Adaptive Execution of Variable-Accuracy Functions *

Matthew Denny
U.C. Berkeley, EECS Dept.
387 Soda Hall
Berkeley, CA 94720-1776, USA
mdenny@cs.berkeley.edu

Michael J. Franklin
U.C. Berkeley, EECS Dept.
387 Soda Hall
Berkeley, CA 94720-1776, USA
franklin@cs.berkeley.edu

ABSTRACT

Many analysis applications require the ability to repeatedly execute sophisticated modeling functions, which can each take minutes or even hours to produce a single answer. Because of this expense, such applications have largely been unable to directly use such models in queries, with either on-demand or continuous query processing technology. Query processors are hindered in their ability to optimize expensive modeling functions due to the "black box" nature of existing user-defined function (UDF) interfaces. In this paper, we address the problem of querying over sophisticated models with the development of VAOs (Variable-Accuracy Operators). VAOs use a new function interface that exposes the trade-off between compute time and accuracy that exists in many modeling functions. Using this interface, VAOs adaptively run each function call in a query only to an accuracy needed to answer the query, thus eliminating unneeded work. In this paper, we present the design of VAOs for a set of common query operations. We show the effectiveness of VAOs using a prototype implementation running financial queries over real bond market data.

1. INTRODUCTION

1.1 Motivation

Many important applications require the repeated use of expensive analysis functions. For example, power companies use models that predict power usage based on variable inputs such as weather conditions. These companies need to run queries using analysis functions to determine the weather conditions that would cause different parts of their grids to become overloaded [5]. As another example, consider securities traders who use numerical models to price securities based on market data. In order to monitor continuous query results involving these prices, traders must re-run the models as the underlying market data changes. A further example of such applications is in the area of supply chain management (SCM), where users will soon be able to run inventory replenishment models in real-time in response to data provided by emerging RFID technology [12].

Unfortunately, such sophisticated applications often have serious performance problems. In many of these applications, the models require minutes or even hours to compute a single data point, even on modern processors (e.g. [28, 11]). Thus, function execution can easily become a bottleneck. For streaming systems where the function arguments are data streams, the problems are exacerbated; A system may not have the processing power to keep up with the incoming stream updates. At present, analysts have to choose between using less complex (and hence, less accurate) models and running the models less frequently. Neither option is optimal.

If these functions are run in the context of a query, the query processor may be able to reduce the compute cost. While today's query processors attempt to avoid expensive function calls by either predicate re-ordering [23, 3, 22, 21] or caching [20, 7], such work fails to address the remaining problem of optimizing the *execution* of the function calls that still have to be run. As a result, the applicability of those solutions is limited. In this paper, we address this problem via a new query processing approach called VAOs (Variable-Accuracy Operators). VAOs are based on the insight that many modeling functions, (such as those implementing certain numeric functions) allow a trade-off between compute time and accuracy. VAOs are built upon a new interface to User Defined Functions (UDFs) that provides the system with finer-grained control over function execution, and thus more opportunity for optimization. That is, in contrast to current systems, where UDFs export a "black box", all-or-nothing interface, VAOs are able to adaptively vary the compute time in functions where if more work is applied, a more accurate answer is obtained.

VAOs perform common query operations (e.g. predicate evaluation, aggregates) that require the execution of expensive functions. In a query plan, VAOs replace both the module that executes a function, as well as the operator that evaluates the result. For the example query plan in Figure 1 for a MAX aggregate over a function result, a single VAO would replace both the function execution and aggregation modules shown. Using the new UDF interface, the VAO adaptively adjusts the amount of work done by the function according to the accuracy needed by the given operation.

*This work was funded in part by NSF under ITR grants IIS-0086057 and SI-0122599, by the IBM Faculty Partnership Award program, and by research funds from Intel, Microsoft, and the UC MICRO program. Denny was supported in part by the Siebel Scholars Fellowship.

VLDB '06, September 12-15, 2006, Seoul, Korea.

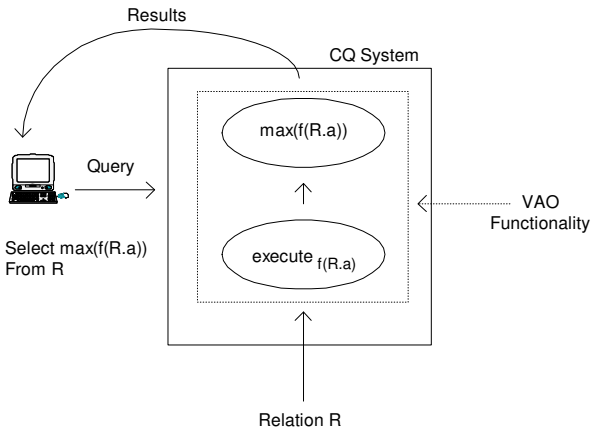

Figure 1: An example system running a simple MAX query. A MAX VAO combines the function execution with the aggregate calculation, enabling adaptive, incremental execution of the expensive function.

In our figure, the VAO needs to allocate work so that it accurately determines the largest value produced without performing unneeded work on function executions that ultimately produce smaller values. As we will show later, these VAOs often yield drastic performance improvements.

1.2 Example Application

To both demonstrate the need for VAOs and illustrate the VAOs approach, we detail the bond trading application from above, which we will use as a running example throughout the paper. As mentioned above, traders often use *bond models* to find a price for a given bond. For bonds that do not trade on an open market, pricing data is often not made public, and traders must use models to obtain prices. These models output a bond price based on input data about the bond and current economic data such as interest rates. As economic and bond data changes, bond traders may want to run models on each bond in real-time, and answer queries such as:

Q1: Find all bonds priced above $100.
Q2: Find the value of my bond portfolio, which is a weighted sum of bond prices.
Q3: Find the best performing (i.e. highest valued) bond.

In these queries, models must execute quickly because traders need to run a model for each bond issue each time an input changes. In the case of interest rate inputs, the rate is typically calculated using the price of a U.S. Treasury Bond, which changes every 2 minutes on average[1]. Therefore, models must be run quickly in order to be practical.

Unfortunately, bond models such as [11] can be computationally expensive, requiring time on the order of minutes or more. These models require numerical solutions for partial differential equations (PDEs) that cannot be solved analytically. These numerical PDE solvers return approximate prices, where the accuracy of the resulting price depends on the amount of compute work used in the solver.

[1]Determined by observing real-time interest rate data on [30] from 10/13/04 to 11/09/04.

While we concentrate on bond models in this example, similar PDE solvers are used in fields as varied as fluid mechanics [29], semiconductor process design [13], and high-energy physics [24][2].

Similar to the processing shown in Figure 1, queries Q1-Q3 can be run in CQ engines with the models supplied as UDFs. Given this architecture and "black box" UDF interface, however, CQ engines cannot control the accuracy of UDF calls. Therefore, these engines must always run models so all answers are accurate enough to answer *any* query. In financial applications, this means running all models with an error of less than $.01. Since prices can only be accurate to $.01 anyway, models can effectively report a price as a single real number.

In many cases, these systems often do too much work to process a query. For instance, consider a system running query Q3 over 2 bonds which are worth $105 and $95, respectively. Suppose that a model call reports both bond values within $.01 accuracy. In this case, the system could determine the max value without running the lower valued bond to as high of accuracy, thus requiring much less work.

1.3 Overview

To deal with this problem, we present VAOs, which are operators that combine the function execution and the operations over the results. By combining these two operations, the VAO can change the function execution based on the operations performed on the results.

VAOs use a UDF interface which lets them control the work-accuracy trade-off inherent in many functions. Using this interface, functions return upper and lower bounds, not single values. The initial bounds from a function are initially very coarse, as they result from the minimal amount of compute work for the function. If the bounds are not accurate enough to produce an answer for the operator, the VAOs can use the new interface to refine the bounds, which also requires more CPU cycles. A wide variety of numeric functions have an inherent work-accuracy trade-off, and we have modified a variety of numeric algorithms to accommodate the VAO interface.

For a VAO MAX operator in the above example, suppose the functions provide initial bounds of [$98, $110] and [$90, $101]. Since these bounds overlap, the operator must refine the bounds so that a) maximum value is found, and b) the value is within a certain error tolerance (e.g. $.01 for bonds). As the VAO can make refinements over either (or both) bounds, each VAO needs a refinement strategy that attempts to conserve work by considering both query operation the data involved.

We have designed VAOs for selection predicates and 4 aggregates. To evaluate our designs, we implemented prototype VAOs and ran experiments using real bond data and models. Under realistic market conditions, these experiments show that VAOs run functions up to two orders of magnitude faster than traditional operators. In additional experiments on synthetic data, we found that VAOs are robust in many experiments explicitly designed to stress VAOs.

[2]While a survey of PDE solvers can be found in Chapter 12 of [2], we discuss PDE solvers in more detail in Sections 2 and 4.

1.4 Contributions and Roadmap

The contributions of this paper are as follows:

- We describe a new UDF interface which exploits the trade-off between work and accuracy inherent in many expensive functions.
- We discuss our modifications to a large class of numeric algorithms which allows them to be implemented with the VAO interface.
- We present a new class of continuous query operators, the VAOs, which use the new interface. VAOs adjust the work in a function according to the accuracy needed by the query.
- We discuss experimental results with VAO prototypes. In experiments using real bond data and models, VAOs provide up to two orders of magnitude improvement over traditional operators. With synthetic data, VAOs exhibit robust performance in many experiments designed to stress VAOs.

The rest of this paper is as follows. Section 2 presents related work. Section 3 gives a general overview of query processing with VAOs, as well as a detailed description of the new UDF interface. Section 4 describes the modifications needed for numeric algorithms implemented with the VAOs interface, and Section 5 discusses the designs of specific VAOs. Section 6 discusses performance results, and Section 7 concludes the paper.

2. RELATED WORK

While previous database research deals with expensive function optimization, this prior work attempts to avoid function calls instead of optimizing the execution of calls that must be made. Therefore, most of this work is complementary to VAOs. Work on static queries concentrates either on predicate re-ordering [3, 22, 21] or caching [20]. Most continuous query research [14, 15, 16] does not concentrate on expensive predicate evaluation. The work in the TCQ system [16, 23], however, uses a query processing mechanism called an Eddy, which can potentially re-order predicates to avoid expensive function calls.

Our work in [7] presents CASPER, a caching system for expensive functions in continuous query systems. CASPER caches *predicate result ranges*, which are ranges of parameters where the results of expensive predicates are known. To compute these predicate result ranges, CASPER uses a new UDF interface. We view the integration of VAOs with CASPER, which entails integrating their UDF interfaces, to be interesting future work.

In addition to UDF optimizations, the database community has done significant work on using approximate answers to reduce the cost of expensive operations, which is the general approach of VAOs. Much of this work, however, uses probabilistic techniques which require specialized assumptions about the data and the expensive operations. For example,the *approximate predicates* presented in [27] are cheaper versions of exact predicates with known false positive and false negative probabilities. VAOs function in situations where neither such predicates nor the corresponding probabilities exist. Other approximation techniques [4, 9, 10] require probability distributions over the data. To adapt these techniques to the queries presented here, a probability distribution would be needed that relates both the underlying data and the outputs of the functions. VAOs require no such distribution.

The online aggregation work in [19] uses a principle similar to VAOs of continually refining error bounds until the answer has similar accuracy. The online aggregation system computes probabilistic error bars for aggregates by sampling relational data, and does not support user-defined functions. In contrast to this probabilistic approach, the systems in [26, 25] compute approximate aggregation queries by using deterministic error bounds. This work deals with aggregates over data coming from a large number of distributed data sources, and is focused on reducing communication cost of data transfer rather than compute cost.

In the scientific computing literature, there is a wide body of work on efficient solvers for expensive numerical functions. Many of these solvers, including those for PDEs, ordinary differential equations, and numerical integration problems, have the VAOs property that more work gives more accurate answers. As mentioned above, these solvers are used in fields as disparate as fluid mechanics [29], semiconductor process design [13], and high-energy physics [24]. A survey of these solvers and their applications can be found in any numerical analysis textbook (e.g. [2]), and the literature is much too vast to cite here.

In the scientific computing literature, the most related work to VAOs is *Adaptive Mesh Refinement* for PDEs [1]. As we explain later in Section 4, many PDE solvers require the creation of a mesh, or multidimensional grid, of values which determines both the accuracy and compute time of a solution. Adaptive mesh refinement iteratively changes the mesh size in order to find a solution of acceptable accuracy while conserving compute cycles. These techniques are designed primarily to optimize the solution to a single numerical function, and do not address the execution of multiple functions needed by a declarative continuous query.

3. GENERAL OVERVIEW

In this section, we give an overview of both VAOs and the new UDF interface that they use. VAOs change the processing of queries with any UDF that a) returns a real number, and b) has an inherent trade-off between work and accuracy. In this section, we first describe traditional operators processing UDFs with "black box" interfaces, and then show how VAOs improve upon these operators using the new interface.

3.1 Traditional Operators

Figure 2 shows how traditional operators process results of UDFs. In this figure, we show a system processing a selection predicate "$model(IR.rate, BD) > 100$", which would be a predicate similar to that found in Q1 from Section 1. In this predicate, BD is a relation containing a tuple for each bond in the market, and IR is a stream that contains the interest rate in the field $IR.rate$. $model()$ is a function that takes an interest rate and a BD tuple, and returns a price for a bond at the given rate. In this example, *model()* is based on PDE solvers which require more compute time for higher accuracy. Section 4 provides more information on PDE solvers commonly used in bond models.

In this figure, the system first executes *model()* in a function execution module, and then evaluates the result with

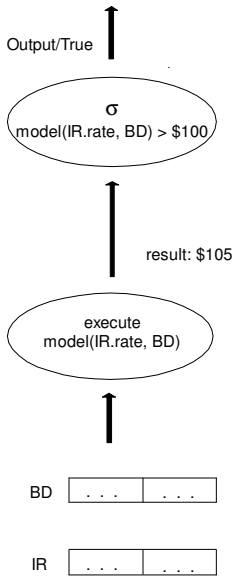

Figure 2: Evaluation of $model(IR.rate, BD) > \$100$ for a sample tuple pair with traditional operators.

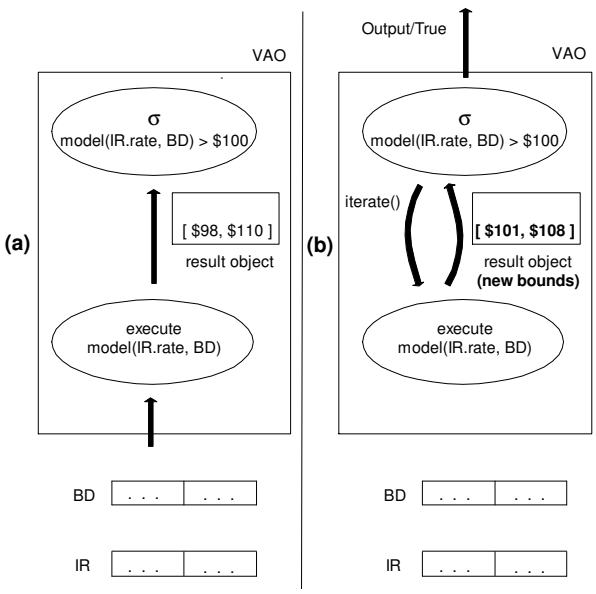

Figure 3: Evaluation of $model(IR.rate, BD) > \$100$ for a sample tuple pair with VAOs. (a) shows the function returning a result object, and (b) shows the VAO iterating over the object to refine the bounds.

a selection operator. For simplicity, we assume the absence of caching; function caches as described in [20] can be used with both traditional operators and VAOs, and do not affect our discussion of function execution.

Note that the selection operator is separate from the execution module, and has no control over UDF execution. Therefore, the function call costs are completely dependent on the function itself and its arguments. A function always runs with the same accuracy, which must be sufficient to answer any predicate. In our example, *model()* determines each price to an accuracy within \$.01. Since any error below \$.01 in a price is effectively irrelevant, the price returned can be used in any operator evaluation.

In many cases, however, *model()* does not need to return a value with such high accuracy. For the *model()* value of \$105 in Figure 2, *model()* could have been run with any accuracy within \$5, and the predicate would still evaluate to true. Since the selection operator has no control over function execution, *model()* always runs with \$.01 accuracy and the corresponding cost. Even if the operator could change function accuracy, the operator does not know the accuracy needed *a priori* for each set of input tuples. While an accuracy of \$5 may be enough for the *model()* result in Figure 2, results for other IR, BD tuple pairs may be closer to the predicate constant and require higher accuracy.

3.2 VAOs

Figure 3 shows a system with VAOs executing the same predicate described above. Here, the inputs to the function flow into the VAO, which is responsible for executing the function and applying the predicate. With this architecture, the VAO can use the selection operator to influence the function execution.

To control the function execution, VAOs take advantage of an *iterative* interface for user-defined functions. With this interface, VAOs can iteratively increase the accuracy of function results by using more CPU cycles. For many functions, particularly those of numerical nature, this interface works quite well. For example, many numerical solutions for root finding and integration[3] are based on iterative techniques, and thus can be implemented in this interface. Even for functions that are not iterative in nature, such as PDE solvers, we can often still use the VAO interface. Section 4 details how such solvers can be implemented with this interface.

With the iterative interface, the first call to a UDF returns a *result object* to the VAO instead of a value. Each object provides:

- H and L: Numeric data members which are high and low error bounds, respectively, for the function value.
- *iterate()*: A member function which the VAO can call to refine the bounds, at the cost of more CPU cycles.
- *minWidth*: A numeric data member which indicates the bounds width ($H - L$) under which the answer is considered as accurate as possible and no more *iterate()* calls should be run.

Unlike traditional operators, VAOs operate over result object bounds instead of single values, and they can iteratively refine these bounds in order to obtain an answer. All result object processing is encapsulated in VAOs, unless function results or result aggregates are in the operator output. In this case, the query also needs to specify a *precision constraint*, which is a maximum bounds width for the output. Precision constraints have been used in other query processing work, such as [25, 26].

Figure 3(a) shows initial bounds for an example result object. Each object initially provides very coarse-grained

[3]Elementary iterative root finders are surveyed in Chapter 2 of the textbook [2]. The same book also covers iterative numerical integration techniques in Chapter 4.

bounds that require the minimal compute time for a function. In this case, the bounds encapsulate the predicate constant, and the predicate result is unknown. In this case, the VAO sends the object back to the executor, which iterates over the object to get more accurate bounds.

A selection VAO iterates until either a) the bounds no longer contain the selection constant, or b) the bounds width falls below *minWidth*. The latter condition is a stopping condition needed by most iterative techniques. Without this, a VAO could iterate over an object infinitely many times, eventually resulting in infinitely small error bounds [4]. Here, the *minWidth* for all *model*() results is $.01. If the bounds still contain the constant and have width less than *minWidth*, the operator considers the function value equal to the constant, and produces the appropriate result.

In Figure 3(b), we show the result of an iteration over our example object. The new bounds are both greater than $100, so the operator knows that the predicate is true. Note that the error bounds are still much larger than $.01, which, as we show in our experiments, often results in a drastic compute time savings over "black box" function execution.

Given this design, VAO UDFs obviously have a different cost model than traditional UDFs. Consider $f(\langle args \rangle)$, which denotes a function call with argument value list $\langle args \rangle$. If f is implemented as a traditional UDF, assume the call has an execution cost[5] $cost_{trad}(f, \langle args \rangle)$. Now consider the function implemented in the VAOs interface. For iteration i where $\mathcal{F}^{f,\langle args \rangle,i}$ is the state of the system immediately before the ith iteration, the cost of the iteration is:

$$cost_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i}) = get_{state}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i}) + exec_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i}) + store_{state}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i})$$

Here, $exec_{iter}$ is the cost of the actual iteration execution. In addition, the iteration must also get and store state in the result object, which is represented by get_{state} and $store_{state}$, respectively. If a function requires N iterations, the system will save work using VAOs if:

$$cost_{trad}(f, \langle args \rangle) > \sum_{i=1}^{N} cost_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i})$$

Our explanation of VAOs above holds for simple operators such as selection, where the operator considers one result object at a time. VAOs that consider sets of result objects, such as the MAX operator discussed in Section 1, are more complicated. For example, iterating over any result object comprising an aggregate value may affect the error of the aggregate. Such VAOs must choose iterations from among the objects in the set in order to obtain an answer. Each of these VAOs requires an *iteration strategy*, which chooses iterations that compute an answer without excessive compute work.

A VAO with an iteration strategy requires two changes to our VAOs description above. First, the VAO must choose each iteration according to its strategy, which takes some amount of compute time. To account for this, we add the cost of choosing an iteration, $chooseIter(f, args\rangle, \mathcal{F}^{f,\langle args \rangle,i})$, to the cost of each iteration. We will discuss this cost in Section 5 for each operator for completeness, but this cost is not significant in our experiments. Second, these VAOs require information on the relative costs and benefits of different iterations in order to choose between them. Result objects supply this information with the following additional data members:

- *estCPU*: The estimated CPU cost of the next call to *iterate()*.
- *estL* and *estH*: Estimates on the L and H bounds that would result from the next call to *iterate()*.

A result object must update these estimates each time an iteration is run. In the next section, we discuss how each VAO uses this information to choose iterations. In Section 4, we discuss how this information can be easily obtained for a large class of numerical functions.

4. VAO FUNCTIONS

As discussed earlier, VAOs can be used to process queries with UDFs where there is a trade-off between accuracy and compute time. While many numeric functions exhibit this quality, current solvers must be modified to accommodate the VAO interface. We have designed such modifications for a wide variety of solvers, including partial and ordinary differential equation solvers, numerical integrators, and root solvers. Due to lack of space, we only report on our PDE (partial differential equation) solver here; the rest of the solvers are discussed in [8]. While we continue to use a bond model as our running example, we note once again that these solvers are used in applications ranging from high-energy physics to semiconductor process design, and thus have a wide range of applicability.

Before describing our modifications, we first give a brief statement of the PDE problem, as well describe a common solver technique. In many UDFs such as bond models, the output is the solution to a function which does not have a closed form. A partial differential equation may be known, which describes the change in function value with respect to the changes in parameter values. For example, consider a bond model where a price depends on the result of a function $F(x,t)$, where x is the interest rate and t is the time. Here, time is measured from 0 (the current time) to time t_{mat} (the time that the bond matures). In many cases, we do not know F, but we do know a PDE describing F; Figure 6 shows a PDE used in a real bond model [28]. In this PDE, all variables are known a priori except for x, t, F, and the partial derivatives of F. In addition to the PDE, we often know the value of F at specific *boundary* conditions. For example, we know that the value of a bond is 0 after the last payment is made at maturity, so $F(x, t_{mat}) = 0$ for all x. Given a PDE and a set of boundary conditions, we need a solver that numerically approximates the value of F for a given set of parameters. For a bond model, we typically

[4]Actually, enough *iterate()* calls on a limited-precision machine would eventually result in round-off error for many numerical methods. In many cases, this error could significantly change the answer. See Chapter 1 of [2] for further information on round-off error.

[5]We assume this cost includes argument and output value marshalling. The work in [21] provides a more detailed breakdown of UDF cost, but this level of detail is not necessary here.

$$\frac{1}{2}\sigma^2\frac{\delta^2 F}{\delta x^2}(x,t) + [\kappa\mu - (\kappa+q)x]\frac{\delta F}{\delta x}(x,t) + \frac{\delta F}{\delta t}(x,t) - rF(x,t) + C = 0$$

Figure 4: Sample PDE

want the value $F(x_{cur}, 0)$, where x_{cur} is the current interest rate and 0 is the current time.

PDEs such as the one shown in Figure 6 are solved using *finite differencing*. Finite differencing solutions involve finding a mesh of function solutions at different parameter values. The mesh contains the solution at the desired parameters, as well as the solutions known from the boundary conditions. Figure 7 shows a mesh of F solutions from our example bond model at different x and t values. In this figure, there is an entry for the needed solution, $F(x_{cur}, 0)$, as well as a whole column of solutions for the boundary condition $t = t_{mat}$. In our example mesh, the entries are equally spaced on the x and t dimensions, with adjacent entries separated by *step sizes* of Δx and Δt, respectively.

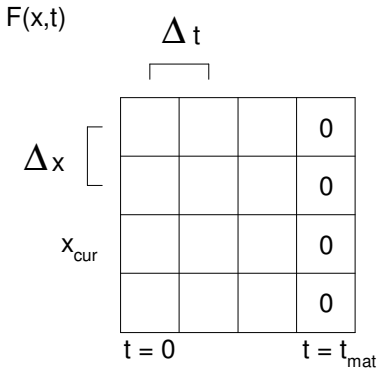

Figure 5: Mesh of solutions at different x and t for our sample PDE, with step size Δx and Δt and boundary conditions filled in.

To solve this mesh, a finite differencing solver first fills in the values known from the boundary conditions. In our example, all mesh entries are 0 in the column where $t = t_{mat}$. Using these values, the solver then incrementally computes mesh entries with the PDE, using finite difference estimates for the partial derivatives. In our example, the solver works backwards from the values where $t = t_{mat}$, and continues until it has the solution for the needed mesh entry, $F(0, x_{cur})$. The finite difference estimates depend on the solver used, and a survey of PDE solvers can be found in Chapter 12 of [2].

Like many numerical techniques, PDE solvers provide approximations to the true answer, and both the error and compute work needed depends on the step sizes. The compute work is proportional to the number of mesh entries, which is inversely proportional to the step sizes. As the step sizes decrease and more work is added, however, the error typically goes down. For our example PDE, the solver used in our experiments yields error of the form $O(\Delta t + \Delta x^2)$. Unfortunately, we often only have a form for the error, which is difficult to determine exactly. Since we need real-valued error bounds for the VAOs interface, we estimate the error using extrapolation techniques[6], which use the big-O error form and solutions at different step sizes to derive a function for the error.

Since these extrapolation techniques vary based on the form of the error, we explain them in terms of our example PDE. Given the error form above for our PDE, we approximate the function for the error with $K_1\ \Delta t + K_2\ \Delta x^2$, where K_1 and K_2 are constants. This formula ignores any higher order terms hidden by the big-O form, but it provides a starting point for estimating the error. To estimate K_1 and K_2, we compute the same PDE solution multiple times with different step sizes. Suppose we compute three solutions to our PDE (F_1-F_3) with the same x and t values, but using different step sizes as shown in Table 1. While F_1 is computed with step sizes Δt_* and Δx_*, F_2 and F_3 are computed with one of the two step sizes cut in half, making them more accurate.

Value	Δt	Δx	formula
F_1	Δt_*	Δx_*	$A + K_1\Delta t_* + K_2\Delta x_*^2$
F_2	$\frac{\Delta t_*}{2}$	Δx_*	$A + \frac{1}{2}K_1\Delta t_* + K_2\Delta x_*^2$
F_3	Δt_*	$\frac{\Delta x_*}{2}$	$A + K_1\Delta t_* + \frac{1}{4}K_2\Delta x_*^2$

Table 1: PDE solutions, with associated step sizes and formulas used in extrapolation.

For each value we compute, we can express the value as the sum of the accurate answer A and our estimated error formula. These formulas are shown for each value in Table 1. Of course, we do not know A, but we can estimate K_1 and K_2 using these formulas and the solutions that we have computed. Simple arithmetic tells us that $F_1 - F_2 = \frac{1}{2}\ K_1\ \Delta t$, and thus $K_1 = 2\ \frac{F_1 - F_2}{\Delta t}$. Similarly, $K_2 = \frac{4}{3}\ \frac{F_1 - F_3}{\Delta x^2}$.

If the formula $K_1\ \Delta t + K_2\ \Delta x^2$ characterized the error exactly, we could compute K_1 and K_2 exactly, and subtract off the error terms from the solution to obtain A, the accurate answer. Unfortunately, the formula does not consider terms hidden in the big-O form. Therefore, the formula only yields an approximation on the error, and the extrapolation technique will compute different K_1 and K_2 values at different step sizes. After running each bond in our experiments at different time and space steps, we found that K_1 and K_2 can vary in magnitude by a factor of 1.5 and 2.5, respectively. Since K_1 is always positive and K_2 was always negative in our experiments, we know the accurate answer A is bounded conservatively from below by $F_1 - 1.5K_1\ \Delta t$ and from above by $F_1 - 2.5K_2\ \Delta x^2$.

With such extrapolation techniques, we can easily implement the entire VAO interface for these PDE solvers. When a new result object is created, it finds L and H as described above, using very coarse (i.e. large) step sizes. On each iteration, the result object halves one of the step sizes, finds a new solution, and updates the error bounds by updating

[6]See Chapter 4 of [2] for a discussion of extrapolation techniques.

the error formula. The compute work is directly proportional to grid size, so each iteration requires twice the work of the previous iteration. Note that this error formula can also estimate the error for other step sizes. The object uses this formula to both a) ensure that it halves the step size that yields the most error reduction on each iteration, and b) update after each iteration the $estL$ and $estH$ fields, which predict the bounds after the next iteration. The only other field provided by a result object is $estCPU$, which estimates the CPU cost of the next iteration. For most solvers, the CPU cost in terms of FLOPs[7] can be calculated as a function of the step sizes, so the object can easily keep $estCPU$ up to date.

In most cases, the cost of this result object compares favorably to running the PDE solver in a traditional UDF with high accuracy. If a function call $f(\langle args \rangle)$ is running the solver, Section 3.2 gives us general equations for the cost using both the traditional UDF and VAO interfaces. The cost of the traditional UDF, which is $cost_{trad}(f, \langle args \rangle)$ in Section 3.2, is the cost of running the PDE solver at a fine enough grid to give a result within \$.01. On the other hand, the cost of a VAO iteration, which is $cost_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i})$, has several components. In our experiments, the work done in choosing an iteration is trivial, so we will assume that $chooseIter$ is negligible. Also, retrieving and storing result object state (get_{state} and $store_{state}$) is only a few CPU operations, so we assume that $cost_{iter}$ is roughly the cost of executing the solver ($exec_{iter}$). We can thus characterize $cost_{iter}$ by doubling the work done in the previous iteration, as shown below:

$$cost_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i}) \approx exec_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i}) \approx 2\ cost_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i-1})$$

Suppose a result object requires N iterations to obtain an answer within \$.01. On the last iteration, the VAO will do as much work as the traditional UDF PDE solver. Therefore, the previous iterations add extra overhead compared to the traditional UDF. Assuming that the initial iteration requires a relatively small amount of work, the doubling of the work on each iteration means that:

$$\sum_{i=1}^{N} cost_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i}) \approx 2\ cost_{trad}(f, \langle args \rangle)$$

Since the last iteration requires approximately the same compute time as the traditional UDF, the following equation also holds:

$$\sum_{i=1}^{N-1} cost_{iter}(f, \langle args \rangle, \mathcal{F}^{f,\langle args \rangle,i}) \approx cost_{trad}(f, \langle args \rangle)$$

Therefore, if a VAO requires less than $N-1$ iterations for a function, it will save work compared to a traditional UDFs. In the experiments presented in the next section, we show that this is usually the case.

5. VAO DESCRIPTIONS

Now that we have discussed the VAO interface and the implementation of a common numerical function, we now focus on the designs of specific VAOs. Since we described the selection VAO in the last section, we now concentrate on aggregation VAOs. The execution module is the same for all VAOs, so we focus on the operator portion of the VAOs here. For all aggregates, the VAOs must process a set of result objects to obtain a single operator output. As explained above, these VAOs each require an iteration strategy to choose iterations that provide an aggregate output without excessive compute work.

In our designs, each VAO uses a *greedy* iteration strategy. That is, a VAO continually picks the iteration which is best among the current choices until the operator produces an answer. This strategy is based on the insight that iterative techniques converge, meaning that later iterations for a given result object usually yield less error reduction than earlier ones. In the case of PDE solvers, later iterations also often require more CPU cycles than previous ones because the solver uses more information on each iteration. Therefore, the iteration that currently yields the most benefit per CPU cycle is often the best global choice.

Of course, the criteria for choosing the best iteration depends on the operator itself. Below, we describe each VAO and give the greedy heuristic for choosing an iteration from among the current objects. We also characterize the cost of using each heuristic to choose an iteration, which is a cost represented by $chooseIter$ in the cost equations above.

In our aggregate VAO designs, we assume that the aggregates are in the output of the query. Because the output of an aggregate over bounded values is also a bounded value, the user must specify a precision constraint ϵ with each aggregate. The precision constraint provides a limit on the bounds width of each resulting aggregate.

5.1 MIN and MAX

Given a set of objects O, the MAX VAO returns the bounds of an object $o_{max} \in O$ such that for all other objects $o_i \in O$, either:

1. $o_{max}.L > o_i.H$, or
2. Both o_i and o_{max} have overlapping bounds, and their respective bound widths are less than their $minWidth$ values.

In the first case, o_{max} is clearly larger that o_i. In the second, the system cannot determine if o_{max} is larger than o_i because both objects reached their stopping conditions [8]. MAX returns bounds on o_{max} no larger than the user-supplied precision constraint ϵ. Note that MIN is symmetric to MAX, so we do not discuss it here in order to conserve space.

Since bounds for o_{max} within ϵ can easily be found once the VAO identifies the object [9], finding o_{max} is the primary challenge in MAX processing. Suppose the MAX VAO is

[7] FLoating point OPerations

[8] For those familiar with approximate distributed caching work in [25], this work uses the following alternative definition for the bounds returned by MAX over bounded data: $[max_{o_i \in O} o_i.L,\ max_{o_i \in O} o_i.H\ ]$. Unlike our definition, the upper and lower maximum bounds can come from different objects. However, this property is unacceptable in many applications which want bounds on the maximum object (i.e. "Give me bounds on the bond with the largest value.").

[9] To find bounds within ϵ, note that $o_{max}.minWidth$ must be larger than ϵ. To ensure this, the current MAX implementation returns an error if ϵ is less than $max_{o_i \in O}(o_i.minWidth)$

Object	L	H	estCPU	estL	estH
o_1	97	101	4	98	99
o_2	95	103	4	96	101
o_3	100	106	4	102	104

Table 2: Data Members for example initial result objects evaluated by an aggregate VAO.

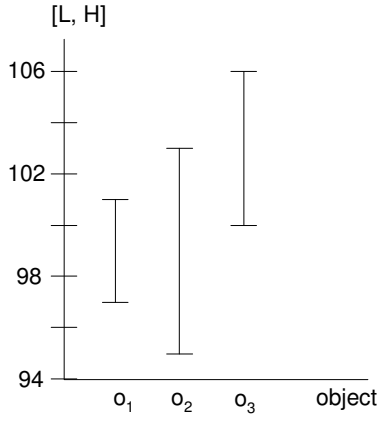

Figure 6: L and H bounds from objects in Table 1, shown graphically.

running over the objects o_1-o_3 shown in Table 2 (bounds shown graphically in Figure 4). The VAO needs to run iterations until o_{max} is found, as shown in Figure 5. Here, the operator knows that $o_3 = o_{max}$ because it has the largest bounds and there is no overlap between o_3 and any other object. Thus, the greedy heuristic should choose the iteration which provides the most overlap reduction per CPU cycle between o_{max} and the other objects. Unfortunately, the greedy heuristic has no way of knowing which object is o_{max}, since finding o_{max} is the objective.

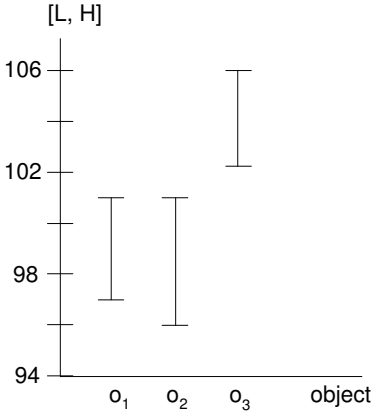

Figure 7: L and H bounds from Table 1 after MAX VAO has run iterations and found that $o_{max} = o_3$.

To deal with this problem, the VAO uses an educated guess for o_{max}, which we define as o'_{max}. The VAO currently sets o'_{max} to the object with the highest upper bound, although different criteria can be used if more information is available. The VAO then iterates over the object that reduces the total overlap the most per cycle between o'_{max} and other objects. When the o'_{max} guess no longer has the largest upper bound, the algorithm changes its guess and chooses another iteration. The algorithm keeps choosing iterations until either a) no other objects overlap with o'_{max}, or b) o'_{max} and the objects that overlap with it all hit their stopping conditions.

To demonstrate the MAX heuristic, consider a VAO choosing an iteration from among o_1-o_3 in Table 2/Figure 4. Here, o_3 is o'_{max}, so the algorithm should choose the iteration that will reduce the overlap the most, per CPU cycle, between o_3 and the other two objects. To estimate this overlap reduction for the next iteration of object o_i, the VAO computes the effect on overlap if the bounds of o_i shrink to [$o_i.estL$, $o_i.estH$]. With o_1, for example, only the reduction of $o_1.H$ to $o_1.estH$ will reduce the overlap between o_1 and o_3. Therefore, the overlap reduction is at most $o_1.H - o_1.estH$, limited by the current overlap between o_1 and o_3. Since this current overlap is $o_1.H - o_3.L$, the estimated overlap reduction for o_1 is $\min(o_1.H - o_3.L, o_1.H - o_1.estH) = \min(101 - 100, 101 - 99) = 1$.

Using similar computations, the estimated overlap reduction for o_2 and o_3 is 2 and 3, respectively. Since objects o_1-o_3 have the same estimated CPU cost ($estCPU$), the VAO chooses to iterate over o_3. Here, o_3 has the highest estimated overlap reduction, primarily because o_3 is o'_{max} and the iteration reduces overlap of o_3 with both o_1 and o_2.

In practice, the cost of choosing an iteration is quite reasonable. To choose the first iteration to run, the VAO must find o'_{max} and compute the overlap reduction estimates for all objects. This estimate can be computed in constant time for each object except for o'_{max}, where iterations reduce overlap between o'_{max} and all other objects. Thus, computing estimated overlap reduction takes $O(N)$ time for N objects. Finding the maximum estimate also takes $O(N)$ time without indexing, though indexes could certainly make this search more efficient. Once an iteration is run, the VAO has to update the overlap reduction estimates, which takes $O(N)$ time if o'_{max} was iterated over and constant time for any other object. If the algorithm changes its o'_{max} guess, the VAO has to recompute the overlap reduction estimates for each object. These computations take $O(N)$ total time, as we showed when discussing the algorithm's initial iteration choice. Overall, the VAO requires at most $O(N)$ time to choose each iteration. As more iterations are run, however, N decreases because the VAO eliminates objects from consideration which have bounds too low to be the maximum. As we will show in Section 6, the iteration choice cost is negligible in our experiments compared to function execution cost.

5.2 AVE and SUM

AVE and SUM are effectively computed by the same VAO. In addition to an object set O, this VAO also takes a set of weights W. Each $o_i \in O$ has a unique associated weight $w_i \in W$, where each w_i is a nonnegative real number. This operator effectively finds the weighted sum of the object values at each extreme; that is, it finds [$\sum_{o_i \in O} w_i \; o_i.L$, $\sum_{o_i \in O} w_i \; o_i.H$]. The operator must make iterations until these computed bounds are within the specified precision constraint ϵ, or the bounds for each o_i are narrower than $o_i.minWidth$. With N objects in O, this operator produces an average if each w_i is $\frac{1}{N}$, and it produces a sum if each w_i

is 1.

An iteration over an object o_i will increase $o_i.L$ and/or decrease $o_i.H$, provided that $o_i.H - o_i.L$ is not already less than $o_i.minWidth$. Therefore, the greedy heuristic simply chooses the iteration that yields the most estimated error reduction per CPU cycle, weighted by w_i. The VAO estimates the weighted error reduction for each object o_i with the formula $w_i\ [(o_i.estL - o_i.L) + (o_i.H - o_i.estH)]$. For the objects o_1, o_2, and o_3 shown in Table 2, the estimated error reduction is 1, 1, and $\frac{4}{3}$, respectively. Since the objects have the same estimated CPU cost in this example, the VAO iterates over o_3.

To initially run the greedy heuristic, the VAO has to compute the estimated error reduction per CPU cycle for each object. Since each estimate is computed in constant time, these computations require $O(N)$ total time for N objects. The VAO can choose the largest error reduction in $O(N)$ time without indexing. Once the VAO runs an iteration, it must update the object's estimated error reduction per cycle, which again takes constant time. Thus, the VAO can choose an iteration in O(N) time without indexing. While the VAO can choose iterations in sub-linear time using indexes such as heap queues [6], we found such optimizations unnecessary in our current experiments.

6. PERFORMANCE

Now that we have explained the VAO designs and potential applications, we now discuss VAO performance. To evaluate our designs, we implemented prototype VAOs and use them to run a variety of experiments. In this section, we present results on bond trading continuous queries similar to Q1-Q3 in Section 1. These experiments show two major results. First, experiments with real bond data and models show that VAOs often drastically outperform traditional operators under real market conditions. Second, experiments with synthetic data show that VAOs are robust under many scenarios explicitly designed to stress VAOs.

All queries in these experiments require a bond model, which in turn requires interest rates and bond data. In our experiments, We use the bond model presented in [28], which requires a numeric PDE solver. To implement this model with our VAO interface, we used the techniques discussed in Section 4. For interest rates, we use the 10-year Constant Maturity U.S. Treasury yield for days between January 3 to January 31, 1994 [17].

For the bond data, we use both real and synthetic data sets. For the real data, we use bond data on 500 mortgage backed securities issued between January and December of 1993 [10]. For the synthetic data, we designed data sets that impair the performance of the VAOs. As we show below, the VAOs are sensitive to the distribution of the function results. Therefore, we generate bond data such that the results have a distribution that reduces the performance of the VAOs. The distributions used are operator-specific, and thus we discuss them with the results for each operator below.

To create these distributions, we used the following process. First, we iterated over each bond in our real data set until we knew the result for each bond within \$.01. We then used a random number generator [18] to generate a distribution of bond model results for the same number of bonds as in our real set. We then create a random one-to-one mapping between the generated bond results and the real bonds, and compute the difference between each generated result and corresponding result from the model. When executing an iteration over a synthetic bond, we run the iteration over the corresponding real bond, and then shift the resulting bounds by the computed difference. This results in bounds that, given enough iterations, converge to the desired distribution of results.

All 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 the Fedora Core 1 Linux distribution. The prototype processes interest rate streams over the continuous queries in our experiments, and reports the wall-clock time for the processing. Although new interest rates depend on the Treasury price and arrive every 1-4 minutes, the following experiments show processing time for one interest rate, the opening rate for Jan. 3, 1994. We show one interest rate because a) the traditional operator experiments take a large amount of time on one processor, and b) the interest rate value seems to have little effect on the processing time of our queries. We ran similar experiments with the high and low interest rate in our data set, and found that the results show similar trends.

As a baseline, we implemented traditional operators and used them to run the same queries on the same data. Since traditional operators cannot adjust function accuracy according to query, we implemented a "black box" version of the model that always returns an answer with less than \$.01 error. To implement this model, we ran each bond through the model with the VAO interface, and iterated over each result object until the error was less than \$.01. For each bond, we recorded the step sizes needed to obtain this error. When we run bonds with the "black box" interface, we run the PDE solvers with these step sizes to ensure that we obtain \$.01 error. Note that these function calls often underestimate the time needed if the function was used in a production system. In these calls, the model knows a priori the step sizes needed to get the desired accuracy, and no further work has to be done to ensure that the error is acceptable.

Below, we present results on three types of queries: selection, max aggregation, and sum aggregation.

6.1 Selection Results

In this section, we discuss selection queries which find bonds that are greater (or less) than some selection constant, similar to Q1 in Section 1. Here, we first present experiments using our real bond data, which influenced the design of the synthetic data experiments which we present later.

Using our real bond data, we ran queries with different constants using our VAOs, as well as traditional operators. The constants are set to yield different selectivities for the operator. Figure 8 and 9 plot the runtimes for different selectivities for a selection query with a $>$ (greater than) and $<$ (less than) operator, respectively.

These figures show the runtimes using the both the selection VAO (*vao*) and a traditional operator (*trad*). The traditional operator runtimes are constant because performance doesn't depend on the query. In all these experiments, the selection VAO outperforms the traditional operator by over two orders of magnitude. In fact, under real

[10] Specifically, these bonds are Freddie Mac Gold PC 30-year Mortgage Backed Securities. We are greatly indebted to Nancy Wallace at the U.C. Berkeley Haas School of Business for providing this data.

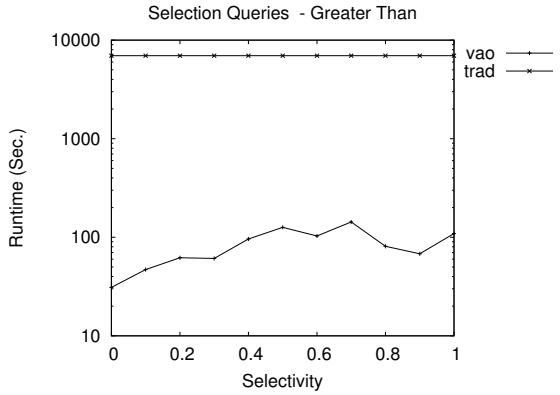

Figure 8: **Runtimes for selection query with greater than predicate. Selection predicate set to yield different selectivities.**

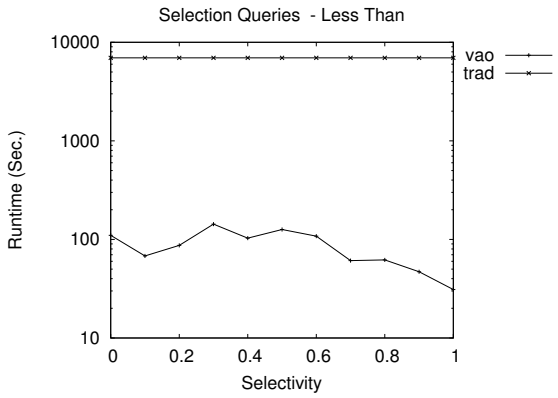

Figure 9: **Same as Figure 8, except query uses less than predicate.**

market conditions, VAOs effectively enable practitioners to run these queries in real time. Since new interest rates arrive as frequently as every minute, the traditional operator would require over 100 processors, where the VAOs would only require a few[11].

In addition to the drastic performance improvement from the VAOs, these graphs exhibit two characteristics that seem strange at first blush. First, neither graph exhibits a monotonically increasing performance improvement with selectivity, which is expected of most queries with a selection predicate. Second, note that the runtime for any selectivity s in Figure 8 is the same as the runtime for $1 - s$ in Figure 9.

Both of these characteristics appear because the performance of the VAO does not depend on selectivity. Instead, it depends on the proximity of function results to the constant. For example, consider a VAO with a highly selective constant where many function results are close in value to the constant. Although the VAO eventually eliminates many results, it still has to run many results to highly accurate bounds in order to answer the query. On the other hand, a less selective VAO with no results near the constant can answer the query with few iterations over each result. Due to this property of VAOs, we do not see a monotonically increasing performance improvement with selectivity because the runtime is determined by the number of bonds that are close to any constant. In our real data set, this number is not strongly related to the selectivity of the constant. Also, note that an experiment with any selectivity s in Figure 8 has the same constant as the selectivity $1 - s$ in Figure 9, which explains why such pairs of experiments have identical runtimes.

Using the knowledge gained from these experiments, we created experiments explicitly designed to stress selection VAOs. If the VAO has higher runtime when bonds are close in value to the constant, we can raise the runtime by decreasing the difference between bond model results and the constant. To do this, we generated several different Gaussian distributions of bond values, and ran experiments with the selection VAOs. The mean of these distributions was set to the VAO constant, while we varied the standard deviation to control the distance of the results to the constant. The results are shown in Figure 10.

[11]The models are easily parallelizable; see [11] for details

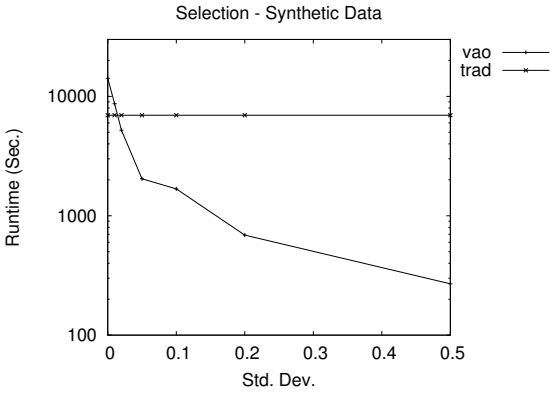

Figure 10: **Selection VAO and traditional operator with synthetic data.**

In this figure, the pathological case occurs at 0 standard deviation, where all bonds have the value of the predicate constant. Here, the VAO is actually more expensive than the traditional operator. The VAO has to run each model to the same accuracy as the traditional operator, but it also has the overhead of the previous iterations. Fortunately, the VAO performance improves quickly as the standard deviation rises from 0. Since the VAO becomes much cheaper than the traditional case at only \$0.05 standard deviation and the standard deviation of our real bond prices is approximately \$7.78[12], we conclude that the VAO performs quite well except in the most pathological cases.

6.2 MAX Aggregate Results

Given the selection results, we now turn to queries that find the largest bond price from our set of bonds, similar to query Q3 in Section 1. Like the selection VAO, we first consider experiments over real bond data. In these experiments, we have runtimes for our MAX VAO, a traditional aggregate operator, and an operator with a theoretically optimal iteration strategy (Optimal). In order to provide a

[12]Although the distribution is somewhat centered around the mean, it is a real data set and does not resemble a theoretical distribution.

fair comparison, the VAO and the optimal operator return a bounds with less than $.01 error. This is the same accuracy returned by the "black box" functions used by the traditional operators.

Unlike the VAO, the optimal operator knows in advance the bond that has the maximum value. Therefore, it iterates over this bond until the error is less than $.01, and then iterates over other bonds until no error bounds overlap with the maximum bounds. As running the maximum bond to an accuracy higher than $.01 is useless, this operator provides an optimal iteration strategy, albeit one which requires knowledge of the maximum bond a priori.

Operator Type	Runtime (sec.)
Optimal	108
VAO	111
Traditional	6953

The runtime for each operator is shown above. For traditional operators, the MAX runtime is effectively identical to the selection runtime because the same amount of work is done in the functions. Note that the VAO takes almost two orders of magnitude less time to answer the query than the traditional operator. At real market data rates, VAOs can again run the query with only a few processors, while traditional operators require over 100.

When comparing the VAO to the optimal operator, the extra work in the VAO is only 3 seconds, which is less than 3% of the total work in the optimal case. Most of this extra work comes from the fact that the VAO is initially wrong in its guess for the maximum bond, and must eventually correct itself. The remainder of the overhead, which is less than .1 second, comes from the VAO's more complex iteration strategy. As stated in Section 5.2, the time to choose an iteration is linear on the number of bonds that still have the potential to be the maximum. In our experiments, only 5 bonds meet this criteria after the initial iteration, and each iteration potentially eliminates another bond. Therefore, only a small number of bonds were considered for iteration by the VAO in our experiments.

Given our experiments over real data, we now turn to synthetic data experiments. The MAX VAO is sensitive to the distribution of results, but in a different manner than the selection VAO. As the MAX VAO must find the maximum results, the MAX VAO has higher runtime when more results are clustered around the maximum. To simulate this clustering, we again generated bond model results from a Gaussian distribution, but we only took prices from the lower half of the distribution. In the worst case of 0 standard deviation, all bonds are the same value, and the VAO must run all bonds to $.01 accuracy to determine that they are all the same. As the standard deviation rises, fewer and fewer bonds have model results near the maximum.

Figure 11 shows the results of the synthetic data experiments with the MAX VAO and the traditional operator. As in the selection experiments, the MAX VAO only performs worse than the traditional operator in the worst cases. At $0.10 standard deviation, the VAO significantly outperforms the traditional operator, and it continues to drop as the standard deviation rises.

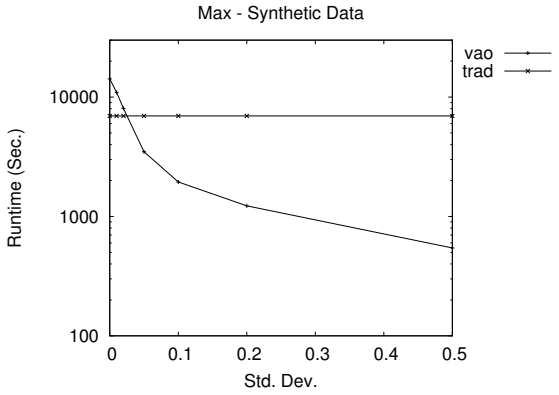

Figure 11: MAX VAO and traditional operator with synthetic data.

6.3 SUM Aggregate Results

In this section, we present results for SUM aggregate queries that compute a sum of bond values at the current interest rate. Unlike MAX and selection, the SUM VAO is not directly affected by the distribution of function results Instead, the performance of VAOs depends on how much each result is weighted in the sum. Weighted sums are often used to find the value of a portfolio, for example, where each security price is weighted by the number of shares held. If some of the results are heavily weighted, the system can run more iterations over these results, since the error in the lightly weighted results has less effect on the overall sum. If results are equally weighted, however, the system has less opportunity to save cycles by adjusting the iteration strategy.

To evaluate the SUM operator, we ran different SUM queries with weights generated with what we call a *hot-cold* scheme. With this scheme, we set a constant total amount of weight, and partition the bonds into a *hot* and a *cold* set. In the experiments shown here, the hot set includes 10% of the total bonds chosen randomly, and the cold set contains the remaining bonds. In our experiments, we vary the amount of total weight that is allocated to the bonds in the hot set. As more weight is allocated to the hot set, we see a few bonds that are more heavily weighted, and should see more performance benefit from the VAOs.

The total weight in each of our experiments is 500, the cardinality of our bond set. Each VAO query has a precision constraint of (500)($.01) = $5. This constraint reflect the error bounds on the traditional SUM operator when all bonds are run with $.01 error.

Figure 12 presents the runtimes for SUM queries with different weight percentages allocated to the hot set. In this figure, the traditional operator actually outperforms the VAO for low percentages. In these scenarios, the VAO can do relatively little optimization, and the VAO has the extra cost of running intermediate iterations. As the percentage gets larger, however, the VAO optimizations eventually outweigh the overhead, and the VAOs are up to over 4 times faster than the traditional operator. Overall, this figure shows the expected results; although the SUM VAO is not useful when results are nearly equally weighted, the VAO shows significant performance improvement when a few results are heavily weighted.

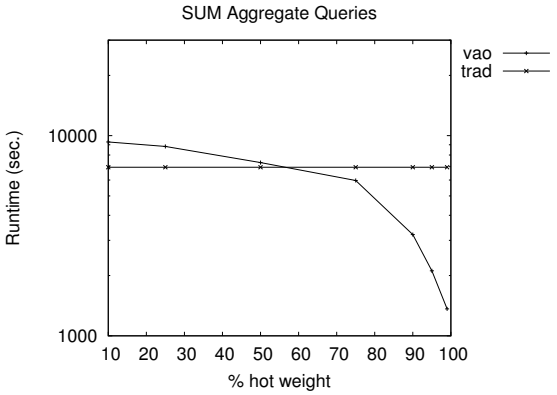

Figure 12: SUM aggregate queries with different percentages of weight on the hot set.

7. CONCLUSION

In this paper, we address the execution of expensive user-defined functions (UDFs) in either static or continuous queries. While previous work focuses on avoiding expensive function calls, systems still need to optimize the actual function executions. These optimizations are impeded by the "black box" UDF interface used by continuous query systems, which do not allow the systems any control over function execution.

To speed up expensive functions, we exploit the trade-off between accuracy and compute work inherent in many functions. That is, we optimize functions that return more accurate answers if more compute cycles are used. We present Variable-Accuracy Operators (VAOs), a new class of operators which computes each function result only to an accuracy needed to answer the query. To adjust accuracy, VAOs use a new function interface that returns upper and lower bounds on the result, and allows the VAO to refine the bounds by using more compute cycles. In this paper, we describe our VAO designs for selection and 4 aggregate operators. We also demonstrate how a large class of numeric functions can be implemented with the VAO interface.

To evaluate these operators, we built prototype VAOs and ran experiments with bond trading queries using real financial data and a numeric bond model. In these experiments, the VAOs outperformed traditional operators with "black box" interfaces by up to over two orders of magnitude for some queries. In addition to experiments with real market data, we also found VAOs to be robust in performance tests explicitly designed to stress VAOs,

8. REFERENCES

[1] J.C. Brown. Adaptive mesh refinement (tutorial). http://www.cs.utexas.edu/users/dagh/Tutorial/jcb_mitra/tut_jcb_mitra.html.

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

[7] M. Denny and M. J. Franklin. Predicate result range caching for continuous queries. In *SIGMOD*, 2005.

[8] M. M. Denny and M. J. Franklin. Adaptive execution of variable-accuracy functions. Technical report, U.C. Berkeley EECS Dept., 2006.

[9] A. Deshpande, C. Guestrin, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[10] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB*, 1999.

[11] 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.

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

[13] E. W. Egan. Multi-scale problems in modeling semiconductor processing equipment. In *Mathematics in Industrial Problems: The IMA Volumes in Mathematics and its Applications, Volume 88, Part 9*. Springer-Verlag, 1998.

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

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

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

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

[18] Gnu scientific library. http://www.gnu.org/software/gsl/.

[19] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

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

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

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

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

[24] R. P. Mount. Scientific computing at slac. http://researchcomp.stanford.edu/hpc/archives/SLAC-RMount-aug15-v03.pdf.

[25] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *VLDB*, 2000.

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

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

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

[29] P. Wesseling, editor. *Principles of Computational Fluid Dynamics*. Sprenger-Verlag, 2000.

[30] Yahoo! finance. http://finance.yahoo.com/.