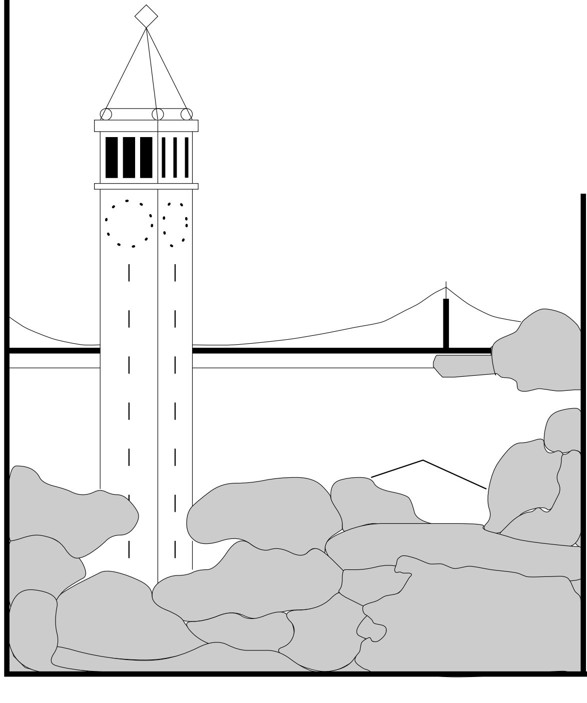
Continuous Query Processing with Real-Valued Functions

Matthew Denny *Michael J. Franklin*



Report No. UCB//CSD-4-1312
March 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Continuous Query Processing with Real-Valued Functions

Matthew Denny Michael J. Franklin

University of California, Berkeley
{mdenny,franklin}@cs.berkeley.edu

Abstract

Many analysis applications in areas such as finance and energy monitoring require the repeated execution of expensive modeling functions over streams of rapidly changing data. These applications can often be expressed declaratively, but current continuous query processing technology does not provide adequate performance in the presence of these expensive functions. For important classes of real-valued functions evaluated in predicates, we use Taylor approximations to determine ranges of stream inputs for which the system knows the outcome of the predicates for potential query results. We show how to integrate this technique into a prototype continuous query processor. We then report on experiments for a financial application using real bond market data. The experiments show that our techniques significantly reduce the number of function calls compared to traditional memoization.

1 Introduction

In many analysis applications, users need to execute expensive *real-valued* functions on streaming continuous data. We define *real-valued* functions as those that take at least one real number as input and produce real numbers as results. In bond markets, for instance, bond traders typically use *bond models* to determine a price for a given bond. Bond models output a price, usually as a real number, based on input data about the bond and current economic data. Bond traders need models because, unlike stock markets, complete and current bond price information is often unavailable. Traders often make deals over the phone, and the prices in these deals are not made public. As economic and bond data changes, bond traders may want to run data through their models in real-time, and receive alerts based on these results. Many bond trading applications can be expressed declaratively. Consider the following continuous queries:

```
/* Q1: Tell me when a bond with a 30-year maturity
   in my portfolio is worth more than C dollars */
SELECT BD.*,IR.*
FROM BondData BD, IntRate IR
WHERE model(BD,IR.rate) > C AND BD.maturity = 30yr
AND BD.numHeld > 0;

/* Q2: Tell me when a bond with a 30-year maturity
   not in my portfolio is worth more than a
   bond with a 30-year maturity in my portfolio */
SELECT BM.*,BP.*,IR.*
FROM BondData BP, BondData BM, IntRate IR,
WHERE model(BM,IR.rate) >  model(BP,IR.rate)
      AND BP.maturity = 30yr AND BM.maturity = 30yr
      AND BP.numHeld > 0 AND BM.numHeld = 0;
```

In these queries, BondData is a relation that contains a tuple for each bond in the market. IntRate is a stream of the current market interest rate, measured with the current yield on a 10-Year U.S. Treasury Bond as IntRate.rate. *model* is a bond model that takes a BondData tuple and a current interest rate in the form of a Treasury yield. Generally, bond data used in models changes infrequently [1], so we can represent it as a relation. On the other hand, there exists one 10-Year Treasury yield at a given time, and this yield changes many times per day. *model* takes a real value as input (*IntRate.rate*) and outputs a real price; thus, it is a real-valued function.

Unfortunately, many bond models compute prices based on expensive numerical methods, and some input data such as interest rate changes rapidly. For example, the authors of [9] needed a cluster just to run their model in experiments involving only a few bonds. Bond traders could not possibly run this model for every bond on the market each time interest rate data changes. For real-time applications, traders must either use cheaper models that include less information or use stale function results to answer queries. Neither of these solutions is ideal for applications where timely and accurate data are critical.

This work has been partially supported by the National Science Foundation undter the ITR grants IIS0086057 and SI0122559 and by research funds from IBM, Intel, Microsoft, and the UC MICRO program.

[1]See [14, 6] for more information on bond data used in bond models.

In addition to bond traders, many other users would like to run current data through expensive real-valued functions and receive alerts based on the result. For instance, power companies have models that predict how much power they will need in the future based on inputs such as weather conditions. As the weather changes, these companies would like to run continuous queries that notify them when specific grids may soon be short of power [5]. In addition, supply chain management applications may require continuous queries involving predictive models, especially with real-time inventory information that RFID tags will soon provide [10].

Although many of these applications can be expressed as continuous queries, current continuous query engines are of little help in minimizing expensive function calls. Most continuous query engines focus on the problems of high data rates or large data volumes (e.g. [11, 12, 13]). In most database literature that addresses expensive functions, the systems memoize function results[2]. If a system encounters a function call that has been run before, it can evaluate the function by simply consulting its cache. If the function inputs come from streams with fields in the real domain, however, the inputs may change rapidly, and the system may not encounter cache hits often. In our bond trading application, for example, economic data such as the interest rate changes by small amounts many times during the day. If a system does not often encounter the exact same stream values more than once, memoization by itself is not useful.

The problem with these caches is that they have no information about the function other than the cached values. Thus, these caches cannot form ranges around the inputs of cached values where anything about the function value is known. For a large class of real-valued functions, however, we can often obtain information on how a function result varies with an input value. For such functions, we can use *Taylor approximations* to find upper and lower bounds on the function result for stream input values other those that are cached [2].

To this end, we present a system that uses Taylor approximations to reduce the number of real-valued function calls made in a continuous query. Specifically, the system optimizes predicates in a query that contain real-valued functions with one stream field as input. In these predicates, the system uses Taylor approximations to compute a stream input range for which each potential query result either satisfies or does not satisfy the predicate. If the stream input values fall within these ranges, the system can output exact results for queries such as Q1 and Q2 without running functions each time the stream changes.

If functions are as expensive as the model in [9], a cluster is likely needed to compute these values. At many times, cycles may become available on these clusters. In finance, for example, the system does not have to process new interest rates or run any functions when markets are closed. If cycles become available on a cluster, the system can compute more functions to expand the ranges. With larger ranges, the future stream values are more likely to fall inside these ranges. The system schedules these function calls according to a heuristic that tries to minimize the number of future function calls that the system will have to make in real-time to evaluate the predicates. We currently have a working prototype that runs bond trading queries with real bond data, interest rates, and bond models. Our experiments show that the prototype requires significantly fewer online function calls than a system that only memoizes results.

[2]See [17] and Section 12.1 of the survey [16] for a discussion of memoization in databases.

2 General Approach

In this paper, we use queries Q1 and Q2 as running examples throughout. Consider the function *model* in Q1 and Q2. Assume *model* is any function that takes only information about a bond and a real interest rate as input, and returns a deterministic[3] real price (e.g. [6, 9, 26]). Also, assume that *model* has a continuous second derivative with respect to interest rate, and that the system has information on how the *model* result varies with interest rate. In this case, the system can use Taylor approximations to compute analytical minimum and maximum bound functions for *model* at all interest rates given one function result at a specific rate. These bounds are deterministic and conservative.

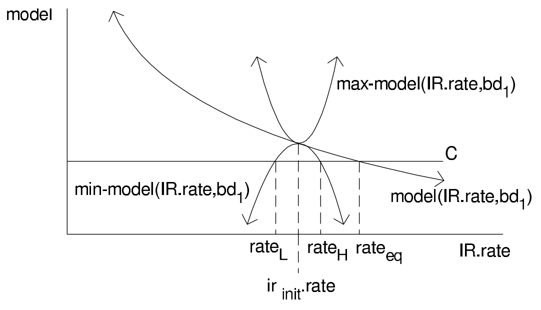


Figure 1: Plot of $model(IR.rate, bd_1)$ with Taylor Bounds

Figure 1 shows $model(IR.rate, bd_1)$ for some tuple $bd_1 \in$ BD from Q1. The system cannot determine $model(IR.rate, bd_1)$ at any given $IR.rate$ without an expensive function call. Suppose the system computes a result for $model(ir_{init}.rate, bd_1)$. The system can use this result with Taylor approximations to compute analytical minimum and maximum bounds on $model(IR.rate, bd_1)$ for all $IR.rate$. These bounds are shown in Figure 1: $max\text{-}model(IR.rate, bd_1)$ is the upper parabola, and $min\text{-}model(IR.rate, bd_1)$ is the lower parabola. Intuitively, these two functions use information on how *model* varies with *IR.rate* to determine a conservative bound on how much *model* can change as *IR.rate* moves away from $ir_{init}.rate$. Therefore, the system knows with certainty that the result of $model(IR.rate, bd_1)$ at

[3]Some bond models are stochastic, but we deal with only deterministic ones here.

any *IR.rate* is between *min-model*$(IR.rate, bd_1)$ and *max-model*$(IR.rate, bd_1)$. Section 3 describes in detail the computation of functions such as *min-model* and *max-model*, which we call *Taylor bounds* for *model*.

Suppose the system is running a continuous query with a selection predicate that includes a real-valued function. Assume this function has one real-valued stream input and meets the above continuity conditions. An example of such a predicate is $model(BD, IR.rate) >$ C in Q1. Figure 1 shows C as a horizontal line. In this figure, the system knows that $model(bd_1, IR.rate) >$ C for all *IR.rate* in the range $(rate_L, rate_H)$ because *min-model*$(IR.rate, bd_1)$ $>$ C in this range. As long as *IR.rate* values stay inside $(rate_L, rate_H)$ and bd_1 does not change, the system does not need to compute a function with bd_1 to evaluate the predicate.

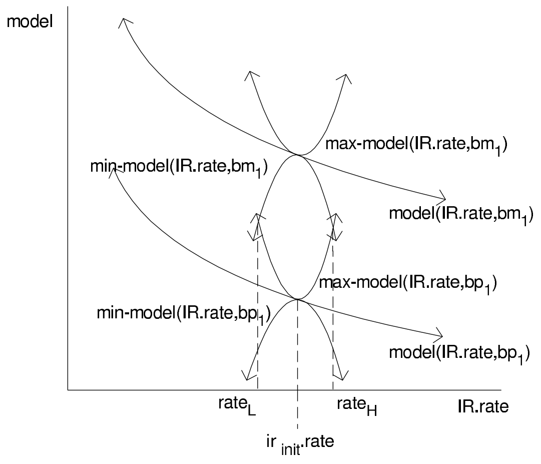


Figure 2: Plot of $model(IR.rate, bp_1)$ and $model(IR.rate, bm_1)$ with Taylor Bounds

The system can also find stream value ranges for queries with join predicates that include similar real-valued functions. In Q2, for example, assume that there exist tuples $bm_1 \in$ BM and and $bp_1 \in$ BP such that Q2 needs to evaluate $model(IR.rate, bm_1)$ $> model(IR.rate, bp_1)$. Figure 2 shows a plot of $model(IR.rate, bp_1)$, $model(IR.rate, bm_1)$, and their Taylor bounds computed with function results at ir_{init}. The system knows that bm_1, bp_1 satisfies the join predicate for the range where *min-model*$(IR.rate, bm_1) >$ *max-model*$(IR.rate, bp_1)$.

In addition to ranges where predicates are satisfied, the system finds ranges where predicates are not satisfied in a similar manner. In some cases, the system may not be able to find or expand a range. The system has provisions for handling these cases gracefully. As long as the stream values stay in the computed ranges, the system can return exact answers to queries such as Q1 and Q2 without running a single function. If some ranges do not contain the corresponding stream values, the system quickly finds these ranges and attempts to run the appropriate functions in real-time. If this is not possible, the system notifies the user of the potential query results that could not be evaluated. Our system currently supports continuous queries with real-valued functions in both simple selection and join predicates.

As mentioned in Section 1, the system may run functions on a cluster, which may have spare cycles at certain times of the day (e.g. non-market hours for financial systems). When compute cycles become available, then our system can iteratively compute more functions to expand these ranges. In order to expand a range, our system computes a new range and merges it to an existing range. Given a range $(rate_L, rate_H)$ in Q1 for which the system knows $f(IR.rate, bp_1) >$ C, the system forms a new range by solving at $model(rate_L, bd_1)$ or $model(rate_H, bd_1)$, and computing new Taylor bounds. If this range intersects $(rate_L, rate_H)$, the system merges the ranges. The system expands ranges for join predicates in a similar manner. The system has an efficient scheduler that attempts to minimize the number of future real-time function calls needed by intelligently choosing ranges to expand.

Section 3 describes our system in detail. To simplify our discussion, we first discuss our system in the context of a query with one selection predicate involving a real-valued function. In Section 4, we present the extensions needed to handle join predicates. Section 5 presents related work, and Section 6 concludes the paper.

3 Simple Selection Queries

3.1 Simple Selection Specification

This section explains the processing of a single continuous query such as Q1 that has one selection predicate involving a real-valued function. The system optimizes predicates with functions that take one real value from one input stream, and have a continuous second derivative with respect to this input. As in Q1, the predicate with the real-valued function can be in a boolean combination with other predicates that involve only relations. Relation tuples are allowed to change in our system. If a relation tuple is updated, however, ranges involving the updated tuple have to be recomputed. For the rest of this section, we assume that relation tuples are constant to simplify our explanation. Throughout this section, we explain these queries using Q1 as a running example. In this example, we assume that *IR.rate* has minimum and maximum possible values of RATE_MIN and RATE_MAX, respectively.

3.2 Range Computation

When the system begins processing Q1, it must find initial ranges for all tuples that must be evaluated by the selection predicate with the real-valued function. For this predicate, the system creates a set *relset*, which contains the combinations of tuples that must be evaluated by the predicate. *relset* is a subset of the cross product of the relations involved in the predicate. In the case of Q1, the only relation in *model(BD,IR.rate)* $> C$ is *BD*, so *relset* is a set of *BD* tuples. In Q1, these are the *BD* tuples such that $BD.maturity = 30$yr and $BD.numHeld > 0$. The calculation of *relset* from predicates containing only ranges requires straightforward static query processing, and is not discussed further here. For each $bd \in relset$ in Q1, the system computes ranges for *IR.rate* in which it knows the value of *model(bd,IR.rate)* $> C$.

In order to compute these ranges, the system must compute Taylor approximations. To do this, the system has to evaluate the function for each tuple in *relset* with an initial stream value. In Q1, we assume the interest rate stream only has one valid interest rate at any one time. That is, a new interest rate tuple invalidates the current one[4]. Let the interest rate tuple at the time the system begins running Q1 be ir_{init}. For each $bd \in relset$ in Q1, the system evaluates $model(ir_{init}.rate, bd)$. It also computes the first derivatives of *model* with respect to *IR.rate* at $ir_{init}.rate$ ($\frac{\delta model}{\delta IR.rate}(ir_{init}.rate, bd)$). In addition, we assume the system can compute a range for the second derivative for *bd* for all *IR.rate*, ξ_{bd}. For continuous real-valued functions, first and second derivatives can always be estimated using finite differencing with at most one or two extra *model* calls [2]. Also, a conservative bound on the second derivative is often easy to obtain for some functions, as we show in Section 3.5.

Given this information, Taylor's theorem states that *model(IR.rate,bd)* for any *IR.rate* is estimated by $model(ir_{init}.rate, bd) + \frac{\delta model}{\delta IR.rate}(ir_{init}.rate, bd)(IR.rate - ir_{init}.rate)$ with a conservative error bound of $\frac{1}{2}(\xi_{bd})(IR.rate - ir_{init}.rate)^2$ [2]. Thus, *min-model(IR.rate,bd)* and *max-model(IR.rate,bd)* in Figure 3 yield a conservative bound on the function $model(IR.rate, bd)$ for all *IR.rate*.

Given that many of these expensive functions use numerical techniques to begin with, the system must accommodate error in either $model(ir_{init}.rate, bd)$ or $\frac{\delta model}{\delta IR.rate}(ir_{init}.rate, bd)$. If the estimate for either value has significant deterministic error, then the system uses the lower error bound for these values in *min-model* and the upper error bounds for them in *max-model.* ξ_{bd} encapsulates any second derivative error because it is a conservative range. In the bond model used in our experiments, error is not large enough to affect performance. The rest of the paper does not discuss error, except where its presence affects the processing.

Figure 1 in Section 2 plots a sample function, $model(IR.rate, bd_1)$, as well as $max\text{-}model(IR.rate, bd_1)$ and $min\text{-}model(IR.rate, bd_1)$. As described in Section 2, the system knows that $model(IR.rate, bd_1) > C$ for all $IR.rate : rate_L < IR.rate < rate_H$ because $min\text{-}model(IR.rate, bd_1) > C$ between $rate_L$ and $rate_H$. The system computes $rate_L$ and $rate_H$ by solving $min\text{-}model(IR.rate, bd_1) - C = 0$ for *IR.rate*. Since $min\text{-}model(IR.rate, bd_1)$ is quadratic with respect to *IR.rate*, the system can easily solve this equation using the quadratic formula. Of course, there may be less than two solutions to this equation, which would mean that $model(IR.rate, bd_1)$ never crosses C on at least one side of $ir_{init}.rate$. If there is no solution less than or greater than $ir_{init}.rate$, then the *rate* range for which $model(IR.rate, bd_1) > C$ has an endpoint of either MIN_RATE or MAX_RATE, respectively.

If $model(ir_{init}.rate, bd_1)$ were less than C, the system would simply solve $max\text{-}model(IR.rate, bd_1) - C = 0$ to find a range where $model(IR.rate, bd_1) > C$ is not satisfied. The system processes queries with other selection operators similarly. In some cases, the system may not be able to find a range for bd_1 at $ir_{init}.rate$. These cases occur when a) $model(ir_{init}.rate, bd_1) =$ C, b) $model(ir_{init}.rate, bd_1)$ has error bounds that include C, or c) the computed range is narrower than some user-defined tolerance (call it MIN_RNG)[5]. In these cases, the system forms an *uncertain range* centered around $ir_{init}.rate$ with width MIN_RNG. An uncertain range indicates a range where the system does not know whether or not the predicate is satisfied. If an *IR.rate* value falls within an uncertain range for bd_1, the system must execute *model* at the new *IR.rate* to answer the predicate.

After finding ($rate_L$,$rate_H$) pairs for each $bd \in relset$, the system represents each range as a ($rate_L$,$rate_H$,*isSat*,*bd*) tuple, where isSat = {true,false,uncertain} depending on if the predicate is satisfied, not satisfied, or uncertain, respectively, in ($rate_L$,$rate_H$). With only one *IR* tuple in the stream at any time, all initial ($rate_L$, $rate_H$) ranges are computed with the same rate, and each range surrounds this rate.

To speed query processing, the system builds separate interval indexes (see [22]) on the tuples where isSat = true and isSat = uncertain. Call these indexes the *satisfied* and *uncertain* range indexes, respectively. When a new *IR* tuple ir_{new} enters the system, the system probes the satisfied range index with $ir_{new}.rate$ to find all ranges where ($rate_L$, $rate_H$) contains $ir_{new}.rate$. These ranges represent the *bd* tuples that satisfy the predicate for ir_{new}. The system then probes the uncertain range index with $ir_{new}.rate$ in a similar manner to find any *bd* tuples for which it needs to run *model* functions in real-time. If the system does not have the compute cycles to run all needed functions, it notifies the user of the *bd* tuples that it cannot evaluate at $ir_{new}.rate$.

If ir_{new} has a *rate* that does not fall in a

[4]The processing of streams with multiple tuples at any one time is a straightforward extension of the description in this section, unless otherwise noted.

[5]We found that for very small ranges, there was significant roundoff error when we calculated the endpoints. We set MIN_RNG = .0005 % in our experiments. This value is half the .001 % granularity that interest rates are reported at, and approximately 11 orders of magnitude greater than the precision of a double floating point number [2].

$$\begin{aligned}
\textit{min-model(IR.rate,bd)} &= model(ir_{init}.rate, bd) + \frac{\delta model}{\delta IR.rate}(ir_{init}.rate, bd)(IR.rate - ir_{init}.rate) + \\
&\quad .5 * min(\xi_{bd})(IR.rate - ir_{init}.rate)^2 \\
\textit{max-model(IR.rate,bd)} &= model(ir_{init}.rate, bd) + \frac{\delta model}{\delta IR.rate}(ir_{init}.rate, bd)(IR.rate - ir_{init}.rate) + \\
&\quad .5 * max(\xi_{bd})(IR.rate - ir_{init}.rate)^2
\end{aligned}$$

Figure 3: Taylor Bounds on *model(IR.rate,bd)*

$(rate_L, rate_H)$ range for some *BD* tuple $bd \in relset$, the system again either runs the function in real-time or notifies the user that it cannot evaluate ir_{new}, bd in the predicate. To detect such potential results quickly, the system uses data structures that we introduce in Section 3.4.

3.3 Range Expansion

As time passes, more cycles in a system may become available for computing functions, and the current *IR* tuple likely changes. If the *rate* field of the current *IR* tuple falls outside of the range or falls in the uncertain range for some tuple $bd \in relset$, the system uses these cycles to compute another range for *bd*. Otherwise, the system uses the compute cycles to efficiently expand the ranges that are not uncertain so that these ranges are more likely to contain future *IR.rate* values. The expansion algorithm handles uncertain ranges gracefully by only expanding them when necessary and actually shrinking them when possible.

First, we describe the expansion algorithm for *normal* ranges, or those ranges that are not uncertain. Fortunately, the nature of Taylor's Theorem allows the system to easily expand normal ranges. As shown in Figure 1, *min-model* and *max-model* provide conservative bounds on *model*, and the bounds get more conservative as *IR.rate* moves away from $ir_{init}.rate$. For a given range tuple, the system can get more accurate (and therefore wider) bounds by solving $model(rate_L, bd)$ and $model(rate_H, bd)$ along with their derivatives, and computing ranges around each of them. To obtain a larger value for $rate_H$, the system computes a range $(rate'_L, rate'_H)$ as described in Section 3.2, except it substitutes $model(rate_H, bd)$ for $model(ir_{init}.rate, bd)$, $\frac{\delta model}{\delta IR.rate}(rate_H, bd)$ for $\frac{\delta model}{\delta IR.rate}(ir_{init}.rate, bd)$, and $rate_H$ for $ir_{init}.rate$. Since the ranges are conservative bounds, the $rate_H$ field in the tuple can be replaced with the larger $rate'_H$. The system obtains lower values for the $rate_L$ field in a similar manner. Since expanding an $(rate_L, rate_H)$ range on one side requires a different function value and derivative than the other, the system can expand each side independently of the other.

Expansion of ranges in this manner works well as long as a $(rate_L, rate_H)$ range for some $bd \in relset$ does not approach an *IR.rate* where $model(IR.rate, bd) = C$. Consider Figure 1, where $rate_{eq}$ is defined such that $model(rate_{eq}, bd_1) = C$. Note that the distance between $rate_H$ and $rate_{eq}$ is less than the distance between $ir_{init}.rate$ and $rate_{eq}$. Thus, as $rate_H$ approaches $rate_{eq}$, there exists a smaller amount that $rate_H$ can grow such that $model(IR.rate, bd_1) > C$ is still true for all $IR.rate : rate_L < IR.rate < rate_H$. After enough expansions of $rate_H$, subsequent expansions will be small enough that they are essentially useless. If the system expands $rate_H$ for bd_1 and $rate_H$ grows by less than RNG_MIN, the system creates a new adjacent uncertain range $(rate_{L,U}, rate_{H,U})$ for bd_1 such that $rate_{L,U} = rate_H$ and $rate_{H,U} = rate_H + RNG\_MIN$. Creating an uncertain range is the only way the system has of expanding the range past $rate_{eq}$, and we describe this expansion below.

Given the conditions for the formation of an uncertain range, an uncertain range for tuple bd_1 in Figure 1 either contains or is near an IR.rate such as $rate_{eq}$ where the function crosses C. Note that in Figure 1, it appears as though a significantly large range could be formed to the right of $rate_{eq}$ where the predicate is not satisfied. Thus, the system's goal in expanding uncertain ranges should be to expand uncertain ranges only until the system can form an adjacent normal range. In the algorithm described below, the uncertain range expansion algorithm often actually shrinks ranges.

Consider the uncertain range $(rate_{L,U}, rate_{H,U})$ computed above for bd_1 that is adjacent to $(rate_L, rate_H)$. When the system attempts to compute a range past $rate_{H,U}$, it expands $rate_{H,U}$ to the right. Before doing so, however, it computes $model(rate_{H,U}, bd_1)$ as well as $\frac{\delta model}{\delta IR.rate}(rate_{H,U}, bd)$, and tries to compute a new normal range $(rate'_L, rate'_H)$ around $rate_{H,U}$. If the system succeeds in computing a new range with width greater than RNG_MIN, then the system sets this range adjacent to $(rate_{L,U}, rate_{H,U})$ by setting $rate_{H,U}$ to $rate'_L$. Since $rate'_L < rate_{H,U}$, the system can actually shrink uncertain ranges.

If the system cannot find a range $(rate'_L, rate'_H)$ with width greater than RNG_MIN, then it has no other choice but to expand the uncertain range. Since the system assumes no information about the predicate in uncertain ranges, it can increase $rate_{H,U}$ by any amount and the uncertain range is still correct. The current prototype uses the following multiplicative increase scheme: for the ith expansion of a given $rate_{H,U}$, set $rate_{H,U}$ to $rate_{H,U} + iRNG\_MIN$. Before each subsequent expansion, the system attempts to form a new normal range around $rate_{H,U}$ as described above.

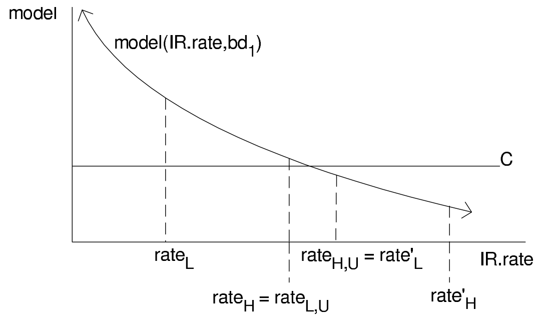


Figure 4: *model*($IR.rate, bd_1$) with 3 Ranges

Figure 4 shows what ranges might look like for bd_1 after multiple expansions to the right. If the system starts with $(rate_L, rate_H)$ and finds that $rate_H$ grows less than RNG_MIN, it creates an adjacent uncertain range $(rate_{L,U}, rate_{H,U})$. After zero or more expansions of $rate_{H,U}$, the system finds a normal range $(rate'_L, rate'_H)$ which is larger than RNG_MIN. In this case, it sets $rate_{H,U}$ to $rate'_L$ so that the ranges are adjacent. In our experiments, uncertain ranges often arise, but the system is able to shrink them to a point such that they are generally smaller than the granularity of the interest rate stream. Thus, they do not cause any real-time function calls in our experiments.

Given this algorithm, one may wonder why the system does not find $rate_{eq}$ more quickly by solving $model(IR.rate, bd_1) = C$ for $IR.rate$ using an iterative method such as Newton's method [2]. If the system can quickly find $rate_{eq}$ within some error tolerance, the system could easily form the ranges in Figure 4. If $model(IR.rate, bd_1) = C$ at more than one $IR.rate$, however, many solvers are only guaranteed to find one of these solutions without more information. Thus, the solvers may not return the closest solution to the range that the system is trying to expand. Also, solvers such as Newton's method require multiple calls to *model* that may not help expand the ranges. For instance, solving $model(IR.rate, bd_1) = C$ with Newton's method and initial guess $rate_H$ could require multiple *model* calls with bd_1 and an $IR.rate$ $> rate_{eq}$. If the system is trying to expand the range $(rate_L, rate_H)$, these calls are of no help.

3.4 Computation Scheduling

As new compute cycles become available, the system must determine a schedule for either expanding ranges or creating new ranges for Q1. To do this, the system builds a set of *ValidRanges* for each $bd \in relset$. A ValidRange for *bd* (we denote the ith ValidRange for *bd* by $vr_{i,bd}$) contains a set of adjacent ranges computed for *bd*. For example, the three ranges in Figure 4 compose a ValidRange for bd_1. The ValidRange keeps track of which ranges are uncertain and which are normal. The low and high endpoints of a ValidRange (denoted $vr_{i,bd}.L$ and $vr_{i,bd}.H$ for ValidRange $vr_{i,bd}$) are the low end of the lowest range and the high end of the highest range, respectively. Expanding a ValidRange is equivalent to expanding one of the ranges on either end of a ValidRange. With Q1, the system begins with one range per *bd* as described in Section 3.2, so the system begins with one ValidRange per *bd*. When a new range is computed for a tuple *bd* because an *IR.rate* value falls outside of its ValidRange, the system needs to keep more than one ValidRange for *bd*.

Given all the ValidRanges for each $bd \in relset$, the system needs a means to schedule computations for different $bd \in relset$. The system attempts to schedule computations so that the number of future function calls needed in real-time is minimized. However, the system does not know how much each ValidRange will expand given more computation, nor does it have information on future values of *IR.rate*. Thus, it cannot find a theoretical optimal schedule. Instead, it schedules on the heuristic presented below.

The following description of the heuristic works for streams which contain only one tuple at a time, such as an interest rate stream. The heuristic can be generalized to more complex streams, but we defer this more lengthy description to the appendix at the end of this technical report. The heuristic is greedy, meaning that the system invokes the scheduler to choose a function to compute every time the system has the resources to compute another function. Assume the current rate is $ir_{cur}.rate$. The system keeps two priority queues, each of which hold one ValidRange per $bd \in relset$. The ValidRange for a tuple *bd* in each queue is the ValidRange for *bd* that is either closest to or contains $ir_{cur}.rate$. When the system begins processing Q1, there is only one ValidRange per *bd* $\in relset$, so each queue contains the only ValidRange for each $bd \in relset$. One of the queues is sorted by the ValidRange low endpoints in descending order, and the other by the ValidRange high endpoints in ascending order. Call the ValidRanges at the top of these queues vr_{*,bd_L} and vr_{*,bd_H}, respectively. When the scheduler is invoked, it runs the following algorithm:

1. If $ir_{cur}.rate - vr_{*,bd_L}.L < vr_{*,bd_H}.H - ir_{cur}.rate$, dequeue vr_{*,bd_L}; else, dequeue vr_{*,bd_H}. Set $vr_{*,bd_{deq}}$ equal to the ValidRange dequeued.
2. If $vr_{*,bd_{deq}}$ contains $ir_{cur}.rate$, expand $vr_{*,bd_{deq}}$ in the direction of the endpoint that is closest to $ir_{cur}.rate$; else,
   (a) Compute $model(ir_{cur}.rate, bd_{deq})$ and $\frac{\delta model}{\delta IR.rate}(ir_{cur}.rate, bd_{deq})$
   (b) Compute a new ValidRange using these results, and set $vr_{*,bd_{deq}}$ equal to this ValidRange
3. If $vr_{*,bd_{deq}}$ intersects any other ValidRanges for bd_{deq}, merge these ValidRanges into $vr_{*,bd_{deq}}$.
4. Insert $vr_{*,bd_{deq}}$ on the queue from which the scheduler dequeued the ValidRange in step 1.

The example in Figure 5 both illustrates the scheduler and shows the intuition behind the heuristic. Assume three tuples in *relset*: bd_1, bd_2, and bd_3. Also assume that the 5 ValidRanges in Figure 5 exist. With $ir_{cur}.rate$ as shown in Figure 5, vr_{1,bd_1}, vr_{1,bd_2}, and vr_{1,bd_3} are on the queues. When the scheduler runs, vr_{1,bd_1} is vr_{*,bd_L} and vr_{1,bd_2} is vr_{*,bd_H}. Note that

vr_{1,bd_1} and vr_{1,bd_2} denote the two ranges that contain $ir_{cur}.rate$ but have the closest low and high endpoints, respectively, to $ir_{cur}.rate$. Thus, the system expands the ValidRange with the closest endpoint to $ir_{cur}.rate$, which is vr_{1,bd_1}

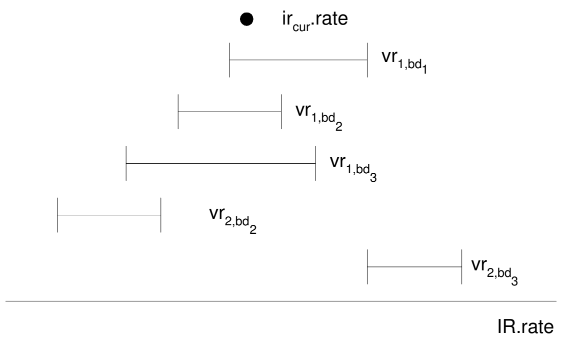


Figure 5: Example ValidRanges

Such a scheduler is efficient under the assumption that *IR.rate* is *approximately continuous.* That is, two consecutive values of *IR.rate* are somewhat close to each other, rather than exhibiting large jumps in value on a regular basis. With the exception of relatively rare market shocks, many financial processes such as interest rates follow this pattern [14]. If *IR.rate* is approximately continuous, then *IR.rate* is likely to move outside of vr_{1,bd_1} or vr_{1,bd_2} before it moves outside of vr_{1,bd_3}. Thus, expanding either vr_{1,bd_1} or vr_{1,bd_2} is likely the best option for minimizing future real-time computation. Now, suppose that vr_{1,bd_2} and vr_{1,bd_3} do not exist; consequently, vr_{2,bd_2} and vr_{2,bd_3} are on the queues instead of vr_{1,bd_2} and vr_{1,bd_3}. In this case, vr_{*,bd_L} is now vr_{2,bd_3}, vr_{*,bd_H} is now vr_{2,bd_2}, and the system computes a new ValidRange for vr_{2,bd_3}. Again, the system is making a good choice; vr_{2,bd_3} is further from $ir_{cur}.rate$ than the closest ValidRange for any other *bd* tuple. If *IR.rate* is *approximately continuous*, such computation should often reduce real-time computation and cache misses.

This system works well as long as the ValidRange on the queues for each tuple *bd* is the ValidRange for *bd* that is either closest to or contains the current $IR.rate$. Occasionally, a new IR tuple (call it ir_{new}) has a *rate* such that a ValidRange for some *bd* not on a queue now either contains $ir_{new}.rate$ or is closer to $ir_{new}.rate$ than the current ValidRange on a queue. To handle these cases, the system updates the queues in a lazy manner. Before the system invokes the scheduler, the system scans all ValidRanges with the bd_L or bd_H tuples before processing. If the system finds another ValidRange for either tuple that should go on the queue instead, it dequeues the appropriate ValidRange, enqueues the ValidRange found, and makes this scan again with the tuples from the ValidRanges now at the top of the queues. Only when the system scan finds that the ValidRanges at the top of the queues do belong there does it start the above scheduling algorithm.

Since this scheduler runs every time the system has resources to compute another function, it must run efficiently. Using heaps for the queues, the ValidRange enqueue and dequeue each require a number of operations that is logarithmic on the queue size [8]. The queues also provide an efficient data structure for finding *bd* tuples without a ValidRange that contains the current $IR.rate$. The system finds these tuples by walking each queue from the top and finding all ValidRanges until it finds a ValidRange that contains *IR.rate*. In heaps, This walk can be performed with an in-order traversal which requires operations linear on the number of ValidRanges returned [8].

3.5 Performance

To evaluate the performance of our system, we ran a prototype with a query similar to Q1, using real data and bond models. In these experiments, we show that a) our system requires significantly fewer function calls than a system using only memoization, and b) our system still performs well in experiments designed to stress our system.

The query we run is identical to Q1, except it runs the predicate *model(BD,IR.rate)* $> C$ on the entire BD relation. In our experiments, *BD* is a table consisting of 1668 Freddie Mac Gold PC 30-year mortgage-backed securities issued between January and December of 1993 [6]. *IR* is a stream describing the 10-Year Constant Maturity U.S. Treasury yield, with *rate* being the yield itself [15].

Unfortunately, we were only able to find a stream of Treasury yields at the granularity of a day. This did not allow us to test how our system reacts to intraday changes in the market. To simulate intraday changes, we used a *Brownian bridge* to construct a random path of yields between the yields for two consecutive days given a certain interest rate *volatility*. Details on implementing Brownian Bridges for financial processes can be found in [19]. We use a volatility for interest rates that reflect real market conditions as a default from [9], but we also vary this parameter in our experiments. The experiments here use interest rate data from January 3 to January 31, 1994, generating an intraday interest rate for every minute of market time. In keeping with the convention of most financial data providers (e.g. [7]), interest rates are reported at the granularity of .001 %. Many models used exhibits high error at high interest rates or interest rates near 0, so we set MIN_RATE to 0.5% and MAX_RATE to 100%. In the recorded history of the 10-year Treasury yield from 1800, the yield never leaves this range [15].

Although we have pointed out examples of expensive bond models [9, 26], these models proved too expensive for our simulations. For instance, pricing one bond and derivative at a \$.01 error with the model in [26] requires work which took a minimum of over 26 minutes on a Pentium 4 2.4 Ghz PC with 1.2 GB

[6]We are greatly indebted to Nancy Wallace at U.C. Berkeley's Haas School of Business for helping us to obtain this data.

of RAM. Although this model can be parallelized, we did not have a dedicated cluster to run large numbers of experiments for long periods of time. Therefore, we used a much cheaper model based on the Cox-Ingersoll-Ross interest rate model [6]. For each bond, the model outputs a first derivative and corresponding error by using finite differencing. For this model, we know a priori that the second derivative always decreases with interest rate. Thus, we can get a second derivative range ξ_{bd} for each bond tuple *bd* by determining the second derivative via finite differencing at MIN_RATE and MAX_RATE. Since the ranges should be conservative, we widen each range on both sides by 10 times the estimated error from the finite difference method. We use these ranges as a default, but vary them in some of our experiments.

The prototype and experiment code was written in C++. All experiments were run on a Pentium 4 2.4 GHz PC with 1.2 GB of RAM running RedHat Linux 7.3. In all experiments presented here, the system begins with values and derivatives for all bonds at the last interest rate before the first day of the experiment data (Dec. 31, 1993). The system then processes the interest rate stream, and is allowed a limited number of calls per day (*numCallsPerDay*) to expand or compute new ranges. We assume an 8 hour market day, so for 8 hours of simulated time the system processes one interest rate per minute and is allowed $\frac{1}{3}$ *numCallsPerDay*, equally spaced out over the day. After the simulated market closes, the system can use the remainder of *numCallsPerDay* to create or expand ranges. All experiments report *%CallsNeeded*, the cumulative percentage of functions at each interest rate where the system failed in evaluating a predicate with the function. This occurs when the system a) could not use its ranges to evaluate the predicate, and b) did not did not have the resources to run the function in real-time.

Before starting our performance study, we ran tests to calibrate the selection predicate constant C. In these tests, we set C to different quantiles of the initial bond values. Setting C to be the median of all initial bond values yields the highest *%CallsNeeded.* As Section 3.3 explains, when $model(ir_{cur}.rate, bd)$ is close to C, very little benefit occurs from range expansion for *bd*. With C set to the median of the bond values, the most bonds have ranges that expand slowly. For all experiments, we set C to be the median of all initial bond values.

Range Caching	Only Memoization
1092	2401930

Table 1: *criticalNumCallsPerDay* for Systems Using Range Caching and Only Memoization

In our first set of experiments, we determine the *numCallsPerDay* needed to evaluate all predicates in the query with our data. These experiments report *criticalNumCallsPerDay*, which is the smallest *numCallsPerDay* where $\%CallsNeeded = 0$ and the system can evaluate all predicates. Since $\%CallsNeeded$ varies inversely with *numCallsPerDay*, we can find *criticalNumCallsPerDay* using an iterative solver based on bisection method with an error of 2 *numCallsPerDay* [2].

The results are shown in Table 1 for both our system (Range Caching) and an engine using only memoization. Note that *criticalNumCallsPerDay* is three orders of magnitude larger for a system with only memoization than for our system. The system with only memoization exhibits such poor performance because it uses cached function values only if it encounters the exact interest rate more than once. In an environment where the interest rate is moving frequently by small amounts, memoization alone is of little help.

Given that we are using real market data, these experiments are likely indicative of how much compute power each system would need to run this application in the current market. Thus, a system with memoization could not support this query if models are as expensive as [26]. On the other hand, our system exhibits a more reasonable *criticalNumCallsPerDay*, and probably can support this application with a reasonably sized cluster running the functions.

vol Mult	900-Ra	1800-Ra	2700-Ra	2700-Me
1	.00001	.00000	.00000	.95296
2.5	.00063	.00009	.00000	.96437
5	.00233	.00111	.00050	.98090
10	.00496	.00291	.00163	.98941
15	.00755	.00429	.00277	.99191
20	.01029	.00556	.00391	.99323

Table 2: *%CallsNeeded* vs. *volMult*

In the following experiments, we vary experiment parameters that cause more stress for our system. Table 2 shows *%CallsNeeded* for various interest rate volatilities for 900, 1800, and 2700 function calls per day (900-Ra,1800-Ra, and 2700-Ra resp.). Volatility is measured by *volMult*, the multiple of the default volatility discussed earlier. Although the volatility where volMult = 1 was determined in [9] to reflect real market conditions, increasing the volatility gives us the opportunity to make the interest rate less continuous. An increase in volatility should result in less effective scheduling and thus the need to compute more real-time function calls. As a baseline, we compare our results against *%CallsNeeded* for an engine using only memoization with 2700 calls (2700-Me).

First, note that the 2700-Me still performs much worse than even the 900-Ra experiment at any volatility. At lower volatilities, our system evaluates most or all predicates, and the range of volatilities with all predicates evaluated grows as the number of functions computed per day rises. At higher volatilities, *%CallsNeeded* is still low, especially for larger *numCallsPerDay.*

In addition to varying volatility, we also ran experiments where we instead varied the size of each ξ_{bd} range for each bond *bd*. Increasing this range should make the *min-model* and *max-model* parabolas shown in Figure 1 thinner, which results in smaller ranges. In these experiments, we multiply the upper bound of each ξ_{bd} by *xiMult* and the lower bound by $-xiMult$[7]. The results show that the system is only marginally sensitive to the sizes of each ξ_{bd} for *xiMult* from 1 to 100, and thus we do not show the data here.

3.5.1 Performance Summary

From our experiments, We draw two conclusions. First, a system with memoization likely requires too many function calls to run our bond trading selection query under real market conditions if functions are even moderately expensive. However, our system requires a small enough number of function calls that the system can probably run the query given reasonable compute resources. Second, our system degrades gracefully when we vary parameters that should stress the system, and it still drastically outperforms a system with only memoization.

4 Simple Join

In this section, we extend the description from the last section to accommodate similar queries with a join predicate involving real-valued functions. To do this, we use Q2 from Section 1 as a running example. Q2 only has one predicate with real-valued functions, *model(BM,IR.rate) > model(BP,IR.rate)*. The techniques used to process Q2 can support any query with one join predicate which involves at least one real-valued function. The functions do not need to be the same, so long as they satisfy the conditions described in Section 3 and take the same real-valued stream input.

Section 4.1 explains the basic processing for joins as a straightforward extension of the processing discussed in Section 3. Section 4.2 discusses an optimization on this basic processing, and Section 4.3 presents performance experiments.

4.1 Range Computation, Expansion, and Scheduling

In many ways, the processing of join predicates is similar to the processing of selection predicates. When the system starts running Q2, it computes a set *relset* which contains pairs of tuples from the set BM $\times$ BD that satisfy the other predicates in the query. For each bm, bd tuple in *relset*, the system then computes *IR.rate* ranges where the predicate is satisfied, not satisfied, or uncertain. The system uses these ranges to evaluate the predicates and schedule new computations in the same way it does in Section 3. Join predicate processing differs with selection predicate processing only in the computation and expansion of ranges.

[7]Since actual second derivative ranges are always positive, we multiply the low ends by a negative number to expand the range.

Suppose the system starts running Q2 with an initial IR tuple ir_{init}. In this case, The system computes *model* and its derivative at $ir_{init}.rate$ for each *bm* and *bp* in a *relset* pair. With this data, the system computes Taylor bounds for each *bm* and *bp*. Figure 2 in Section 2 plots two sample functions $model(IR.rate, bm_1)$ and $model(IR.rate, bp_1)$, as well as the corresponding Taylor bounds functions, evaluated at $ir_{init}.rate$. The system knows that $model(IR.rate, bm_1) > model(IR.rate, bp_1)$ for the $IR.rate$ range $(rate_L, rate_H)$, where $min\text{-}model(IR.rate, bm_1) > max\text{-}model(IR.rate, bp_1)$. The system finds $rate_L$ and $rate_H$ by solving $min\text{-}model(IR.rate, bm_1) - max\text{-}model(IR.rate, bp_1) = 0$. If $model(ir_{init}.rate, bm_1)$ and $model(ir_{init}.rate, bp_1)$ were a) equal, b) within one another's error bounds, or c) used to compute a range of width less than RNG_MIN, the system would create an uncertain range around $ir_{init}.rate$ with width RNG_MIN.

After initialization, the system can expand normal ranges for *bm,bp* pairs in *relset* by computing a new range and merging it with an existing range, similar to the description in Section 3.3 The system also expands uncertain ranges as described in Section 3.3, with the system attempting to form an adjacent normal range before each expansion. Just as Section 3.3 showed expansions creating a small uncertain range around $rate_{eq}$ where $model(rate_{eq}, bd_1 = C$, the system expands ranges so that it creates an uncertain range around $IR.rate$ values where $model(IR.rate, bm_1)$ and $model(IR.rate, bp_1)$ cross.

When computing new ranges after initialization, the system may avoid calling functions by using cached computed function values. For example, suppose the system needs to compute a range for bm_1, bp_1 at $rate_{new}$ some time after the ranges are initialized. Since initialization, the system may have computed *model* for bm_1 and bp_1 at different rates when computing or expanding other ranges involving bm_1 and bp_1. When the system performs *model* computations, it memoizes the function results and derivatives. When the system has to compute a new range for bm_1, bp_1 at $rate_{new}$, it finds cache entries for bm_1 and bp_1 that have the closest $rate$ to $rate_{new}$. The system uses these cached values to compute Taylor bounds, and attempts to compute a range. Since the cached values may have been computed at different times, however, either cached $rate$ value may be different than $rate_{new}$. Because of these potential inequalities, the computation of ranges is slightly different.

Let $rate_{c,bm_1}$ and $rate_{c,bp_1}$ be rates closest to $rate_{new}$ where the system has cached *model* and derivative values for bm_1 and bp_1, respectively. Using the cached values corresponding to $rate_{c,bm_1}$ and $rate_{c,bp_1}$, the system computes Taylor bounds for bm_1 and bp_1 as before. Given that neither $rate_{c,bm_1}$ nor $rate_{c,bp_1}$ may be equal to $rate_{new}$, the system has to determine if the predicate is satisfied or not at $rate_{new}$. If $min\text{-}model(rate_{new}, bm_1) > max\text{-}model(rate_{new}, bp_1)$, the system knows the predicate is satisfied at $rate_{new}$. If $max\text{-}model(rate_{new}, bm_1) \leq min\text{-}model(rate_{new}, bp_1)$, the system knows the predicate is not satisfied. If neither of these inequalities hold, the cached values are of no use, and the system must compute *model* and its derivative at $rate_{new}$ for bm_1 and bp_1. If one of these inequalities holds, the system finds a range using the Taylor bounds as described above. If the system finds a large enough range that contains $rate_{new}$, it does not have to call *model* for bm_1 or bp_1 at $rate_{new}$.

4.2 Tolerance Optimization

Unfortunately, algorithms in Section 4.1 do not perform well when the functions involving many of the *bm* and *bp* tuples are nearly identical to one another. Consider the case where the functions involving bm_1 and bp_1 are nearly identical, as shown in Figure 6. With the current *IR* tuple ir_{init}, the system according to Section 4.1 attempts to compute the range $(rate_L, rate_H)$. If the functions have intersecting error bounds or subsequent expansions grow the range less than RNG_MIN, then the system creates an uncertain range and makes future expansions accordingly. For nearly identical functions, uncertain range expansions using the multiplicative scheme eventually yield ranges that encompass both RATE_MIN and RATE_MAX. Thus, the system must run *model* for almost any *IR.rate* in order to answer the predicate for bm_1, bp_1. Unfortunately, nearly identical bonds are common in our data, and thus present a significant problem.

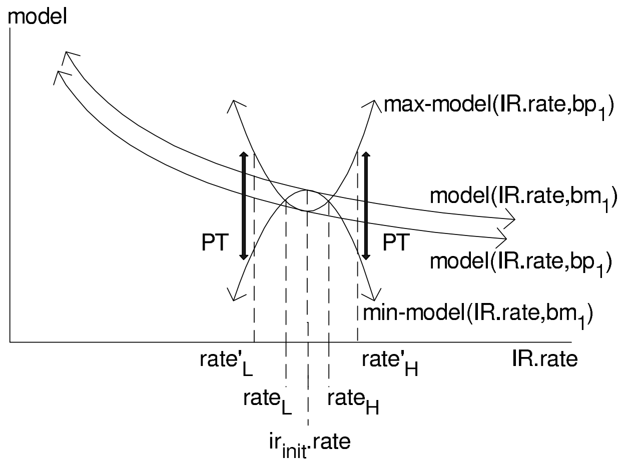


Figure 6: Plot of nearly identical $model(IR.rate, bm_1)$ and $model(IR.rate, bp_1)$ with predicate tolerance PT and Taylor bounds (only shows 2 bounds for clarity)

To handle this case, our system takes advantage of the fact that many predicates only need to be evaluated within a certain *tolerance*. In the bond trading example, bond models typically output real numbers that are rounded to the machine's precision. As long as the query outputs the correct answer within \$.01 of the bond prices, however, the traders do not care. We define a tolerance for each predicate PT as the maximum margin by which the system may report an incorrect answer. For instance, if a query reports that $model(ir_{init}.rate, bm_1) > model(ir_{init}.rate, bp_1)$, the user knows that $model(ir_{init}.rate, bm_1) - model(ir_{init}.rate, bp_1) > -PT$. Similarly, if it reports $model(ir_{init}.rate, bm_1) \leq model(ir_{init}.rate, bp_1)$, the user knows that $model(ir_{init}.rate, bm_1) - model(ir_{init}.rate, bp_1) \leq PT$.

With PT defined, the system only slightly modifies its range computations. In the example in Figure 6, the system instead finds the range $(rate'_L, rate'_H)$ where $model(ir_{init}.rate, bm_1) - model(ir_{init}.rate, bp_1) > -PT$. The system computes $rate'_H$ and $rate'_L$ by solving $min\text{-}model(ir_{init}.rate, bm_1) - max\text{-}model(ir_{init}.rate, bp_1) = -PT$. As shown in Figure 6, ranges are much larger than the ones computed in Section 4.1, which means that the system is less likely to create an uncertain range.

With PT defined, the system expands ranges as described in Section 4.1, with one small modification. When the system evaluates the functions at, say, $rate'_H$, it may find that $model(rate'_H, bm_1) < model(rate'_H, bp_1)$. This case can occur because the predicate can actually now change values within a normal range. In this case, the system just computes a new normal range $(rate''_L, rate''_H)$ with a different *isSat* value by solving $min\text{-}model(rate'_H, bp_1) - max\text{-}model(rate'_H, bm_1) = -PT$. The system sets this range to be adjacent to $rate'_H$ by setting $rate''_L$ to $rate'_H$.

Note that predicate tolerance cannot be set in a system that only uses memoization. Since a cache of only function values has no knowledge of derivatives or how a function varies, it cannot put any bounds on the input values it caches and guarantee that a predicate is true within a certain tolerance. Such a guarantee is important in applications like arbitrage, where price differences of even a few cents can be critical.

4.3 Performance

To test our join algorithm, we ran our prototype on a query similar to Q2 in the same experiments as presented in Section 3.5. The query we run is identical to Q2, except it runs the join predicate over the entire cross product of the bond data relations. We use the same data, bond model, and simulation environment as Section 3.5, except that we need two tables of bond data. To this end, we split the bond data we have evenly between the two tables. With default parameters from Section 3.5, we ran the simulated interest rate streams over different bond workloads and measured *%CallsNeeded*. To create each workload, we sorted the bonds by model output at the initial interest rate. While each table contains one half of the total data set, we vary the percentage of tuples that each table takes from the top half and the bottom half of the list.

The results show that the highest *%CallsNeeded* occurs when each table takes an equal number of tuples from each half of the list. This workload has the highest number of tuples in different tables with initial function values that are close to one another, so the ranges are smaller. Since this workload stresses our system the most, we use it for the remainder of our experiments.

Range Caching ($0 PT)	Range Caching ($.01 PT) [8]	Only Memoization
119353	0	2401930

Table 3: *criticalNumCallsPerDay* for Systems Using Range Caching and Only Memoization

In our first set of experiments, we determine *criticalNumCallsPerDay* for this query as we did for the selection query in Section 3.5. Table 3 shows *criticalNumCallsPerDay* for our system with $0 and $.01 predicate tolerance (PT), as well as a system using only memoization [9]. With PT = $0, our system still outperforms an engine with only memoization, but we see worse performance than with the selection query in Section 3.5. This drop in performance is due to the join predicate evaluating a large number of join pairs with large uncertain ranges.

With PT = $.01, *criticalNumCallsPerDay* = 0 for our system, which means the system can evaluate all predicates in the experiment with only the initial ranges it has for the bonds. This result is even better than the *criticalNumCallsPerDay* for the selection query in Section 3.5 because the nonzero PT effectively eliminates uncertain ranges at the default parameters. Also, we set the selection constant in Section 3.5 such that the most bonds have slowly expanding ranges at the initial interest rate.

Given that we are using real market data, we see that, with the correct tolerance, our system is likely to support the join query in real market conditions. In the future, we plan to build a predicate tolerance into simple selections as well to see if we can reduce *criticalNumCallsPerDay* for these queries as well.

vol Mult	900-Ra	2700-Ra	900-Pt	2700-Pt	2700-Me
1	.0233	.0228	.0000	.0000	.9530
2.5	.0430	.0295	.0000	.0000	.9643
5	.1457	.0586	.0074	.0007	.9809
10	.3417	.1899	.1037	.0304	.9894
15	.4749	.3006	.2394	.1196	.9919
20	.5717	.3997	.3517	.2080	.9932

Table 4: %CallsNeeded vs. Volatility

[8] Note that the system computes initial ranges for each join pair before the first market day of experiment begins. 0 *criticalNumCallsPerDay* means that the system was able to evaluate all predicates with these initial ranges.

[9] Since both the selection and join queries need to run each bond on each interest rate, the engine using only memoization has the same *%CallsNeeded* for both queries with all other inputs being equal. We repeat these numbers from Section 3.5 for convenience.

For the remainder of our experiments, we vary experiment parameters that cause more stress for our system. To make the interest rate stream less continuous, we run the join query with our data and different interest rate volatilities, indicated by the volatility multiplier *volMult*. Table 4 shows the *%CallsNeeded* from these experiments for 900 and 2700 *numCallsPerDay*, where PT = $0 (900-Ra and 2700-Ra) and with PT = $.01 (900-Pt and 2700-Pt). As a baseline, we again run the join query with only memoization at 2700 *numCallsPerDay* (2700-Me). As *volMult* rises, *%CallsNeeded* rises more quickly for our system (both Ra and Pt) than it does in the experiments for the selection query in Section 3.5. As *volMult* rises, we see more interest rates that are far from the initial interest rate. At these interest rates, the ranges in the selection query expand much more quickly than they do near the initial interest rate because of the way we set the selection constant. In the case of the join query, the ranges do not expand more quickly. Despite this performance drop, even the 900-Ra experiment for the join query at *volMult* = 20 still outperforms the 2700-Me experiment at *volMult* = 1. As expected, the Ra experiments exhibit a higher *%CallsNeeded* than the Pt experiment at the same *numCallsPerDay* and *volMult*.

xi Mult	900-Ra	2700-Ra	900-Pt	2700-Pt	2700-Me
1	.0245	.0229	.0000	.0000	.9530
5	.0728	.0289	.0000	.0000	.9530
10	.1193	.0485	.0000	.0000	.9530
25	.2587	.1064	.0601	.0030	.9530
50	.4313	.1647	.1622	.0276	.9530
100	.5633	.2130	.3609	.0928	.9530

Table 5: %CallsNeeded vs. *xiMult*

In addition to varying volatility, we also vary the size of each second derivative range by varying the parameter *xiMult* as we did in Section 3.5. Table 5 shows results with similar configurations to those in Table 4, except we vary *xiMult* instead of *volMult*. As described in Section 3.5, larger second derivative ranges should result in Taylor bounds such that the system computes smaller ranges and more uncertain ranges. Unlike the selection query, the join query seems much more sensitive to the second derivative range, especially where *PT* = $0. Larger second derivative ranges result in Taylor bounds that compute smaller ranges. Thus, the system creates a large number of uncertain ranges for more bond pairs at higher *xiMult*. When *PT* = $.01, the system creates far fewer uncertain ranges, and the system only exhibits nonzero *%CallsNeeded* at the highest *xiMult* values.

4.3.1 Performance Summary

We see that our system requires a small enough number of function calls such that the system can probably run our bond trading join query in real market conditions given reasonable compute resources. However, we may need to specify a small predicate tolerance to reduce the number of function calls. Again, the system with only memoization likely requires too many function calls to run this query if functions are even somewhat expensive. In response to parameters that stress the system, our system still outperforms the system using only memoization. However, our system is more sensitive to these parameters here than when it runs a selection query. In any experiment, a small nonzero predicate tolerance significantly reduces the number of needed function calls.

5 Related Work

As stated before, most continuous query research [11, 12, 13] does not concentrate on expensive predicate evaluation. Work on expensive predicate evaluation such as [3, 21, 18] focuses on static query optimization and either assumes that function memoization occurs or does not mention it at all. Work in [17] focuses on function caches within the execution of one static query. *Persistent caches* store results across multiple static queries. Section 12.1 of the survey [16] provides a bibliography of ideas on such caching. The function indexes in [23] provide similar functionality as memoization. None of this work attempts to compute ranges around memoized input which can be used to process queries. Query processing over *approximate predicates* is discussed in [25], where approximate predicates are cheaper versions of exact predicates with known false positive and false negative probabilities. Our system functions in situations where such predicates do not exist.

The concept of predicate tolerance is similar to the *precision constraint* in [24]. The precision constraint is the amount of error that a user will tolerate in query results. This work deals with minimizing communication costs in aggregate continuous queries over distributed data sources, which is a significantly different problem than we consider here. The work in [4] deals with probabilistic queries over streaming data sources by representing the data sources as probability distributions. To run our queries probabilistically, we would need both a probability distribution of the stream data as well as information on how each function invocation transforms that distribution. Our system assumes neither of these.

In the numerical analysis literature, there is a wide body of work on solvers and interpolation schemes for real-valued functions with one parameter [2]. These techniques provide information about a function without solving it at every parameter value. To our knowledge, none of this work finds a range of parameter values for a function where a predicate is satisfied. Some work in the optimization field also deals with real-valued functions [20], but the techniques are specific to optimization problems.

6 Conclusion

This paper deals with the problem of executing continuous queries with expensive real-valued functions in the predicates. If a real-valued function in a predicate takes a field from a stream as input, the system may have to recompute each function every time the stream changes. If the stream field is a real number and changes frequently, the system may not frequently encounter the same value twice. In this case, traditional memoization is not helpful.

To this end, we present a system that reduces the number of function calls in continuous query predicates for a large class of real-valued functions. Using Taylor approximations for the functions, the system computes ranges of stream inputs in which a predicate value is known for each potential query result. If the stream values fall in these ranges, the system does not have to compute any functions to evaluate the predicate. If the system has spare compute cycles, it can compute more functions to incrementally expand these ranges, thus increasing the chance that future stream values fall in the ranges. The system schedules these function calls using a heuristic that attempts to minimize the function calls needed in real-time to evaluate the predicates in the future.

To evaluate our system, we built a prototype and ran experiments with bond trading queries using real data. In these experiments, our system requires far fewer function calls than a system using only memoization. In fact, the number of calls required by a system using only memoization likely prohibits supporting our workload for even moderately expensive functions. When we vary experiment parameters designed to stress our system, our system still significantly reduces the number of function calls needed over a system with only memoization.

7 Acknowledgments

David Liu, Yanlei Diao, and Wolf-Tilo Balke all provided insightful comments on earlier drafts of this paper. Discussions with Joe Hellerstein in the early stages of this work were very helpful in shaping our direction. David Bindel and Andrew Zimdars provided valuable comments on the mathematical foundations of this work, and Amol Deshpande was kind enough to help us in our related work search.

A Appendix: Processing for More Complex Streams

Although Sections 3 and 4 deal with continuous query processing where streams contain one tuple at a time, the system can easily accommodate different streams with only slight modifications. Here, we describe the system running the query Q1 with a significantly more liberal definition of stream. The processing of queries with join predicates (e.g. Q2) with such streams follows in a straightforward manner from the discussion here. We leave the processing of even more complicated streams (e.g. streams with windows) as future work.

Instead of the definition in Section 3.1, let IR be a stream of tuples with two fields, *id* and *rate*. *id* is a key on a tuple such that a stream contains at most one tuple with a given key at any given time. When a new tuple enters the stream, it updates all the fields of the current tuple with the same key. Here, we assume the set of keys is constant. For instance, a current stock quote stream is a similar stream, where the key is the ticker symbol. Although it is unlikely that an interest rate stream has more than one tuple at any given time[10], we assume here that IR has more than one tuple so that we can use our running example of Q1.

Since $|IR| > 1$ at any given time, note that $f(IR.rate, BD) > C$ is now basically a join predicate. *relset* now contains a subset of tuple pairs from $IR \times BD$ that satisfy all other predicates in the query. Since all other predicates in $Q1$ contain only the relation BD, *relset* only changes if either one of the BD tuples or one of the $IR.id$ key values changes. If the tuples in *relset* change, the system must recompute the ranges associated with the updated tuple pairs.

First, the system needs to compute ranges for every *ir,bd* pair in *relset*. While the system may need a maximum of $|relset|$ *model* calls, it will probably need less because some of the *ir.rate* values for *ir* tuples paired with a given *bd* tuple will fall into previously computed ranges. The system forms and expands these ranges as described in Section 3.2 and Section 3.3. For each *ir* tuple, the system keeps an uncertain and satisfied range index just as it does in Section 3.2. When new tuple enters the system and updates a tuple *ir*, the system probes the satisfied index for *ir* to find *bd* tuples that satisfy the predicate with the new *ir.rate*. The system also probes the corresponding uncertain index to find *bd* tuples for which the system needs to run *model* in real-time. The system also uses a modified scheduler to schedule function computations, as explained below.

Let the definition of ValidRange for each ir, bd tuple pair be analogous to the definition of ValidRange for each *bd* tuple in Section 3.4 (denote the ith ValidRange of the ir, bd pair as $vr_{i,ir,bd}$). Let *irset* be the set of *ir* tuples in IR that are in a *relset* tuple pair. Instead of keeping only 2 ValidRange priority queues, the system keeps 2 queues for each $ir \in irset$. The queues for the tuple *ir* contain a ValidRange for each tuple pair in *relset* that contains *ir*. For each *ir* tuple, one of the queues is sorted by the ValidRange low endpoints in descending order, and the other is sorted by the ValidRange high endpoints in ascending order. For a given ir, bd tuple pair, the system keeps the ValidRange on the *ir* queues that either contains or is closest to *ir.rate*. As in Section 3.4, the system invokes the scheduler to choose a function computation each time new cycles become available. The new scheduling algorithm, shown below, is a variation on the original algorithm that accommodates 2 queues for each $ir \in irset$. For this algorithm, let vr_{*,ir,bd_L} be the ValidRange at the top of the queue for the tuple *ir* that is sorted by the ValidRange low endpoints. Similarly, let vr_{*,ir,bd_H} be the ValidRange at the top of the queue for the tuple *ir* that is sorted by the ValidRange high endpoints. When the system has compute cycles available, the scheduler runs the following algorithm.

1. Find vr_{*,ir_L,bd_L} such that $ir_L = min_{ir \in irset}(ir.rate - vr_{*,ir,bd_L}.L)$. Also, find vr_{*,ir_H,bd_H} such that $ir_H = min_{ir \in irset}(vr_{*,ir,bd_H}.H - ir.rate)$.
2. If $ir_L.rate - vr_{*,ir_L,bd_L}.L < vr_{*,ir_H,bd_H}.H - ir_H.rate$, dequeue vr_{*,ir_L,bd_L}; else, dequeue vr_{*,ir_H,bd_H}. Set $vr_{*,ir_{deq},bd_{deq}}$ equal to the ValidRange dequeued.
3. If $vr_{*,ir_{deq},bd_{deq}}$ contains $ir_{deq}.rate$, expand $vr_{*,ir_{deq},bd_{deq}}$ in the direction of the endpoint that is closest to $ir_{deq}.rate$; else,
   (a) Compute $model(ir_{deq}.rate, bd_{deq})$ and $\frac{\delta model}{\delta IR.rate}(ir_{deq}.rate, bd_{deq})$
   (b) Compute a new ValidRange using these results, and set $vr_{*,ir_{deq},bd_{deq}}$ equal to this ValidRange
4. If $vr_{*,ir_{deq},bd_{deq}}$ intersects any other ValidRanges for ir_{deq}, bd_{deq}, merge these ValidRanges into $vr_{*,ir_{deq},bd_{deq}}$.
5. Insert $vr_{*,ir_{deq},bd_{deq}}$ on the queue from which the scheduler dequeued the ValidRange in step 2.

To understand the intuition behind this scheduler, consider the basic scheduler in Section 3.4 where $|IR| = 1$ ($IR = \{ir_{cur}\}$). If all ValidRanges in the queues in the basic scheduler contain $ir_{cur}.rate$, we showed that the scheduler expands the ValidRange with the closest endpoint to $ir_{cur}.rate$. If at least one ValidRange does not contain $ir_{cur}.rate$, then the scheduler computes a new range for the ValidRange on the queues which is furthest from $ir_{cur}.rate$. The new scheduler exhibits similar behavior, except ValidRanges have different *ir* tuples, and the scheduler considers each ValidRange in terms of its own *ir.rate* instead of the one $ir_{cur}.rate$. This heuristic works well if a) for all $ir \in irset$, *ir.rate* is approximately continuous, and b) all *ir.rate* have the similar variances, so that it is no more important to expand ranges around any single *ir.rate*. If assumption b does not hold, then the scheduler can compensate by using differences in standard deviations instead of scalar distances to find $vr_{*,ir_{deq},bd_{deq}}$ in steps 1 and 2 above.

The system needs to ensure that the ValidRange for a given ir, bd pair on a queue is the ValidRange for ir, bd that either contains or is closest to *ir.rate*. The system ensures this condition for each queue the same way it does in Section 3.4. When a tuple *ir* is updated, the system can find the *bd* tuples with no ValidRanges that contain *ir.rate* by walking down the queues for the *ir* tuple. As in Section 3.4, the system walks down these queues until it finds a ValidRange that contains *ir.rate*.

If $|IR|$ is small, the scheduler above is efficient. If $|IR|$ is large, the scheduler may need to keep many

[10]While there are many different measures of the interest rate, most models that take one interest rate only use one such measure.

more ValidRanges in queues than the scheduler described in Section 3.4. Given the amount of research in the area of disk-based sorting (see [1] and the references therein), we are confident that we can design efficient queues that spill to disk. Since our current workloads do not require queues that spill to disk, we leave this design as future work. The scheduler could keep an index on the top of each of these queues to speed the search for $vr_{ir_{deq},bd_{deq}}$, but the index would have to be modified when any IR tuple is updated.

References

[1] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. P. Patterson. High-performance sorting on networks of workstations. In *SIGMOD*, 1997.

[2] R. L. Burden and J. D. Faires. *Numerical Analysis.* Brooks/Cole, 2001.

[3] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *VLDB*, 1996.

[4] R. Cheng, D.V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.

[5] L Clewlow and C. Strickland. *Energy Derivatives : Pricing and Risk Management.* Lacima, 2000.

[6] L. Clewlow and C. Strickland. *Implementing Derivatives Models.* Wiley, 2000.

[7] Cnn/money.com. http://www.cnnmoney.com/.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[9] C. Downing, R. Stanton, and N. Wallace. An empirical test of a two-factor mortgage valuation model: Do housing prices matter? *Working Paper, UC Berkeley*, 2002.

[10] E. Dyson and E. Dean. Rfid: Logistics meets identity. *Release 1.0, Vol. 21, No. 6*, 2003.

[11] R. Carney et al. Monitoring streams - a new class of data management applications. In *VLDB*, 2002.

[12] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[13] S. Chandrasekaran et. al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[14] F. J. Fabozzi. *Bond Markets, Analysis and Strategies.* Prentice Hall, 2000.

[15] Global financial data. http://www.globalfindata.com/.

[16] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys 25(2)*, 1993.

[17] J.M. Hellerstein and J. Naughton. Query execution techniques for caching expensive predicates. In *SIGMOD*, 1996.

[18] J.M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, 1993.

[19] P. Jackel. *Monte Carlo Methods in Finance.* Wiley, 2002.

[20] J.E. Dennis Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* SIAM, 1996.

[21] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *SIGMOD*, 1994.

[22] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[23] D. Maier and J. Stein. Indexing in an object-oriented dbms. In *Workshop on Object-Oriented Database Systems*, 1996.

[24] C. Olston, J. Widom, and J. Jiang. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[25] N. Shivakumar, H. Garcia-Molina, and C. S. Chekuri. Filtering with approximate predicates. In *VLDB*, 1998.

[26] R. Stanton. Rational prepayment and the valuation of mortgage-backed securities. In *Review of Financial Studies Vol. 8, No. 3*, 1995.