The STRIP rule system is an extension of SQL3-type rules that allows groups of rule actions to be batched together to reduce the total recomputation load on the system. In this paper we describe the syntax and semantics of the STRIP rule system, present an example set of rules to maintain stock index and theoretical option prices in a program trading application, and report the results of experiments performed on the running system. The experiments verify that STRIP's rules allow much more efficient derived data maintenance than conventional rules without batching.

Keywords: derived data, view maintenance, active database system, transaction scheduling, real-time database.

1 Introduction

This paper describes and evaluates the STRIP rule system. STRIP is a main-memory resident soft real-time database system implemented at Stanford. The major new feature of the STRIP rule system over previous active rule systems is its *unique transaction* facility. This facility allows for very efficient incremental maintenance of derived data. The two benefits provided by unique transactions are:

- they allow rule actions to act on database changes *batched* across transaction boundaries, not just within one transaction, and
- they allow the batches to be *partitioned* in any way that reduces the cost of the derived data computation.

Rather than inventing an entirely new active database model, unique transactions in STRIP have been added as an extension to SQL3-type triggers.

STRIP and its rule system are intended for real-time monitoring applications. These applications monitor a dynamic environment to discover the occurrences of "interesting" events. For

*This work was supported by the Telecommunications Center at Stanford University, by Hewlett Packard and by Philips.

†Northwestern University Department of Computer Science. e-mail: adelberg@ils.nwu.edu

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

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

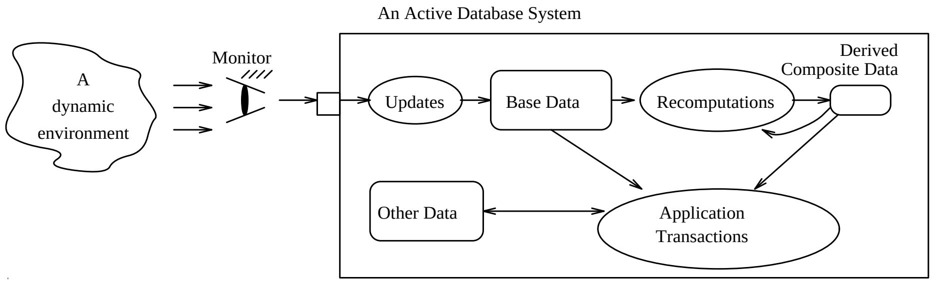

Figure 1: A high level model for the derived data maintenance problem.

instance, a military radar system can track aircraft and signal alerts when a dangerous pattern appears. Similarly, a program trading application can monitor the prices of stocks and other commodities, looking for good opportunities. Figure 1 illustrates the major components of a such a system. 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 data*: 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 (the derived data) represents the aggregate price of 500 U.S. stocks (the base data). 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 (derived data).

The base and derived data are read by application transactions to determine how to react to changes in the environment. The application transactions may also need additional data. For instance, in a trading system, the database would also store the current stock holdings and outstanding trading orders. This type of additional data is labeled "other data" in Figure 1.

The base and other data used by the application need to be stored in the database, but the derived data does not: it could be computed from the base data every time it is needed. Still, it is often far more efficient to precompute the derived data and store it in the database. This is known as *materializing* the derived data. Of course the materialized data will have to be updated whenever the base data changes, but if the ratio of derived data reads to base data updates is high, or if there are tight constraints on access time to the derived data, it may still be worthwhile to materialize it. Also, if only a small portion of the base data is changing at any given time, it is often possible to correct the derived data based only on the changes to the base data rather than recomputing it completely from scratch. This is called *incremental maintenance*.

While few database systems provide direct support for maintaining derived data, it is possible to perform this task using rules in active databases. For instance, in [CW91] the authors demonstrate how rules can be automatically generated to incrementally maintain a broad class of views in a relational database. Applying these results to our program trading example, a rule might be written to update all of the derived data that changes when a stock price changes. After each base data change, the rule will be fired and will modify the materialized derived data accordingly.

The work performed by the rules to maintain the derived data can be very high, for the following reasons:

- Recomputing a derived item may be expensive. For instance, pricing models for financial instruments often involve trend analysis and complicated statistics.

- 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].

- Some base data, often the most frequently changing, is used to compute a lot of derived data. This leads to a multiplicative effect on the base data update rate. For example, a popular stock may have fifty listed options derived from it as well as numerous composite indexes.

As observed in [AKGM96a], it is often the case that a single base datum or set of related base data changes in bursts and then remain constant for a relatively long time. Using the stock example, a small price change in a stock may trigger a burst of quotes until the market makers settle on a new price. This may be followed by minutes of inactivity. If a rule is fired for every individual change, the materialized views will be recomputed many times in a small time window, effectively swamping the CPU.

Instead, the unique transaction facility provided by STRIP's rule system allows the price changes to be collected during the burst and combined into a single recomputation. Intuitively, when base data is changed that is used to compute derived data, a new transaction is created to maintain the derived data, but the transaction is not immediately run. Instead, it is delayed for a period of time specified by the application designer. During this *delay window*, any additional changes to base data that affects the same derived data are grouped with the original changes so that when the new transaction finally runs, it can process all of the relevant changes. This way only one recompute transaction is run instead of one per change. Of course, during the delay window there is an inconsistency in the database since the base data has been changed but the derived data has not. In many applications, such short term inconsistencies are acceptable, especially if the value of the derived data does not move substantially from its previous value. One of the challenges, then, of using unique transactions is to pick a delay window size that improves performance without severely affecting the timeliness of the derived data.

The remainder of this paper will present the following:

- A description of the STRIP rule language and in particular unique transactions (Section 2).

- An example stock market application that serves both to illustrate the power of STRIP's rules and to provide a benchmark with which to evaluate the current STRIP implementation (Sections 3 and 4).

- An analysis of the experimental results of running the stock market application on STRIP version 2.0 (Section 5).

- A description of the implementation of STRIP's rule system that describes how "bound tables" are stored and how unique transactions are managed (Section 6).

- A description of related work (Section 7) and our conclusions from this study (Section 8).

```
create rule rule-name on t-name
    when transition-predicate
    [ if condition ]
    then
        [ evaluate query-commalist]
        execute function-name
        [ unique [ on column-commalist] ]
        [ after time-value ]

transition-predicate ::= event [,event [,event]]
event ::= inserted | deleted | updated [column-commalist]
condition ::= query-commalist
query-commalist ::= query [,query]*
query ::= table-expression [bind as bound-table-name]
column-commalist ::= column-name [,column-name]*
```

Figure 2: Syntax of rules in STRIP.

2 Rule Language

This section presents the syntax and semantics of the STRIP rule language. Since STRIP's rules have been designed as an extension of SQL3-type rules (see, e.g., [DD93][WC96]), rather than describe a complete rule system we will concentrate on the novel features of our rules. The current implementation, which supports most of the details presented in this section, is described further in Section 6.

First we will present the syntax for rule definition in STRIP and then discuss the semantics. The grammar is based on [DD93]. The syntax for creating rules in STRIP is shown in Figure 2.

The *transition predicate* (see Figure 2) in the rule definition specifies which events should trigger the rule. The three possible events are insertions, deletions, or updates to rows in the table on which the rule is defined, `t-name`. If the rule should only be triggered by updates to certain columns, a list of the columns can be attached to the **updated** event.

Event checking occurs at the end of each transaction prior to commit. Consider a particular transaction, T, preparing to commit. First, event checking is performed to determine which rules T has triggered. Then, for every rule that has been triggered, the system evaluates the rule's condition (given by the **if** clause of the rule) which is composed of one or more queries. The condition evaluates to true if there are no queries or if every query returns one or more rows.

The queries in the **if** clause can be over the entire database as well as over the four transition tables for `t-name`: `inserted`, `deleted`, and `new` and `old` for updated tuples. STRIP does not reduce the transition tables to net effect: e.g., if a tuple is inserted and then deleted within the same transaction, it will appear in both the `inserted` and `deleted` transition tables. This gives the user more control by providing an "audit trail" where desired. It is always possible for the application to calculate net effect on its own using the transition tables as provided. The system adds an extra column to the transition tables called `execute_order` which contains a sequence number that orders the tuples changed within the transaction. (If a tuple is updated, the corresponding tuples in the `old` and `new` transition tables will have the same value in their `execute_order` attribute.) Using the `execute_order` column, the application can recreate the order of the changes in the transition tables.

If a rule condition evaluates to true, the queries in the **evaluate** clause are evaluated within the triggering transaction. These queries do not affect the rule condition and are only used to pass tables to the action (see below).

Finally, a new transaction, T', is created to perform the rule action given in the **execute** clause. Rule actions in STRIP are executed by application-provided functions that are linked into the database and are treated as black boxes. The user functions are not called with any parameters but it is possible to pass data into them using the *bound table* mechanism as described below.

Transaction T' is released as soon as the triggering transaction, T, commits unless a delay is specified in the **after** clause. (This *coupling mode* is called "sequentially causally dependent" in [BBKZ93]). One consequence of executing rule actions in a new transaction is that rule processing is simplified. Because the rule condition contains only side-effect free queries, evaluating the condition can never alter the state of the database and hence cannot trigger any new rules. For the same reason, the order that the rules are considered by T at commit time is unimportant. Finally, the time required to process rules within the triggering transaction is much more predictable, which is beneficial in a real-time environment.

A less desirable consequence of performing actions in separate transaction is that the new transaction T' will not automatically have access to either the transition tables or the condition query results from transaction T. If either are needed, STRIP provides a mechanism to explicitly pass them. By using the **bind as** construct within STRIP rule definition, any of the query results from evaluating a rule in T can be passed to T' as a named temporary table. For example, the rule

```
create rule foo on table1
when inserted
then evaluate
        select *
        from inserted
        bind as my_inserted
    execute my_function
```

shows how a transition table can be passed to the transaction T'. Since there is no rule condition, any inserts to table `table1` will trigger the rule. The rule action renames and binds the inserted transition table and then creates a new transaction to execute the function `my_function`. Within the code of function `my_function`, the transition table will appear as an ordinary but read-only table named `my_inserted`. Queries in both the **if** condition and the **evaluate** clause can have their results bound and passed. (The reason for a separate **evaluate** clause is to pass data that does not contribute to the truth or falsehood of the rule condition.) To avoid scoping problems, the names chosen for the bound tables should not be used elsewhere in the database. The one exception is that other rules calling on the same function can use the same bound table name as long as it is defined identically. This is useful if more than one event should be handled by the same action code.

As described so far, every time a rule is triggered and its condition evaluates to true, a new transaction is created to execute a user function. As described in Section 1, however, if many such transactions are being triggered within a short time span it is often more efficient to batch the changes and execute only one transaction to service all of them. The STRIP rule system supports this through the use of *unique transactions*, specified by including **unique** in the **execute** clause. A transaction being unique means that at any given time there is at most one such transaction queued in the system to execute a particular user function. If a rule fires that would trigger another transaction with the same function, no new transaction is enqueued. Instead, the tuples of the bound tables of the new rule firing are appended to those of the bound tables of the currently enqueued transaction. This is true even if the second rule is a different one from the first — the bound tables of all rules executing the same user function are combined (and must be defined identically, as mentioned above). Therefore, when the triggered transaction finally runs, it will have access to all of the changes that occurred since the time it was first triggered. Once a unique transaction begins to execute, its bound tables are fixed and any new rule firings will start a new transaction. An example where unique transactions are used very effectively is given in Section 3, and the implementation of this mechanism is described in Section 6.

As with transition tables, STRIP does not calculate the net effect of bound tables. If a tuple is changed in one transaction and that data is bound into a table, and then the same tuple is changed in another transaction which gets batched with the first, the resulting bound table will contain two entries, one for each change. It is up to the application code to properly calculate net effect if desired. If the application needs to preserve the order of the changes across transactions, the rule should define its bound tables to include a column called `commit_time`. This column is automatically instantiated at bind-time with the commit time of the transaction that produces the row in the bound table.

Sometimes batching all of the transactions that execute the same user function into one transaction is too extreme. For instance, we may want to batch only those transactions that would update the same tuple of a materialized view. In such cases, we want transactions to be unique on the combination of the user function *and* certain column values in the bound tables. This behavior can be achieved by qualifying the **unique** clause with the names of the unique columns.

As a simple example, consider a table $X(A, B)$ and a view V defined as: **select** A, **sum**(B) **from** X **groupby** A. If the rule to recompute V is defined as unique, all changes to X will be batched together in one transaction that will then change many tuples in V. By defining the rule **unique on** $X.A$, only changes to the same tuple in the view will be batched together. That is, there can be as many triggered transactions as there are values for A.

Having informally described the semantics of the STRIP rule system, the next section demonstrates how to use STRIP's rules to maintain data for an example stock market application. The reader interested in a more formal discussion of the behavior of unique transactions is directed to Appendix A.

3 Stock Market Example

In this section we describe a simplified program trading application (PTA) and show how the STRIP rule system can be used to maintain the derived data it requires. The same application will be used in Section 5 to evaluate the performance of the current implementation of the rule system. In practice, program trading systems are custom built by each trading firm and their market models and trading algorithms are closely held secrets. Thus the example we present here is simplified both out of necessity, since very little information is publicly available, and also to focus our attention on the important issues of data management without getting lost in the details of financial modeling. Still, we feel that our application model captures the important features of the real problem, and we will point out how to extend it where appropriate.

The PTA requires the database to maintain three types of prices: stock prices, composite index prices, and theoretical (call) option prices. The stock prices are the base data of the system, and are updated in the database according to the market feed. The composite and option prices, however, are derived data that must be computed from the stock prices. In actuality, the current trend of feed providers is to send more than stock prices with the feeds, including popular composite prices (e.g., Dow Jones Industrial Average (DJIA)) and other derived values. Still, additional derived data, such as that related to proprietary market models, will always need to be computed by the rule system. Because composite averages and theoretical option prices have known functions, are easy to understand, and reasonably reflect the types of data that need to be computed, we choose to compute them as part of the PTA as representative of the proprietary derived data.

The database for the PTA contains the following six tables:

`stocks(symbol,price)` - contains the current price of every stock as reported by the market feed.

`stock_stdev(symbol,stdev)` - contains the standard deviation of the annualized rate return of every stock. The standard deviation is usually computed from the daily closing prices of the stock over a period of years. While periodic recomputation is supported by STRIP, we do not model the computation of standard deviation in this study since we are focusing on the behavior of the PTA during trading hours. For our purposes, this table is base data.

`comp_prices(comp,price)` - contains the computed price of every composite average (e.g., DJIA). This table is a materialized view (defined below) and hence corresponds to derived data.

`comps_list(comp,symbol,weight)` - describes the many-many relationship between stocks and composites. This table is entered by the trading firm and changes very infrequently. It is an example of the "other data" from Figure 1.

`option_prices(option_symbol,price)` - contains the computed price of every listed option. The data in this table is derived and its materialized view definition is given below.

`options_list(option_symbol,stock_symbol,strike,expiration)` - describes the one-many relationship between stocks and options. This table must be updated once every three months when the option exchanges create new options and expunge expired options. In this example we only consider call options so the attribute `strike` is the price at which the option holder can buy the stock `stock_symbol` prior to expiration. We consider this table base data since it reflects the listed options currently being traded on the exchanges.

The tables `comp_prices` and `option_prices` are materialized views with the following definitions:

```
create rule do_comps1 on stocks
when updated price
if
     select comp, comps_list.symbol as symbol,weight,
          old.price as old_price,new.price as new_price
     from comps_list,new,old
     where comps_list.symbol = new.symbol
          and new.execute_order = old.execute_order
     bind as matches
then
     execute compute_comps1

define function compute_comps1
     real composite_change;
     foreach row r in matches
          composite_change = r.weight * (r.new_price - r.old_price);
          update comp_prices
          set price += composite_change
          where comp = r.comp;
end function
```

Figure 3: Standard rule to maintain `comp_prices` for stock price changes.

```
create view comp_prices as
select comp,sum(price*weight) as price
from stocks,comps_list
where stocks.symbol = comps_list.symbol
group by comp

create view option_prices as
select option_symbol,f_BS(price,strike,expiration,stdev) as price
from stocks,stock_stdev,options_list
where stocks.symbol = options_list.stock_symbol and stocks.symbol = stock_stdev.symbol
```

The function f_{BS} computes the theoretical price of an option based on the Black-Scholes pricing model. See Appendix B for the full equation.

In this paper we only focus on the rules that maintain these two materialized views as the stock prices change. In practice, we would need additional rules to handle changes to the other tables, e.g., `comps_list` or `options_list`. We will ignore these additional rules, both due to space constraints and because these other tables change very infrequently.

A straightforward way to handle stock price changes, and the only way available in most rule systems, is through a rule similar to the one shown in Figure 3. The rule `do_comps1` states that when the price attribute of any stock changes, and that stock is used to compute at least one composite, the transaction `compute_comps1` should be run to update `comp_prices`. When `compute_comps1` runs, it will need to know information about the stocks that changed and how they impact the composite prices; this information is assembled when the condition query is checked. Note that the stocks

symbol	price
S1	30
S2	40
S3	50

comps_list

comp	symbol	weight
C1	S1	0.5
C1	S3	0.5
C2	S1	0.3
C2	S2	0.7

comp_prices

comp	price
C1	40.0
C2	37.0

Transaction T_1

S1: 30→31

S2: 40→39

matches

comp	weight	old_price	new_price
C1	0.5	30.0	31.0
C2	0.3	30.0	31.0
C2	0.7	40.0	39.0

Transaction T_2

S2: 39→38

S3: 50→51

matches

comp	weight	old_price	new_price
C2	0.7	39.0	38.0
C1	0.5	50.0	51.0

Figure 4: Stock composite indices example.

condition query must equate the `execute_order` of the `old` and `new` transition tables to be sure that if the triggering transaction changed a stock price more than once, the correct old and new tuple images are being considered together. To allow `comp_prices` to use the data computed in the condition, we bind the results as the temporary table `matches` which is then passed to the action transaction. Using `matches`, the transaction running `compute_comps1` can update `comp_prices` without any other database reads.

Consider the example shown in Figure 4 in which the tables are populated and two simple transactions that change the stock prices are shown. Transaction T_1 changes the prices of S1 and S2 which triggers rule `do_comps1`. The condition of the rule is satisfied since computing the condition query returns three rows. The result of the query is temporarily stored in table `matches`, and a new transaction, T_{1a}, is triggered immediately after T_1 commits. Transaction T_{1a} will execute the function `compute_comps1`, and when it does, it will be able to read the result of the condition query as the table `matches`.

Suppose that before T_{1a} runs, Transaction T_2 executes. Since T_2 also changes the prices of stocks that influence composite prices, it also triggers a new transaction, T_{2a}. Transaction T_{2a} will also have access to a table called `matches` although it will see the second `matches` table shown in Figure 4. Although T_{1a} and T_{2a} update the same tuples in `comp_prices`, they remain in the system as two distinct transactions since `compute_comps1` has not been defined as **unique**. Figure 5(a) shows the two transactions that are enqueued and the bound tables that they will be able to access when they execute.

As mentioned in the introduction, base data often changes in bursts. Hence the situation matches (T_{1a})

comp	weight	old_price	new_price
C1	0.5	30.0	31.0
C2	0.3	30.0	31.0
C2	0.7	40.0	39.0

matches (T_{2a})

comp	weight	old_price	new_price
C2	0.7	39.0	38.0
C1	0.5	50.0	51.0

(a) Transactions triggered by standard rules

matches (T_{1a})

comp	weight	old_price	new_price
C1	0.5	30.0	31.0
C2	0.3	30.0	31.0
C2	0.7	40.0	39.0
C2	0.7	39.0	38.0
C1	0.5	50.0	51.0

(b) Transactions triggered using unique transactions

matches (T_{1a})

comp	weight	old_price	new_price
C1	0.5	30.0	31.0
C1	0.5	50.0	51.0

matches (T_{2a})

comp	weight	old_price	new_price
C2	0.3	30.0	31.0
C2	0.7	40.0	39.0
C2	0.7	39.0	38.0

(c) Transactions triggered using unique transactions on comp

Figure 5: Triggered transactions for composite example.

of Figure 5(a) with multiple `matches` tables can occur frequently. Thus we would like to batch the changes to a particular composite to reduce the amount of recomputation required. This is accomplished using a unique transaction within the rule, along with a delay window of 1 second, as shown in the Figure 6 (changes from the previous rule and user function are boxed). Since `compute_comps2` is defined as unique, only one transaction of this type can be queued in the system at once. Let us contrast the behavior of this rule to that of the previous rule. As before, transaction T_1 enqueues transaction T_{1a} with a three row bound table. Now, however, assuming that T_2 executes within 1 second of T_1, when T_2 commits it does not enqueue a new transaction. Instead, the rows of the bound table that would have been passed to the new transaction are appended to the bound table of transaction T_{1a}. This is shown in Figure 5(b). Thus when transaction T_{1a} finally runs, it will have the combined bound information of both triggering transactions. Note that this is similar to the behavior described in [CCS94] for deferred rules except that unique transactions allow changes to be batched across transaction boundaries.

Batching changes across transaction boundaries can greatly improve performance but it does require that the user rewrite the action function for maximum benefit. Consider the difference between `compute_comps1` and `compute_comps2` (Figures 3 and 6). The code `compute_comps1` steps

```
create rule do_comps2 on stocks
when updated price
if
     select comp, comps_list.symbol as symbol,weight,
          old.price as old_price,new.price as new_price
     from comps_list,new,old
     where comps_list.symbol = new.symbol and new.execute_order = old.execute_order
     bind as matches
then
     execute compute_comps2
     unique
     after 1.0 seconds

define function compute_comps2
     real composite_change = 0.0;
     foreach row r in
        select comp, sum ((new-old)*weight) as diff from matches groupby comp)
        update comp_prices
        set price += r.diff
        where comp = r.comp;
end function
```

Figure 6: Unique transaction approach to maintain `comp_prices` for stock price changes.

through the `matches` table one row at a time, reading the affected composite, incrementally recomputing it, and then rewriting it. This same code would still work with the rule `do_comps2` but it wouldn't take advantage of the fact that the same composite will probably appear many times in the matches table. In contrast, the code in `compute_comps2` groups the changes according to the composite they affect. The `select` statement aggregates the incremental changes to each composite so that each only has to be read, recomputed, and written once.

Rule `do_comps2` is a big improvement over `do_comps1` since it allows batching, but the unit of batching is quite coarse: all of the changes that affect any tuple in `comp_prices` are applied together. A problem with performing all of the changes together is that the recompute transaction can be very long. Longer transactions have two drawbacks:

- They hold locks longer which leads to increased conflicts. More lock conflicts increase both response time and the variance in response time, neither of which is desirable in a real-time environment. The problem of choosing transaction boundaries in active databases was studied in [CJL91] where similar results were found.

- In a real-time system, transactions may have to be restarted either because they miss their deadlines or because a high priority transaction is blocked waiting for a lock that they hold. Longer transactions result in both more restarts and more lost work when a transaction is restarted.

Hence the goal in choosing a unit of batching should be to balance the performance benefits of

```
create rule do_comps3 on stocks
when updated price
if
    select comp, comps_list.symbol as symbol,weight,
        old.price as old_price,new.price as new_price
    from comps_list,new,old
    where comps_list.symbol = new.symbol and new.execute_order = old.execute_order
    bind as matches
then
  execute compute_comps3
  unique on comp
  after 1.0 seconds

define function compute_comps3
    real composite_change = 0.0;
    foreach row r in matches
        composite_change += r.weight * (r.new_price - r.old_price);
    update comp_prices
    set price += composite_change
    where comp = r.comp;
end function
```

Figure 7: Unique transaction approach (on `comp`) to maintain `comp_prices` for stock price changes.

combining like work and the scheduling flexibility of smaller transactions.

With this goal in mind, we note that grouping all of the changes that affect a single composite price is useful because the old value can be read once, updated, and then written once. If the same changes were in separate transactions it would result in many more reads and writes of the composite. Any larger unit of batching does not seem to generate any further efficiency. This suggests that we batch on attribute `comp`, which would yield the grouping shown in Figure 5(c).

The rule to batch on composite name is shown in Figure 7. When `compute_comps3` runs, the `matches` table it sees will only contain changes for a single composite. This results in simpler code than that used in `compute_comps2`: The for loop accumulates all of the weighted price changes from stock changes and then the total change is applied to the composite price.

We have now seen three rules to incrementally maintain the `comp_prices` table. In Section 5, we compare their performance in experiments running on STRIP v2.0.

In addition to maintaining the view `comp_prices`, we also want to maintain the view `option_prices` that contains theoretical option prices. When a stock price changes, the prices of all of the options that are based on it will change as well. A simple rule to recompute option prices when a stock price changes is shown in Figure 8. Note that since option prices cannot be computed incrementally, the transition table `old` is not bound as part of the condition.

As with composite prices, it may be possible to improve on the performance of this rule using unique transactions, but we must choose a unit of batching. The efficiency to be gained by batching is that if an underlying stock price changes more than once in a delay window, we can use its last

```
create rule do_options1 on stocks
when updated price
if
     select option_symbol,stock_symbol,strike,expiration,new.price as new_price
     from options_list,new
     where options_list.stock_symbol = new.symbol
     bind as matches
then
     execute compute_options1

define function compute_options1
     real last_price, stdev;
     foreach row r in matches
          stdev = select sd from stock_stdev where symbol = r.stock_symbol
          update option_prices
          set price = f(r.new_price,r.strike,r.expiration)
          where option_prices.option_symbol = r.option_symbol;
end function
```

Figure 8: Standard rule to maintain `option_prices` for stock price changes.

value to compute the theoretical option price only once. This is different from the previous example in which we had to consider every change. Hence, batching based on option symbol might be a good choice. On the other hand, there are often many more options in the system than stocks, so batching on such a small unit may lead to a huge number of transactions in the system. Hence it may be wiser to batch on stock symbol, collecting all of the options related to a single stock together: this unit of batching may also allow some partial results used for every option to be computed only once. Section 5 reports the results of experiments to determine which unit of batching performs best on STRIP.

4 Performance Evaluation

To evaluate the performance of unique transactions, we have implemented the program trading application described in the previous section using STRIP. In addition to the rules and user functions, we still need a stream of stock price updates to drive the system, and data to populate the tables. Both are described below. In addition, this section describes the hardware used for the experiments and some basic performance numbers for STRIP.

4.1 Update stream

A program trading application is driven by changing stock prices as reported by a market feed. For these experiments, we use the consolidated quote file provided as part of the New York Stock Exchange's TAQ database [New94]. The quote file lists all of the stock price quotes made during actual days of trading on all of the major U.S. exchanges. Each entry contains the symbol of the stock being quoted, the bid and ask prices, and the time of the quote to the nearest second. To try to recreate the actual timing of the trades within the second boundaries, if more than one quote occurs within a given second we spread them evenly over the 1 second interval. For example, if 3 quotes are recorded at time 54 seconds, we will assume that they occurred at 54 ,54.33, and 54.66 seconds.

The experiments reported here are driven by the actual price changes from the TAQ file recorded during trading in January of 1994. Each experiment lasts for 30 minutes during which the price changes that occurred during real trading on the exchange are applied to the system in real-time. On average, each run contains over 60,000 stock price changes.

In real program trading applications, the quote stream arrives over a high speed network, often dedicated for this purpose only. Because we do not have access to a real-time quote service, we chose to load the entire trace into memory before the experiment begins, which has the following benefits:

- It eliminates overhead which is identical for all of the rules, such as market feed handling, allowing us to focus on the differences in rule performance.
- Our test machine is not on an isolated network, so the traffic generated by other users would make a controlled experiment very difficult.

4.2 Populating the tables

The table `stocks` is populated from the stocks quoted in the trace file and contains 6600 stocks. The table `comp_prices` contains 400 composites, each calculated from 200 stocks. The component stocks in each composite are chosen randomly but in direct proportion to their trading activity as measured by the number of price changes in a day of trading. We feel that this is indicative of the distribution of real stocks in real composites: the stocks of large companies which trade frequently are most often used in composites since they tend to be good indicators of entire industries. For instance, Netscape, which trades a few thousand times a day, is more likely to be used as a proxy for the Internet sector than is Spyglass, which trades a few hundred times a day. The many-many relationship between stocks and composites requires 80,000 rows and is stored in table `comps_list`.

In the baseline experiment, there are 50,000 options prices computed which means that the tables `options_list` and `option_prices` each contain 50,000 entries. As with composites, the division of options between the stocks is randomly generated but in relation to the frequency of trading of the underlying stocks. Hence the expected number of listed options for a particular stock is the total number of options times the fraction of the entire trace due to the particular stock. The other attributes of table `options_list`, besides `stock_symbol`, are chosen randomly but from a reasonable range of values. Since we do not actually use the theoretical option prices that are computed, and since the option pricing model is not data dependent, the choice of these attributes is not important.

4.3 Transactions

In a real program trading application, there can be four types of transactions in the system: base data updates, recomputations of derived data, user queries, and expert system code. We have

Action	Time (μsec)
begin task	10
end task	5
begin transaction	1
commit transaction	10
open cursor	60
fetch cursor	15
update cursor	10
close cursor	25
get lock	10
release lock	30
run scheduler	5

Table 1: Basic performance numbers for STRIP v2.0

chosen to focus on the first two because the last two are not necessary to study recomputation costs.

The update transactions are driven by the TAQ quote trace, with one update transaction being run for every price change. The recomputation transactions are triggered by the rules defined in Section 3. The code for the recompute transaction is similar to that shown in Section 3 although since STRIP v2.0 implements only a subset of SQL, some of the aggregation had to be done in the application code rather than in the database. The functions to compute composite and option prices, which appear in Appendix B, are fully implemented. The standard normal distribution function, $\Phi()$, is computed using the error function in the C math library.

4.4 System

The performance numbers reported here are for STRIP running on an HP-735 under HP-UX 9.03 with 144Mb of main memory. No other user processes were run on the system during the experiments. Response times are measured using the Unix function `gettimeofday`. The fraction of the CPU required for an experiment was measured using the Unix call `times`. In order to understand the reported results, some basic timing measurements on STRIP v2.0 (measured using `gprof`) are reported in Table 1.

Let us calculate the approximate timing of a simple transaction that updates one tuple. In STRIP, transactions must be executed within a task, which is the database's unit of scheduling. A task can contain zero or more transactions but every transaction must be contained within exactly one task. If a user sticks with one transaction per task, we can calculate the cost of a simple cursor based update of one tuple as begin_task(10)+begin_transaction(1)+get_lock(10)+open_cursor(60)+ fetch_cursor(15) +update_cursor(10)+close_cursor(25)+release_lock(30) +commit_transaction(10)+ end_task(1) = 172μsec. This results in a throughput of 5814 transactions per second(TPS), which is slightly lower than the 7000 TPS observed in practice. From this we can see that STRIP is a very high performance database. In addition, the overhead of the task/transaction mechanism is not very high, which is beneficial to rules that are unique on smaller granularities since they tend to create more transactions.

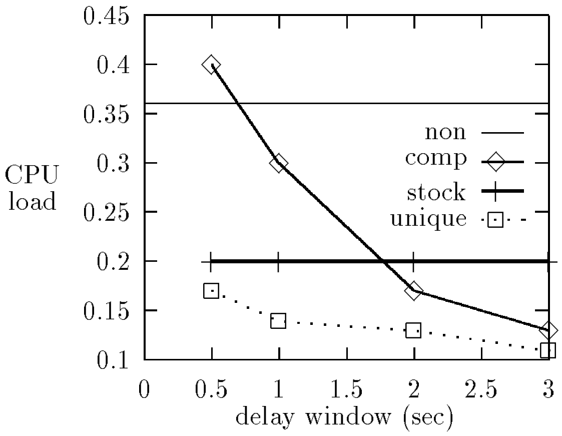

Figure 9: Effects of delay window on CPU usage.

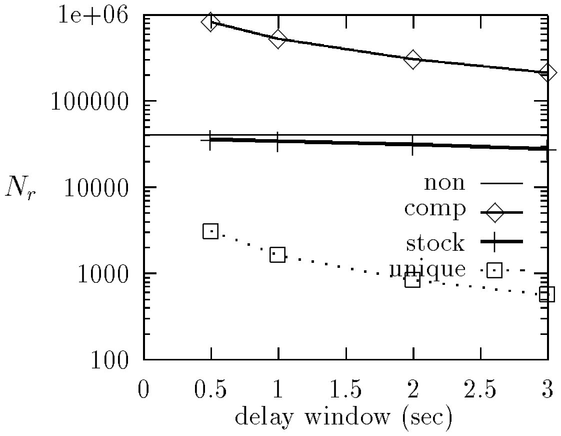

Figure 10: Effects of delay window on number of recomputations, N_r.

5 Results

This section reports selected results of the experiments run on STRIP. By evaluating the system under different settings, we are able to address such questions as:

1. Do unique transactions significantly reduce the recomputation effort?
2. How long should the delay window be?
3. What is the most efficient unit of batching to use?
4. How do different units of batching affect the length of the recompute transactions and hence the schedulability of the system?

The answers to these questions very much depend on whether the derived data can be maintained incrementally, as with `comp_prices`, or if it must be maintained non-incrementally, as with `option_prices`. Therefore we divide our results and discussion into two parts, Section 5.1 for `comp_prices` and 5.2 for `option_prices`.

5.1 Maintaining `comp_prices`

In this section we describe the performance of STRIP maintaining the materialized view `comp_prices`. We ran experiments using both unique and non-unique transactions. For the unique transactions, we considered three units of batching: the entire table (**unique**), per stock symbol (**unique on** `symbol`) and per composite symbol (**unique on** `comp`). We also varied the size of the delay window for the unique transactions from 0.5 to 3 seconds.

The results for CPU utilization are shown in Figure 9. Maintaining the composite using non-unique transactions uses 36% of the processing resources, as indicated by the horizontal line on the graph. Given that our test scenario does not try to support user queries or expert system transactions, 36% is a large part of the system's capacity to already be claimed. As shown by the figure, however, this load can be greatly reduced by using unique transactions.

The performance of the rules, and especially batching on composite, can be better understood by examining the number of recomputation transactions that were run during the experiment, N_r. As Figure 10 shows, batching on the composite symbol causes an order of magnitude more recompute transactions to be run than without any batching. This is to be expected since using the numbers from Section 4, the average stock participates in 12 composites and hence every price change will trigger 12 recomputations on average. When the delay window is small, there is little batching, so virtually every stock change triggers 12 new transactions. As the window is lengthened, more changes start to be batched and the number of recomputations performed decreases. The rule that causes the fewest recomputations is coarse batching (unique), which should be no surprise since there can be only one transaction queued at any given time.

Having to execute more recompute transactions increases the load on the system in two ways. First, there is overhead in starting and completing a task/transaction as reported in Section 4. Second, more recompute transactions means more tasks in the system at the same time which increases the scheduling time. Even in combination, these two sources of overhead are negligible when the number of transactions is small — executing queries and computing user functions dominates the CPU. The CPU load stays flat as the number of transactions increase until a critical region when transaction management costs become comparable to query costs. At this point, small increases in the number of transactions can noticeably increase CPU load. In the reported experiment, the critical region begins at approximately 500,000 recomputations so only unique on composite is affected by transaction overhead. Note that, when its delay is set above 1 second we see that its performance drastically improves.

These results help answer the first three questions listed at the beginning of this section. First, unique transactions dramatically reduce the recomputation load on the system: using a 3 second delay, two of the unique rules require less than $\frac{1}{3}$ of the non-unique processing. This is especially impressive in a main memory database such as STRIP where the access costs are small and hence the potential benefits of batching are small as well. In disk resident databases, where the access cost is higher, we would expect the gains to be even greater. In fact, simulation studies reported in [AKGM96a] do show improvements of an order of magnitude in CPU utilization.

It is more difficult to answer the second question on the optimal size of the delay window because increasing the window continues to decrease the recomputation load. For our experiment, it is clear that the gains due to increasing the delay window seem to taper off beyond 3 seconds. Since many stock market applications can accept delays in this range, the ideal delay window for our application is between 2 and 3 seconds. The exact point at which the a balance is reached between CPU usage and data timeliness depends on the application. One of the strengths of unique transactions, however, is that they allow the application to adjust this tradeoff, even on a view by view basis.

Regarding the best unit of batching to use, the most efficient in this experiment was the coarsest, unique with no qualifying columns. This allowed the system to minimize the number of recomputation transactions, and hence overhead, and perform the most batching. Batching on composite symbol performed almost as well for higher delays since it allowed full batching, but the greater number of transactions that had to be run kept it slightly worse.

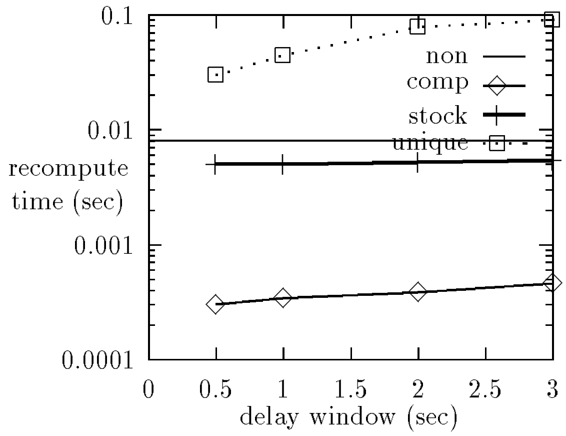

Figure 11: Effects of delay window on recompute transaction length.

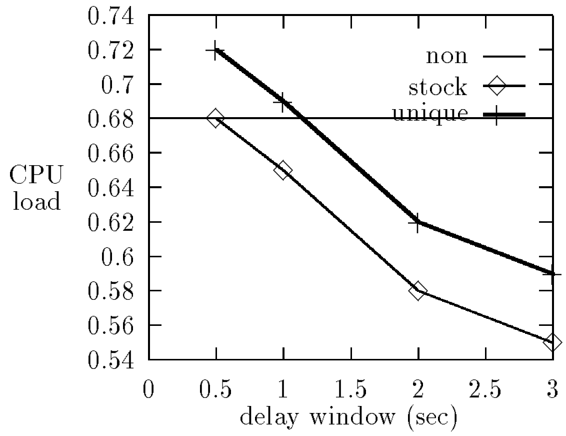

Figure 12: Effects of delay window on CPU usage.

Reducing CPU load is not our only goal, however. Another important goal is to keep the recomputation transaction as short as possible. As discussed in Section 3, shorter transactions hold locks for less time and thus reduce conflicts in the system. Fewer conflicts result in shorter and more predictable response time, a critical metric of a real-time system.

Figure 11 plots the average system time spent per recompute transaction minus queueing time. We can see that while coarsest batching yields the best performance gains, it also results in the longest recomputation transactions. In fact, its recomputation times are an order of magnitude longer than batching on stock symbol or non-batching, and over two orders of magnitude longer than batching on composite symbol. Hence, assuming that short transactions are important, the best overall rule to maintain composite averages is batching on composite symbol.

To summarize, all of the rules using unique transactions achieved enough batching for delays ≥ 0.7 seconds that they outperformed non-unique rules. Coarse batching reduces the CPU load the most but batching on composite symbol is nearly as good and results in better system schedulability. So in this case, the best unit of batching is the one that combines just enough work to allow us to take advantage of the natural overlap without trying to combine too much.

5.2 Maintaining `option_prices`

In this section we describe the performance of STRIP maintaining the materialized view `option_prices`. Whereas `comp_prices` could be incrementally maintained, `option_prices` cannot. Also, weighted composites have a high *fan-in* (200 stock prices), meaning that they are computed from many other data, whereas option prices do not (1 stock price plus 3 other base data items). Hence in this section we explore how these differences affect the performance of unique transactions.

As before, we ran experiments using both unique and non-unique transactions. Again, we wanted to consider three units of batching: the entire table (**unique**), per stock symbol (**unique on** `symbol`) and per option symbol (**unique on** `option_symbol`). After initial experiments, however, we discovered that the *fan-out* from stocks to options was so high that batching on option symbols led to an unmanageable number of transactions in the system. Consequently, the results for batching on `option_symbol` are not shown in the graphs.

Figure 12 shows the fraction of CPU capacity required to maintain `option_prices`. The horizontal line is the performance of the system using non-unique rules. This graph raises two interesting points. First, for delays starting at slightly more than 1 second, both rules using unique transactions outperform the standard approach. For a delay of 3 seconds, batching on stock symbol required 20% less CPU time than non-unique transactions. While this is a significant savings, it is less than was achieved in the previous section. This is due to the high fan-out nature of maintaining option prices as opposed to the high fan-in of maintaining composite averages. In order to batch option recomputations, we need changes to the same stock to occur within the delay window (*temporal locality* in [AKGM96a]), while with the composite rules, we only need changes to different stocks that contribute to the same composite (*temporal-spatial locality* in [AKGM96a]). Thus the nature of maintaining the composite prices yields more batching for a given delay window.

The second interesting point from Figure 12 is that the coarsest unit of batching, unique with no qualifying columns, is not the best performer — batching on stock symbol uses less CPU. This is a contrast from the results in the previous section. In addition, batching on stock symbol achieves this performance despite requiring that almost two orders of magnitude more recomputations be run, as shown in Figure 13. This can be explained as follows:

- Since the number of recomputations required by batching on stock symbol is below the "critical region" (described in Section 5.1), the impact of transaction management on the overall CPU load is negligible.

- Depending on how the rule is written, grouping the rows of the bound table by stock symbol is done in different places in the code. If the rule is defined unique on stock symbol, the rule system groups the rows into different bound tables before the user code is run. If no qualifying columns are added to the unique clause, the rows are grouped in the recompute code provided by the user. Implementation peculiarities of STRIP v2.0 result in the former being slightly faster than the latter. We plan to correct this discrepancy in the next revision of the code.

- The longer running transactions of the coarse batching seem to be preempted more often, both by update arrival and by system processes, causing more context switches.

We anticipate that with further performance tuning of STRIP and its interaction with the operating system the last two points can be addressed. If so, coarse batching and batching on stock symbol will have very similar CPU usage.

Even if the CPU usage is similar, however, batching on stock symbol results in shorter recomputations. As shown in Figure 14, the length of its recomputation transactions are two orders of magnitude smaller than with coarse batching, the benefits of which were discussed in Section 5.1. Combining shorter transactions with less CPU usage, batching on stock symbols is the clear winner in this set of experiments. As with maintaining `comp_prices`, the best unit of batching is the one that allows redundancies in the calculations to be omitted without trying to batch too much together.

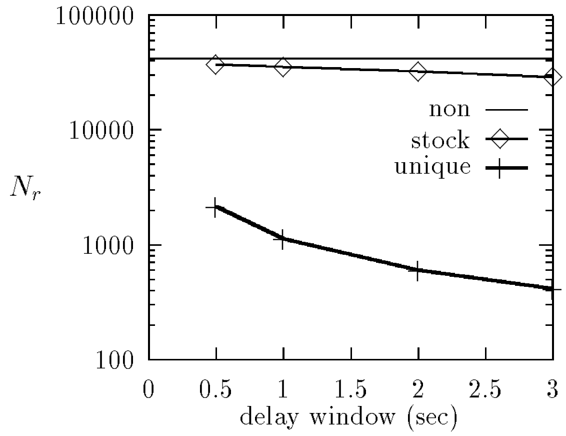

Figure 13: Effects of delay window on number of recomputations, N_r.

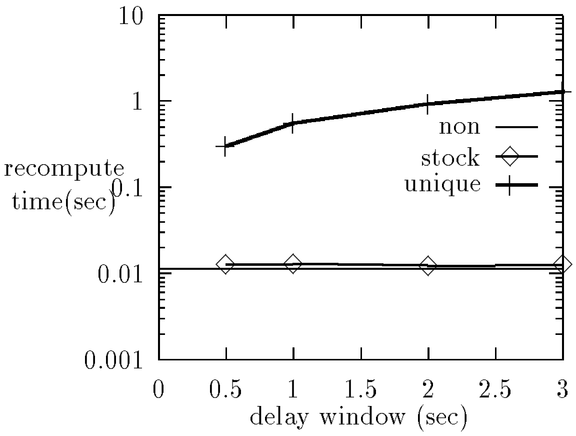

Figure 14: Effects of delay window on recompute transaction length.

6 Rule System

This section describes the implementation of the STRIP rule system. Due to space constraints, we focus only on the implementation issues related to unique transactions.

6.1 Table storage

There are two types of tables in STRIP:

standard tables - that are created and deleted via the SQL **create** and **drop** commands.

temporary tables - that are created and deleted by the database and used to store intermediate query-processing results, transition tables, and bound tables.

The tuples of a standard table will be called standard tuples and tuples of a temporary table temporary tuples. We use "table" or "tuple" to refer to a standard table or tuple unless otherwise noted. Both types of tables are stored in STRIP as linked lists of tuples. The format of the tuples, however, is different.

In standard tables, the values of the attributes are actually stored in the tuple as opposed to storing pointers to the values as in [PTV90]. In the current implementation, only fixed-length fields are permitted so tuple lengths are constant as well. In STRIP, the order of rows in standard tables is unimportant and tables can be indexed using either a hash or red-black tree structure.

For a temporary table, T, storing the actual attribute values in the tuple, t, would be wasteful since for all of our temporary tables we can use pointers instead; this requires one indirection but usually saves quite a bit of space. This suggests storing t as an array of pointers to attribute values. Still, storing one pointer for each attribute requires more memory than is necessary. As proposed in [Rou82], we instead store one pointer to each standard tuple that contributes at least one attribute to t. (For aggregate, computed, or timestamp attributes, the actual value must still be stored, however, since it doesn't exist anywhere else and hence cannot be pointed to.) When temporary table T is created, a static mapping is built that specifies for each column in the temporary table which tuple pointer to follow and what the offset is in the referenced record.

For example, consider a temporary relation $V(A, B, C, D, E)$ that is a natural join of the three relations $R(A, B, C)$, $S(C, D)$, and $T(D, E)$. Using the simplest attribute pointer scheme, each tuple t of V would need an array of 5 pointers. In the STRIP scheme, each tuple is stored as $[t_r, t_t]$ where t_r is a pointer to the R tuple that contributed attributes A, B, and C to t and t_t points to the tuple in T that contributed attributes D and E. Notice that because relation S only has attributes that are contributed by other relations, no pointer to a tuple in S need be stored.[1] The static map for V is the structure $[(R, \theta_{A,R}), (R, \theta_{B,R}), (R, \theta_{C,R}), (T, \theta_{D,T}), (T, \theta_{E,T})]$ where $\theta_{X,Y}$ is the offset of attribute X in the tuples of relation Y.

For temporary tables being used for intermediate results, our approach works well. One complication arises, however, when the temporary tables are being used for bound tables in the rule system: Since the rule condition evaluation and the rule action occur in different transactions, and locks are not held across transactions, tuples referenced in the bound tables can change between the time when the bound table is computed and when it's read. If this were to happen, the bound tables seen by the rule action would no longer reflect the state of the database at the time of condition evaluation. To solve this problem, standard table records are not changed in place — a new record is created and linked into the relation. The old record is removed from the relation but kept in the system until the last bound table that references it is retired, as determined by a reference counting scheme.

The STRIP temporary tuple storage approach requires less memory than other schemes. Also, because data from standard tuples does not need to be recopied into temporary tuples, the amount of copying is minimized which improves system performance.

6.2 Task/Transaction Management

Tasks, not transactions, are the units of scheduling in STRIP. Tasks can be generated by user applications, the rule system, or the import/export system (described in [AKGM96b]). The flow of tasks in STRIP is shown in Figure 15, and is similar to other high performance and real-time databases such as [HSTR90]. Every new task to the system has associated with it a release time specifying when it should be run. Many user tasks have immediate release times and are placed in the ready queue. Some tasks, though, in particular those that will run unique transactions with delays, will have a future release time. These tasks are placed in the delay queue until their release time.

Tasks are serviced in STRIP by a pool of processes. Whenever a process becomes free, it moves a task from the ready queue to the running queue and starts executing its code. If, while running, a task must block waiting for a lock, it is placed in the blocked queue until the lock is granted. When a task finally finishes, results are returned to the user if appropriate and its process looks to the ready queue to find another task. STRIP provides standard real-time scheduling algorithms for tasks such as earliest-deadline and value-density first. Details of the algorithms are provided in [Ade96].

[1] Actually, the current version of STRIP stores a pointer for each joined table regardless of whether it contributes any attributes. The next version will contain the optimization described here.

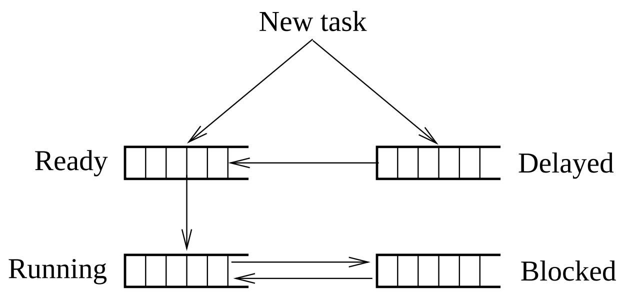

Figure 15: The flow of tasks within STRIP.

6.3 Rule processing

Rule processing in STRIP occurs at the end of a transaction.[2] At this time, the transaction's log is scanned to to see which events have occurred, and hence which rules have been triggered. If a rule is triggered, its transition tables are built during the log pass.

After the pass through the log, each triggered rule is considered in turn. First, its condition is checked. If the results are to be bound, a temporary table is built. If the condition evaluates to true, any other queries in the **evaluate** clause are computed and bound as well. Finally a task is created to perform the rule action. The task is passed the following information through its task control block (TCB):

1. a pointer to the schema of each of the bound tables it will have access to when it runs, as well as a pointer to the the data in the table,
2. the name of the user function it should run, and
3. the amount of time its release should be delayed after this transaction commits.

When all of the rules triggered by a transaction have been processed, all of the triggered tasks will be enqueued in the database. Those with non-zero delays will be placed in the delay queue and the rest in the ready queue.

Once running, a task triggered by a rule is different from other tasks in one way: it has access to bound tables as well as the other tables in the database. Thus, whenever a triggered task tries to access a table, its bound table list must be checked as well as the database catalog. When a triggered task finishes, its bound tables are no longer needed and are reclaimed. Recall that this involves freeing the memory used for the linked list of pointer arrays as well as any old records from standard tables that were only being kept for the bound table.

To extend the base rule system to support unique transactions, we needed a way to discover if a task is already queued in the ready or delayed queues with particular unique column values. To support this lookup, a hash table is built for each type of unique transaction. The hash table is used to hash the unique column values of a task to a pointer to its TCB. The hash table for a particular unique transaction is created when the first rule that executes the transaction is defined.

[2]For statements not performed within a transaction, rule processing occurs at the end of the task or before the next transaction starts, whichever comes first.

To illustrate the use of the hash table, consider one built for the user function `compute_comps3` from Section 3 which is defined unique on `comp`. Suppose a transaction T commits that has changed one stock price, and that the affected stock is used to compute two composites, c_1 and c_2. The bound table `matches` will have two rows, one for each composite. Transaction T will look in the hash table for `compute_comps3` to see if two new tasks have to be created to recompute the two composites. It may find that a task is already queued for c_1, so it adds the relevant row of its bound table to that stored in the TCB of the enqueued task. If c_2, however, does not have a hash table entry, a new task is created and placed in the delay queue. Then an entry is inserted into the table to map the name of composite c_2 to the TCB of the new task. When an enqueued task, T', finally begins to run, it removes its record from the hash table. This and all other modifications of the hash table are guarded by spinlocks.

7 Related Work

In this section, we discuss previous work related to STRIP's rule system. Due to space constraints, we do not discuss previous work on main memory databases other than referring the interested reader to [GMS92]. An overview of previous work in real-time databases is presented in [Ram93].

The results of Section 5 show that using unique transactions can greatly improve the performance of a rule-based system that maintains rapidly changing derived data. Here, we consider to what extent previously proposed rule systems can implement our unique transaction approach. To implement our approach, four requirements are placed on a rule 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 new transaction of the same type should be triggered.

3. **Storage of Transition Tables -** 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 -** As was shown in Section 5, using different units of batching for unique transactions can lead to very different CPU load versus recomputation length tradeoffs. Since the optimal choice is application dependent, the database must give the application designer control over this parameter.

While these four requirements are supported to different degrees by currently proposed rule systems, no system other than STRIP supports all of them. To discuss the various rule systems, we split them into two categories: those that support only *single events* and those that support *composite events*. Single events are usually database operations such as insert, update, etc. Composite events are built from single events using logical or temporal operators. Systems based on single events are far more prevalent and will be considered first. The reader is directed to [WC96] for more details.

7.1 Single event detection

No system with only single events meets all four requirements. Delayed execution is only supported by systems that can specify timing requirements on rule actions, such as HiPAC [DHL90]. Currently no commercial systems provide timing requirements. Unique execution is not supported directly by any proposed system, but can be emulated by using rule "deactivation," where present. 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 set of rules 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.

To our knowledge, no single event system provides direct support for storage of transition tables across transaction boundaries. To simulate this aspect of unique transactions, application programmers will have to create and maintain the information explicitly using additional tables. In addition to being less efficient, the workaround 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 control over the unit of batching.

7.2 Composite event detection

The proposed systems with composite event detection come much closer to meeting the four requirements. In some systems, it is possible to emulate delayed and unique execution by defining a composite event that is composed of one or more similar single events over an interval of time. For instance, an event could be defined as one or more changes to a stock price in the interval [time of first change, time of first change + delay]. The condition check and action can then be triggered by the composite event which will occur only once if the consumption mode is set properly. This approach requires relative temporal events which are not supported by every system (e.g., not by Ode [GJS92]). Also, since the single events can come from different transactions, we require event histories that span transaction boundaries, not supported by, e.g., Sentinel [CKAK94].

The next requirement is storage of transition tables. The best any proposed systems seem to support is event parameters, which are the values of the data that caused the event. For instance, when a stock price changes, the new and/or old value might be included as event parameters. In SAMOS [GD92], the user has to choose which value is used, although both can be supported with additional rules and use of versioning. Even given both values, there are still two shortcomings. First, only data involved in the event is stored and not data computed in the condition check: this is equivalent to limiting the unique columns to come from the transition tables in our model. Second, the single event parameters must be composed in a useful way for the composite events. Most systems do not provide all of the parameter values with the composite event. For instance in SAMOS, only the parameters of the first or last event in the interval are saved. Event parameters can also be used to control the unit of batching to a limited degree. Some systems, such as SAMOS and Ode, can constrain event parameters to be equal as part of composite event definition. In both systems, however, the number of single events must be known a priori, which is not possible in the systems we have studied.

To summarize, rule systems that support composite events come much closer to supporting the unique transaction approach than do those systems with only single events, but they still fall short.

8 Conclusion

In applications where base data changes rapidly, maintaining derived data is a critical component of overall system performance. In this paper, we have presented a rule system that allows very efficient maintenance of derived data in such environments through a new construct called *unique transactions*. Unique transactions allow changes to base data to be flexibly grouped to optimize the derivation functions: Changes can be both batched across transaction boundaries and partitioned by the attributes of the base and derived data. Through experimentation on a main memory DBMS with a special-purpose rule system, we have shown that batching changes significantly reduces the recomputation load on the CPU.

Unique transactions allow the tuning of two parameters: the unit of batching and the length of the delay window. As was shown in Section 5, having these two parameters provides great flexibility in managing the tradeoffs between performance, data timeliness, and schedulibility. It also, however, requires that someone or something choose appropriate values. The optimal choices for both parameters are heavily application dependent, but the results from Section 5 suggest two rules of thumb. First, the unit of batching should be chosen to be just large enough to take advantage of the redundancy in the recomputation but no larger. Second, increasing the size of the delay window yields diminishing returns so a small window should be chosen to begin and only lengthened if performance is not acceptable. Since most of the applications we have mentioned are characterized by a relatively static set of queries and predictable loads, gradual optimization of the two parameters is possible.

Finally, it is often argued that rules are difficult for humans to write and hence should be automatically generated. For example, in [CW91], the authors show how rules can be automatically derived to maintain a certain class of relational views given the view definitions. We are confident that this work can be extended to take advantage of unique transactions as well. By maintaining statistics such as join selectivities and how often tables are updated, it should be possible for a materialized view manager to derive not just the rules to maintain a view but the unit of batching and delay window size as well. This is a subject of future work.

References

[Ade96] B. Adelberg. *STRIP: A Soft real-time database for open systems*. PhD thesis, Stanford University, 1996.

[AKGM96a] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *Proceedings of EDBT*, pages 223–40, 1996.

[AKGM96b] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–7, 1996.

[BBKZ93] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an open system: The REACH rule system. In *Proceedings of the first international workshop on rules in database systems*, pages 111–26, 1993.

[BS73] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637–54, 1973.

[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.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.K. Kim. Composite events for active databases. In *Proceedings of the 20th VLDB Conference*, pages 606–17, 1994.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental rule maintenance. In *Proceedings of the 17th VLDB Conference*, pages 577–89, 1991.

[DD93] C.J. Date and H. Darwen. *The SQL standard*. Addison-Wesley, 3.0 edition, 1993.

[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.

[GD92] S. Gatziu and K.R. Dittrich. SAMOS:an active object-oriented database system. *IEEE Data Engineering Bulletin*, 15(4):23–6, 1992.

[GJS92] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th VLDB Conference*, pages 327–38, 1992.

[GMS92] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–16, 1992.

[HSTR90] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham. Real-time transaction processing: design, implementation and performance evaluation. Technical Report COINS 90-43, Univ. of Massachusetts, 1990.

[New94] New York Stock Exchange, Inc. *The TAQ database*, 3.0 edition, June 1994.

[PTV90] P. Pucheral, J. Thévenin, and P. Valduriez. Efficient main memory data management using the DBGraph storage model. In *Proceedings of the 16th VLDB Conference*, pages 683–95, 1990.

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

[Rou82] N. Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems*, 7(2):258–90, 1982.

[WC96] J. Widom and S. Ceri. *Active database systems*. Morgan Kaufmann, 1.0 edition, 1996.

A Unique transaction behavior

We formally specify the behavior of unique transactions. Assume that in its condition and its evaluate clause, a rule defines n bound tables $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. Let the set of columns of each table T_i be denoted $\mathcal{C}_i = \{c_{i1}, c_{i2}, \ldots, c_{im_i}\}$ and assume that each column name is unique across all tables. Let $\mathcal{U} = \{u_1, u_2, \ldots, u_p\}$ be the p unique columns chosen from $\bigcup \mathcal{C}_i$ and specified in the unique clause of the rule definition. We partition the set of tables $\mathcal{T}$ into two sets depending on whether any of their columns are defined as unique:

$$\mathcal{T}^u = \{T_i \in \mathcal{T} \mid (\exists c_j \in \mathcal{C}_i) c_j \in \mathcal{U}\}$$

$$\mathcal{T}^n = \mathcal{T} - \mathcal{T}^u$$

Further, let

$$B = \prod_{T_i \in \mathcal{T}^u} T_i, \text{ and}$$

$$unique\_cols(k_1, k_2, \ldots, k_p) = \pi_{u_1, u_2, \ldots, u_p} B$$

so that *unique_cols* is a relation containing all of the distinct combinations of the unique columns from the bound tables.

When a rule is triggered and its condition evaluated to true, the system will generate the relation *unique_cols* from its bound tables. Then, it will try to create a new transaction for each tuple in *unique_cols*. Each new transaction will receive only the portion of the bound tables related to its unique column values. If $(v_1, v_2, \ldots, v_p)$ are the unique values for one of the triggered transactions, the set of bound tables passed to it, $\mathcal{T}'$ will be:

$$T'_i = \begin{cases} T_i & \text{if } T_i \in \mathcal{T}^n \\ \pi_{\mathcal{C}_i} \sigma_{u_1 = v_1, u_2 = v_2, \ldots, u_p = v_p} B & \text{otherwise} \end{cases}$$

If a transaction with the same name and the same unique column values is already queued in the system, the tuples of each of its bound tables $T'_1, T'_2, \ldots, T'_n$ will be appended to the those of the bound tables of the enqueued transaction. If not, a new transaction is enqueued with the bound tables.

B Formula for Program Trading Examples

The formula to compute a composite average composed of n stocks is $comp\_price = \sum_{i=1}^{n} w_i p_i$ where p_i and w_i are the price and weighting of the ith stock respectively. For theoretical option prices, we use the Black-Scholes pricing model [BS73], which although known to undervalue options, is still commonly used and reported. The model computes the price of a call option according to the following formula:

$$p_{option} = p_s \Phi(d_1) - \frac{p_e}{e^{rt}} \Phi(d_2),$$

$$d_1 = \frac{\ln(\frac{p_s}{p_e}) + (r + \frac{1}{2}\sigma_{p_s}^2)t}{\sigma_{p_s}\sqrt{t}},$$

$$d_2 = \frac{\ln(\frac{p_s}{p_e}) + (r - \frac{1}{2}\sigma_{p_s}^2)t}{\sigma_{p_s}\sqrt{t}},$$

where

p_s	=	the current price of the underlying stock,
σ_{p_s}	=	the standard deviation of r, used as a measure of the riskiness of the stock,
p_e	=	the exercise price of the option,
r	=	the continuously compounded risk-less rate of return,
t	=	the time remaining before expiration expressed as a fraction of a year,
$\Phi()$	=	the cumulative distribution function for the standard normal function.