

CAVES: A Configurable Application View Storage System*

S. Adah, C. Carothers, C. Chiang, D. Spooner, and G. Yaun

Rensselaer Polytechnic Institute

{sibel,chrisc,chianc,spoonerd,yaung}@cs.rpi.edu

Abstract

Storage management is a well-known method for improving the efficiency of data intensive and networked applications. Today's data management systems handle many non-traditional data formats, ranging from spatial data to images, video and other hybrid representations. This requires the use of specialized methods to query, extract and transform data from multiple, possibly distributed sources. There is a great need to develop efficient and scalable methodologies for storing and reusing the results of computations in such applications. In this paper, we introduce a configurable storage management system that allows programmers to specify a collection of storage management protocols for managing different types of data requests on top of a shared and possibly distributed pool of resources. In addition, dynamic protocol change rules allow the system to shift from one storage management method to another depending on the availability of system resources. Furthermore, the storage management system can be expanded with application specific methods to look-up and re-use stored items. We show how the storage system can be tuned to specific workload specifications with the help of a simulation model that takes into account both the cost of storage management protocols as well as the methods to look-up and re-use stored items.

1 Introduction

In this paper, we introduce a configurable application view storage system called "CAVES" that is designed to work as a middleware system, connecting multiple possibly distributed servers to multiple possibly distributed clients. The main purpose of any storage system is to reduce the overall turnaround time between an application and a data provider. To achieve this, storage systems make use of the possible locality between different data requests and try to optimize the overall performance by storing data that is most likely to be re-requested in the near future in a fast storage medium. In this paper, we consider the local disk of a storage management system a fast medium compared to networked and possibly distant servers that process complex database queries. In the remainder of this paper, we will use the term *view* to refer to the output of a query in any application.

Examples of applications that involve costly queries include data warehousing and data mining applications processing complex queries over large data sets, applications that use spatial aggregations and correlations over complex vector data, biological databases that process highly complex sequence comparison algorithms, large data sets obtained from scientific experiments, etc. In such applications, the time to process and transmit a single query may be rather high even in the presence of indices and the size of the output may be very large. The cost of producing such a data set might be much larger than the cost of writing and reading back the data set from the local disk. In regular storage replacement algorithms, the goal of the system is to optimize the total number of hits. This assumes that the cost of producing each item in the storage is uniform. However, this is not the case for complex queries that

*This research is supported by the National Science Foundation grant IIS98-76932.

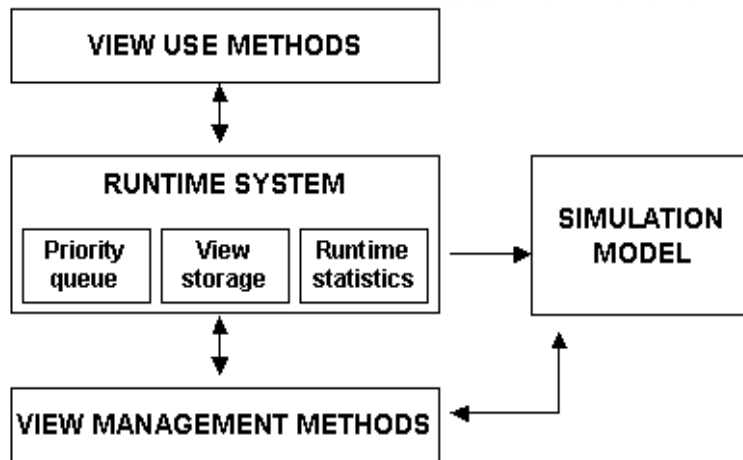


Figure 1: The architecture of the CAVES system.

are transmitted over a network which optimizes the overall savings in time. As a result, the replacement policies for an application may vary greatly based on its workload.

The CAVES system is designed as a general purpose storage management system that can be stored either at a middleware server or at a client machine. Its purpose is to store views from multiple applications and re-use them whenever possible. The CAVES system is fully programmable, making it possible for applications programmers to specify various storage management protocols that are best suited for a given application. The CAVES system also works as a fast prototyping machine. It constantly monitors the general system behavior, collects statistics and measures the effectiveness of the predefined storage management rules for a given instance. This is accomplished with the help of a simulation model of the actual system that can be run periodically to perform specific system optimizations.

The contributions of this paper are the following. We first introduce a general purpose system for managing different query requests from multiple client applications. We show how the CAVES system can be customized to handle different storage management system protocols by means of explicit storage management rules. We show how view lookup, view reuse with different query rewriting methods and view management can be integrated within a single system. In addition, we show how rules that govern the dynamic behavior of the system can be defined and executed. We develop algorithms to maintain the different data structures used in the system. We then show extensive experimental results that support our hypothesis that (1) the optimal storage management protocol may vary greatly based on various factors, (2) a dynamically self tuning system may result in great savings in the overall performance.

2 General Architecture

There are three main parts to the CAVES system:

- A *CAVES instance specification* corresponds to the collection of all application specific storage management rules. These rules are specified by an application programmer using a declarative specification language that can be embedded in an XML document. Examples of different parameters and their use are explained in Section 3. The specifications provided by the programmer are compiled into the runtime system to produce a customized “storage server” for this instance. The following are the main components of the specification:

- *View use methods* - contains the main methods for reusing stored views from different applications. These methods are implemented in advance by a programmer and linked to the general system using commands that are similar to the “create function” command for user defined functions. The methods that are created by an application are one of two types. View lookup methods allow the storage management system to compare the description of two views V_1 and V_2 . These methods will return true if a stored view V_1 may be used to answer a new request V_2 based on a stored property of the views. This attribute can be a description string containing a complete query or a simple range on a relational attribute. The second type of methods are view filter methods which allow the stored views to be used to answer new, more restrictive queries. Many view re-use methods have been suggested in the information integration literature such as query rewriting using views [11, 13, 17]. Other methods may involve a zoom-in operation for vector data, simple selection operations on views, and aggregation of a relation on a specific attribute. Since these methods can be very costly, they should be used only when they are likely to improve the overall system performance. We will discuss how this decision can be handled by the CAVES system.
- *View management methods* - contains the main storage management rules for a given application. The view management methods are based on the notion of priority that describes the importance of a view for the clients of an application. Each view type stored within a single instance of the CAVES system has a set of properties, called view statistics. All the statistics that are to be kept for views are defined by the application programmer. Each statistic has methods for initialization and update. Intuitively, each view may have a set of statistics that are initialized at creation time such as the view creation time and the time to compute that view. In addition, certain statistics are updated as stored views get referenced, such as the number and time of hits, etc. In general, statistics update methods are invoked by view use methods. The priority of a view is defined in terms all of its statistics and a function over them. The formulas that define the priority ordering are also defined by programmers. In addition, these formulas may be allowed to change over time based on various system parameters. The rules that govern this behavior are also a part of the view management specification of an instance.

Both view use and view management methods are tightly connected to each other. The description supplied by a programmer is first validated against a grammar. In addition, the correctness and completeness of all formulas are also verified before an instance is generated.

- *The runtime system* manages a preset disk space whose size is defined by an instance. For each instance, it stores a set of views on disk and all the pertinent statistics for these views in memory. Each view has a statistics called “priority” that is computed based on the current priority formula. The runtime system contains a search structure called a “priority queue” that facilitates the basic operations on these views: removal of views with smallest priority, updating priority of referenced views and insertion of new views into storage (with $O(\log n)$ time for all operations for a queue of size n). In addition, the runtime CAVES system maintains a number of system statistics that range from the mean time of arrival of query requests to the distribution of query requests. The system statistics are used to tune the performance of the system to the actual workload being experienced.
- *The simulation model* is the final component of the system. The model contains a realistic model of the runtime system augmented with the user defined methods for determining priority. The declarative definition of a CAVES instance is used to generate these methods. The simulation model is fed the runtime characterizations of the actual system using the system statistics collected. These statistics are used to run various tests that approximate the actual requests to the system and determine the effectiveness of different rules. In addition,

the simulations are used to discover new storage management rules whenever possible. The refinements over the existing system are fed back to the instance to improve the performance.

We will now describe each part in detail in the following sections.

3 Storage Management in CAVES

In this section, we will describe the application specific components of the CAVES system and show how they are used within the runtime system. We will mostly concentrate on the methods that describe the priority of a view. A CAVES instance always contains the following components: (1) a storage space (SS) defined in terms of a path and a preset size given by $SS.size$, (2) a set of view statistics with methods to initialize and update, (3) a set of query types (QT) that are to be managed by this instance distinguished by a specific port they are attached to, (4) for each query type i , the set of view statistics that will be maintained in the system, (5) a list of order formulas $OF = \langle f_1, \dots, f_m \rangle$ which are formulated in terms of the view statistics and simple arithmetic operators such as $+$, $-$, $*$, $/$, and (6) a weight vector $W = \langle w_1, \dots, w_m \rangle$ that assigns a weight to all the order formulas where $w_i \geq 0$.

Query types in our model are data requests with different computation costs and output sizes. For example, the computation cost of a selection query is much smaller than that of an aggregation or a sequence matching algorithm. In addition, the result of a simple selection may be much smaller in size than that of spatial query that produces a complex vector of values. Suppose a specific shared disk space is allocated to store views of different types. One approach to attacking multiple types than would be to allocate a specific percentage of the space to each type. However, it is possible that the views of a specific type will under-utilize its portion, while another type with many hits runs out of space. It makes sense then to adapt the usage of the disk space to the actual requests being made by the clients. If for a specific reason, a specific query type should be given a higher priority than most other views, then this protocol can easily be coded into statistics and order formulae as described in the next two sections.

3.1 Statistics

Each CAVES instance stores a number of views. The actual contents (body) of a view is stored on disk while the other properties of the views currently in storage are kept in memory. These properties, called statistics are updated periodically based on different factors as explained below. We will use the notation $V.A$ to refer to the *current* value of statistic A for stored view V . In this paper, we assume that the storage management system does not handle updates to the stored views when the underlying data sources change. This can be accomplished with the help of additional routines to handle incremental updates to the data. These methods depend greatly on the underlying application and are out of the scope of this paper. We assume that for time critical views, a specific statistics called “expiration date” will be specified and the stored views will not be used (hence removed from storage) after this time point.

In this model, if no specific view use method is provided for a view type, the default action is to use a statistics called “description”. Two views are expected to contain the same content if their description is identical. We assume that queries intercepted by the storage management system carry a header that contains the description of the query. It is also possible to specify a hash function on the descriptions to speed up the view look-up operation. In this case, a hash structure is generated on top of the views in the system. Examples of statistics that can be specified for a storage management system are listed in Figure 2.

The statistics such as *desc*, *type*, *server*, *size*, *cc*, *body* and *timein* are static values. Their initial value is set at the time the view is admitted to a CAVES instance and this value remains unchanged as a result of view use. The first three are obtained from the query request that initiated from the client. The remaining statistics are obtained from the server that answers this query. The body is the actual view that is returned to the user, *size* and *timein* are

Symbol	Meaning	Domain
<i>desc</i>	Description of the data access request for a view	String or Integer
<i>size</i>	Size of the view	KBytes
<i>type</i>	Type of a view	Integer
<i>body</i>	Body of a view	LOB
<i>priority</i>	Priority of the view (as a function of its statistics)	Real
<i>user</i>	The identity or priority of the user who uses/owns the view	Integer
λ	Average rate of reference	Real
<i>hits</i>	Total number of hits	Real
<i>cc</i>	Total computation cost: computation time + transfer time	ms
<i>usage</i>	Percent of view used for hits	0-100
<i>timein</i>	The time the view was admitted to storage	Time
ref_1, \dots, ref_k	Time of the last k references	Time
<i>server</i>	The network address of the server the view is obtained from	IP
<i>nr</i>	The average network transfer rate for <i>server</i>	Kbytes/sec
<i>sr</i>	The average service rate at <i>server</i>	Kbytes/sec

Figure 2: Example statistics for the CAVES system.

set when the view is admitted to the storage. The computation cost cc is set by default to the total turnaround time for this view. The initial values for all these statistics are then obtained by parsing the query request, measuring the size of the output and timing the reply to the request. These statistics have no update method. If the server returning this view also supplies the actual computation cost, this can be stored as an additional value.

The value of other statistics change as views are referenced or hit by the view use methods. Their default value is normally a null value or zero. As an example, suppose the value of $user$ for a view is the maximum priority among all users that referenced this view. The total number of $hits$ is simply incremented every time a view is referenced. The last k references to the view are shuffled anytime it is hit, i.e. by setting $new(ref_1) = old(ref_2), \dots, new(ref_{k-1}) = old(ref_k)$ and $new(ref_k) = now$ where now is the current time stamp and the functions new and old refer to the new and old values of the given statistics. The $usage$ and λ statistics are implemented somewhat differently. For example, we can compute λ as follows:

$$new(\lambda) = \frac{old(\lambda).old(hits)}{new(hits)} + \frac{C}{(new(ref_k) - old(ref_k)) * new(hits)}$$

where C is a constant value. The $usage$ statistic is useful when we use view re-use methods that filter stored views to answer new user requests as described in Section 2. Suppose a stored view V_1 is used to answer a new query with answer V_2 . Then, $\frac{V_2.size}{V_1.size}$ is called the percentage use for V_1 . The usage statistic is then calculated by multiplying C in the above formula with the percentage use. The final group of statistics such as nr and sr are computed using system functions such as “ping” or by querying databases for statistics such as the average load of a server. Using these functions, we can derive an average rate for both network and server rates. The update methods for these statistics are executed periodically for each server. There is no restriction on what and how many statistics can be used in the CAVES system as long as methods for initialization and update can be defined using simple formulae. However, the usual tradeoff between accuracy and complexity plays an important role in the system performance.

3.2 Order Formulae

The second important component of a CAVES instance is a set of order formulae. Each order formula is defined in terms of a number of statistics in that instance. Each formula returns a numerical value and therefore assigns an ordering among the stored views based on the current values of their statistics. In the following, we will assume the smallest number for an order formula indicates the least desirable item, i.e. the view with the smallest priority. Given an order formula f , we will use the notation $f(V)$ to denote the value of formula f for the statistics of view V . If view V does not contain a specific statistic that is used to compute f , then we will assume $f(V) = 0$.

The well-known replacement schemes in operating systems such as FIFO, LRU (least recently used) and LFU (least frequently used) determine the priority of views based on the values of statistics $timein$, ref_k , and λ (or $hits$). Another formula that considers the age of a view is given by $OF = \frac{cc}{size * (now - timein + c_1)}$ (OF =oldest first) where c_1 is a small constant. In this formula, the more costly or the smaller a view, the more valuable it is. The value of the view decreases linearly by age. A formula HWF (hits with frequency) with frequently used variations based on the number of hits is given as follows:

$$HWF = \frac{least(hits, k) * cc}{size * (ref_k - ref_1)}.$$

This formula is similar to formula OF , however in this case the number of hits and the time interval between the last k hits is considered instead of the age of a view. If a system uses the number of hits to determine the storage management policy, then it is necessary to handle new views in a special way. Since the system is biased towards already existing views, the new views will be removed immediately after they are admitted to the system. To remedy this, an aging function such as OF is used initially for new views, then this function is phased out slowly and replaced by HWF . Hence, if a view is not hit during the time interval in which OF is being used, it will be removed whenever HWF takes over. To mimic this functionality, one can use a weighted sum such as $w_1 * OF + w_2 * HWF$ over these two formulas to determine the final priority of a view. In time, the factor OF will approach to zero and HWF will be the dominant factor for computing priority.

Another order formula FNF (fast network first) given by $FNF = \frac{1}{nr}$ depends on the total number of hits for a view and the amount of time it takes to transmit this view from its server to the storage management system. This function depends on the availability of the network bandwidth to answer this query. The slower the connection to a specific server is, the more valuable are the views that originate from this server. The function FNF may be called *the hardest to re-transmit*. These are examples of some order formulas that can be defined for a storage management system. Many other formulas may be defined for different instances.

Let $OF = \langle f_1, \dots, f_m \rangle$ be the list of all order formulas that are defined for a given CAVES instance. A weight vector $W = \langle w_1, \dots, w_m \rangle$ is a list of real values such that $w_1 + \dots + w_m = 1$. Then, the priority of a view V in this instance is given by the formula: $Priority(V) = w_1 * f_1(V) + \dots + w_m * f_m(V)$. This approach to priority generalizes the above notion of using multiple functions with varying importance when deciding which views to keep and which views to remove.

3.3 View Insertion into CAVES

The final component of a basic CAVES instance is given by a priority queue PQ. The priority queue is a binary search structure with $O(\log n)$ complexity for both deletions and insertions. The priority queue contains the information about stored views in the increasing order of their priority. Whenever a view needs to be removed to free up space, the item with the smallest priority is located and removed from the system in logarithmic time.

Suppose $ST_{\perp} = \langle SS, STAT, OF, W, VIEWS, PQ \rangle$ is a CAVES instance with storage space SS , statistics $STAT$, order formulae OF , weight vector W , the set of all stored views $VIEWS$ with their associated statistics and the associated priority queue PQ . The insertion of a new view V into a CAVES instance ST_{\perp} as shown in Figure 3 is

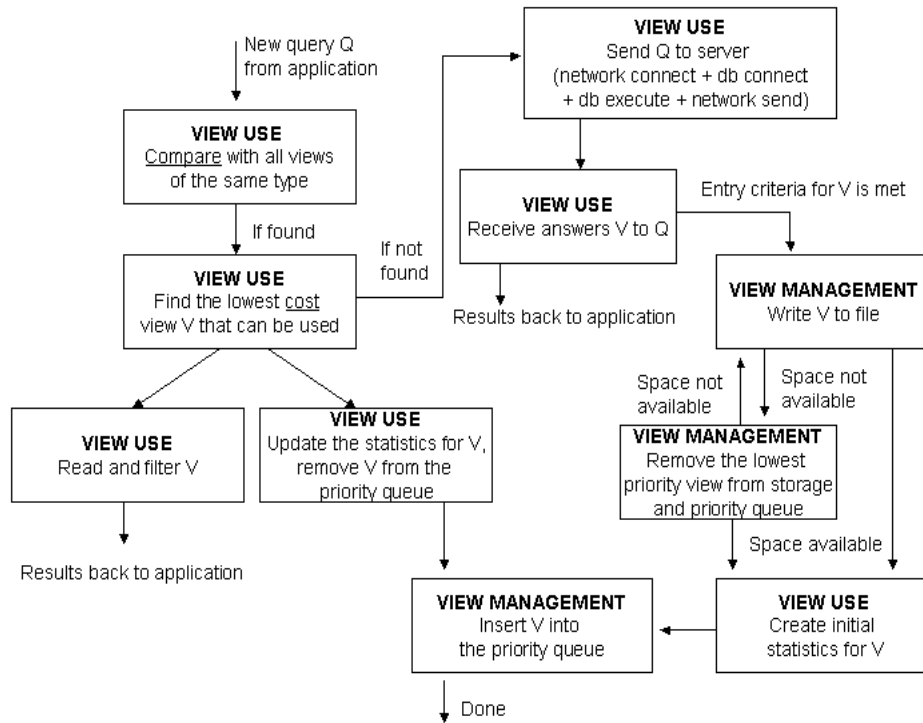


Figure 3: View management in the CAVES system.

performed as follows.

NEWREQUEST(Q)

Set $mincost$ to a high number and $answer$ to null
for each view $V \in PQ$ do

- **if V can be used to answer Q and cost of reuse for $V < mincost$ then**
 - $mincost = \text{cost of reusing } V$
 - $answer = V$

if $answer$ is not null then //a hit is found

- Read $answer$ from disk and compute the answer to query Q
- Remove the view $answer$ from the priority queue
- Recompute statistics for $answer$ using the update methods
- Insert $answer$ to the priority queue at $Priority(answer)$

else // no possible answers are found

- send Q to its server and wait for the reply
- **while answers V_Q to Q are being returned by the server do**
 - return V_Q to the user

- compute the statistics for V_Q
- **if** $V_Q.size < (ss.size - \sum_{V \in \text{VIEWS}} V.size)$ **then** //space available for the new view
 write V_Q to disk and insert V_Q to VIEWS and the priority queue
- **else** //no space for the new view
 - * **let** $free = ss.size - \sum_{V \in \text{VIEWS}} V.size$
 - * find views V_1, \dots, V_p that optimizes the following formula:

$$\text{minimize } \sum_{1 \leq i \leq p} (\text{Priority}(V_i))$$

$$\text{with respect to } (\sum_{1 \leq i \leq p} (V_i.size)) \geq V_Q.size - free.$$
 - * **if** $\text{Priority}(V_Q) > \sum_{1 \leq i \leq p} (\text{Priority}(V_i))$ **then**
 - delete views V_1, \dots, V_p from VIEWS and priority queue
 - write V_Q to disk and insert V_Q to VIEWS and the priority queue

The cost of a successful look-up is given by $O(n)$ where $n = |\text{VIEWS}|$ in the absence of any hash functions on the description of views for fast look-up. In the average, half of the priority queue is scanned if we are not interested in the lowest cost view. If we are not performing post-filtering of the views, then the actual cost of looking up a view V from the storage system is given by $c_1 * n + (c_2 + c_3) * (\log n) + V.size/dtr$ where dtr is the disk transfer rate given in terms of Kbytes/sec, and the constants c_1, c_2, c_3 are the constants for the cost of look-up (comparing two view descriptions), deletion and insertion.

Whenever we need to remove a number of low priority views to free up sufficient space to accommodate the new view, we use a heuristic that approximates the above optimization formula. We simply find the first p lowest priority views whose total size is sufficient to store the new view. In this case, on the average a constant y number of views are scanned for the deletion operation. The cost of an unsuccessful look-up that results in the insertion of a view V and the removal of another view is then given by $V.cc + c_1 * n + (yc_2 + c_3) * (\log n) + V.size/dtr$ assuming that the CAVES server has to wait for the whole view to be transmitted completely before it can be admitted to the system. Recall that the computation cost in our framework is the overall turnaround time for this query which includes the computation cost as well as the data transfer cost. The overall gain of a hit is given by $V.cc - A$ where A is the cost of a successful look up operation. In most cases, the look-up cost c_1 is very small $c_1 \ll c_2, c_3$. However, an application that uses query rewriting with views may require an algorithm that is quadratic or even exponential in the length of the query description string. In this case, c_1 might also be substantial decreasing the overall savings in time for a hit.

3.4 Ghost priorities

In the above algorithm, we assumed that the priority of views changes only when the view is referenced. However, this is not true of the function OF given as $OF = \frac{cc}{size * (now - timein + c_1)}$. If the priority of a view depends on this function, then it changes everytime the clock ticks. In this case, the easiest solution is to resort the entire priority queue (at a cost of $c_3 n \log n$ seconds) periodically.

Another solution is to maintain an auxiliary queue called *time queue* which is ordered with respect to time instants. Hence, time queue is a priority queue but stores time values instead of priorities. The insertion and deletion costs of the time queue are identical to those of the priority queue. Any time a new view V is inserted into the system, its priority is computed for time now. Suppose $p_{now}, p_{now+z}, \dots, p_{now+kz}$ correspond to the values of the priority for view V at time points $now, now + z, \dots, now + kz$ based on the current statistics of this view. If this view is never hit, then these priority values will not change as well. The values are simply snapshots of the priority

of the view at different time points. If we insert view V into the priority queue according to value p_{now} , then this value should be replaced at time point $\text{now} + z$.

INSERTION: (1) compute p_{now} for V and insert V into the priority queue using this value, (2) create a ghost instances of V , set its ghost counter to k and insert it into the time queue at time point $\text{now} + z$. The ghost contains a pointer to the view in the priority queue. Note that the time queue only contains a time value for V , not the actual priority.

TIMED OUTPUT: At any time point, we check the smallest element V in the time queue. If its time value corresponds to the current time point, then we remove V from the time queue and compute its current priority based on the current statistics. If its priority value in the priority queue is different than the new value, then we delete the view from its previous location in the priority queue and re-insert it. We decrement the ghost counter by 1, if it is not zero, then we reinsert it at time $\text{now} + z$. If the counter is zero, we mark V in the priority queue as the major copy.

HIT: If a view V is referenced and it contains a ghost instance in the time queue, then we update its priority in the priority queue as before.

DELETION: Whenever a view needs to be removed from the priority queue to make space for a new view, we check if it is a major copy. Major copies are removed exactly as before. However, if the view in question has ghost instances, then we recompute its priority. If it is still smaller than the priority of the next item in the queue, then we remove it together with all of its ghost instances. If the actual priority of this view is greater, we keep this view and proceed to the next item in the priority queue (we do not update the priority of the view, it will take place in its scheduled time). Such a view is considered to be below sea level for the time being. Whenever a ghost instance is being considered, we check if its current priority falls below the current sea level (i.e. the smallest priority pointed to by the priority queue). If this is the case, then we remove all ghost instances from the time queue and re-set sea level to point to this view.

In this method, the whole priority queue is not re-sorted periodically. Since the value of function OF will decrease in time, it will eventually reach the smallest priority value in the queue. Suppose on the average it takes $< kz$ seconds to reach this point. Then, we divide this time interval into k time points and compute the priority of the views for each discrete time point. The overall overhead of this approach is the cost of maintaining the time queue. If a view typically remains in storage for a very short time, then k should be very small. On the other hand, if k is too small then the priority of views in the priority queue is not accurate most of the time, resulting in many comparisons for a deletion. In the worst case, the deletion time may deteriorate to $O(n)$ since all the views may have actual priority that is larger than the current priority. In this case, the whole priority queue is scanned once before any item can be removed. However, in our system, whenever a view is hit, then its priority is recomputed. If the value of a statistic such as network rate changes, then the queue is resorted. If all other functions produce smaller values with larger time values, then no view will be below sea level.

Assume now that the size of the priority queue remains stable at n . To compute the total overhead of this method, we observe the following. Whenever a view is inserted into the system, k additional copies are generated. In the worst case, the size of the time queue is n . Let's assume insertion and deletion both have the same cost $c \log n$. An additional cost of $A = c \log n$ is incurred due to the initial insertion of the ghost instance into the time queue. Since the ghost instance is removed from the queue one by one, an additional cost of $2A$ is observed at each time interval for a total of k times (giving a total of $2kA$ overhead). Since for each time instant, the view is removed from the priority queue once and reinserted again, an additional cost of $2kA$ is added. With the assumption that the priority of views decrease in time, the deletion cost remains the same, so does the cost of a hit on the storage system. As a result, the maximum total overhead of the ghost implementation is $4kA$.

Let us compare this to the cost of reordering the views. Suppose the time it takes to insert n views is given by n/λ for some arrival rate λ . Assume that the queue is reordered every p seconds, then we will reorder the queue $n/(p * \lambda)$ times. The overhead of each reorder function is given by An , hence the total overhead is $O(n^2 \log n)$.

Whereas, within this time period, the maximum overhead of the ghost implementation is $O(kn \log n)$ for some small constant k . Hence, ghost implementation is an improvement of $O(n)$ over the reordering scheme. Note that this assumes the worst case performance, not the average case performance.

3.5 Dynamic Storage Management

All of the above components of the CAVES system are static and do not change in time. However, it is unlikely that a single storage management protocol will optimize the performance in the presence of varying workloads and system parameters. To this end, the CAVES system incorporates dynamic rules that define how the general priority computation should be changed. Each rule in the system uses aggregates over the statistics of the stored views such as $\min(nr)$, $\max(size)$, $avg(cc)$, $stddev(nr)$ and constants. Each such term is called a weight change term. A weight change atom is any expression of the form $a_1 \text{ op } a_2$ where a_1, a_2 are weight change terms and $\text{op} \in \{=, <>, <, >, \leq, \geq\}$. So far, we have restricted all weight change terms to parameters that can be computed iteratively with a fixed number of variables. Hence, the overhead of maintaining these figures is negligible. A weight change rule in the CAVES system is an expression of the form:

$$A_1, \dots, A_m \rightarrow [s, w, f]$$

where A_1, \dots, A_m are weight change atoms, $s \in \{+, *\}$, w is a real number and f is the identifier for an order formula in the specific instance. The satisfaction of a weight change atom at any given time point is defined in the usual way by evaluating each function at that time point and checking the correctness of the comparison operator. A weight change rule evaluate to true at a time point t if all the atoms in the body of the rule evaluates to true at time t . The sign s in the head of a weight change rule indicates whether the change in the weight should be additive (+) or multiplicative (*). The following algorithm describes how weight changes are used within a CAVES instance:

Algorithm ChangeWeight($(r_1, \dots, r_n), W$)

- Input: r_1, \dots, r_n : a set of weight change rules, $W = \langle w_1, \dots, w_m \rangle$: a weight vector
- **for** $i = 1$ to m **do**
 - **let** $r_i \equiv A_1, \dots, A_k \rightarrow [s, w, \text{OF}_j]$
 - **if** A_1, \dots, A_k evaluates to true at time now then
 - * **if** $s = '+'$ **then** $w_j = \text{greatest}(0, w_j + w)$
 - * **else** $w_j = w_j * w$
- **let** $U = \sqrt{w_1^2 + \dots + w_m^2}$
- **for** $i = 1$ to m **do** $w_i = w_i / U$
- Return the modified weight vector *weight*

As an example, consider the order formula FNF which was defined as $FNF = \frac{1}{nr}$. This formula may not make a significant effect in the overall performance of the system in most cases. It might even have a negative impact on the performance since it tends to favor views from slow servers. If all other factors were equal, then views with the same hit ratio will be preferred over smaller views regardless of their sizes. If this results in many larger views being admitted, it will be a suboptimal solution. It is possible to house many small views instead of a single large view. However, this formula may prove useful in the extreme cases when certain servers are extremely slow and provide views that are referenced fairly frequently and do not hinder performance. In this case, it makes sense to keep these views in storage since the estimated cost of reconstructing these views is too high. This can be accomplished by a weight change rule that increases the weight of this formula whenever the minimum transfer time for one of the

servers falls below a specific threshold and decreases its weight if the minimum transfer time rises above this threshold. This can be defined as follows:

$$\begin{aligned} \min(nr) < c_1 &\rightarrow [+ , 0.1, FNF] \\ \min(nr) > c_2 &\rightarrow [+ , -0.1, FNF] \end{aligned}$$

Note that the weight changes are scaled down in the above algorithm. However, the weight of an order formula may never be a negative value. If no other rules effect the weight of *FNF*, then the above rules serve the purpose of turning *FNF* on and off based on the values c_1 and c_2 . The weight change rules in the system are defined using the above declarative language together with all the other view management methods. They are evaluated periodically. If the underlying weight vector changes as a result, then the whole priority queue is resorted according to the new priority function.

3.6 Runtime cost parameters

One of the main objectives of the CAVES system is to tune the various system parameters based on the actual performance of the system. To this end, we collect and maintain a number of runtime cost parameters (RCP for short). A number of RCPs are used to tune the dynamically changing variables of the CAVES system. The rest are maintained as system statistics and are fed to a simulation model of the system for various tests. This approach mimics the notion of cost based query optimization in relational databases. The objective in this case is to find the best combination of adjustable parameters that optimize the performance.

The RCPs that can be maintained for the runtime system are the following: the mean arrival rate of queries, total number of items in the queue on average, the hit ratio (in hits/sec), savings in time ratio, average space usage ratio, and the average rate of removal and insertion to the priority queue (in views/sec). In addition, for all order formulas that are currently active (have non-zero weight), we keep statistics to measure the general standard deviation of the values produced. The values output by these functions are normalized using their standard deviation. This way the system is not naturally biased towards formulas that produce numbers in larger range.

Since operations such as changing the overall weight vector and checking network and server rates (nr, sr) require the priority queue to be resorted, these operations should be performed rather infrequently. To determine this, we use the savings in time parameter which is measured by $\sum_{all\ requests} cc / \sum_{all\ hits} cc$. This formula disregards the overhead of using the priority queue and reading the view from disk. This is also the measure we try to optimize in our simulations of the system. To measure this, we maintain the current sum of two functions, namely $A = cc$ and $B = hits * cc$. Anytime a view V is hit, we update these values by $A = A + V.cc$ and $B = B + V.cc$. Anytime a view V needs to be retrieved from the server, we intercept the answer and only update A using the same formula. Whenever $A/B < 1.1$, which means the overall savings is below 10%, then we try to improve the performance by checking if any of the weight change rules apply to the current case.

The following are RCPs that are kept as the overall system statistics and are eventually reported to the simulation model along with the above mentioned parameters. We maintain for each query type in the system, the mean arrival rate, the average size and the average time complexity (in terms of seconds). Furthermore, we maintain the distribution of query descriptions that are requested as a histogram. If a hash function is provided for query descriptions for this type, we can build the histogram on the values of the hash function for a more precise figure. The main idea behind this is to measure the locality among the queries for a given view type. For each possible hash value, we maintain a counter on the number of requests for the items in that value. The idea is to keep a counter of all requests regardless of whether they were in the cache or not. In addition, we can build histograms on the size and time complexity values to get a more accurate picture of the distribution of requests along these axes. These

values are then used to generate workloads that approximate the actual behavior of the system. Finally, a more costly approach would be to log all requests in the system and use it to emulate the actual workload in the simulation. We expect, in most cases a histogram will provide sufficient detail for the tests.

3.7 Applications of the CAVES architecture

The CAVES architecture and the underlying storage management protocols described in this paper can be used to implement a variety of different types of protocols with small changes and additions to the overall system. In this section, we will examine some of these methods.

Semantic caching methods [5] use the notion of spatial locality to determine the priority of views. Suppose we consider range selections over a single relation. Then, two views are said to be spatially close to each other if their ranges are overlapping in space. Whenever a specific view V is referenced, then there is high probability that views that are spatially close to V will also be referenced. Hence, a hit on V increases the priority of other views in the storage system which can be implemented in a view look-up function. Whenever we search for V in our storage, we compare against every stored view. For each view, we can also output whether it is spatially close to V . Furthermore, such methods also collapse overlapping intervals which can be added as storage optimization methods that are executed when new views are admitted to the system.

Another possible use of the system is the prefetching of views that are likely to be referenced in the near future. This information can be obtained in a number of ways. We can simply keep a historical log of all past requests and mine this information periodically. We search for items that are frequently requested together within a specific time window. Whenever any of these views is requested, the storage system actively issues requests to retrieve the remaining views in that group. Another possible way to achieve this is to keep information on all possible descriptions that are requested by the system, regardless of whether they are currently in storage or not. For these view descriptions, we build a hash structure and maintain for each description a number of statistics such as total number of hits or average rate of requests. Whenever the value of one of these statistics goes above a certain threshold for some view V that is not currently in storage, the system actively issues a request for this view and attempts to admit it to the storage.

4 The simulation model and environment

The CAVES system is fully implemented in Java. Due to space restrictions, we do not go into the details of the actual system. We have tested the CAVES system against a database server and a request generator and measured the cost of various types of operations performed by the system. The simulation model used in our experiments uses a realistic model of the actual CAVES system. It simulates a priority queue, a number of statistics and order formulas over these. For each priority queue operation, i.e. insertion, lookup and deletion, it factors in the cost parameters that were measured against the actual system. In our simulation, we use five query types with varying characteristics. Each query type has a description (number of distinct possible query requests), *size*, lower and upper bounds on *complexity* and *ff* filter factor. The value $size * ff$ refers to the actual data the query has to read from disk to answer the query. The complexity measure refers to the number of times the data has to be transferred to and from disk. For example, a disk-bound sort operation would have to read the data twice and write once for the temporary step. Hence, it has complexity zero. In our query types, we use equality and range selections with very low complexity and varying filter factor, two join type operations with mid ranges for all values, and finally a high complexity query, such as a multi-pass aggregation.

In the initialization step, the simulator generates all possible requests for these query types. The values of the complexity and filter factor parameters are normally distributed over the given ranges. A workload consists of a

combination of query types with different percentages. The requests are uniformly distributed across query types with the given percentages. Within each query type, the queries are sent based on a normal distribution with a standard deviation of 20%, corresponding to the notion of locality. Hence, certain views for a given query type are expected to be requested more frequently than the others. The simulator takes as input mean arrival time for queries. The query requestes are generated at random intervals that are exponentially distributed around the mean arrival time.

Each query type is sent to a different data server with varying network characteristics. The disk speed of each server as well as the CAVES server are set to be a constant data transfer rate throughout the simulations. The network speed parameters contains minimum, maximum, mean values for data transfer rates as well as the standard deviation of the network speed over time. The changes in network transfer rates are normally distributed over the given ranges. The network speed for a server becomes constant until a change in network speed event occurs. These events occur at time intervals that are exponentially distributed around the mean arrival time for transfer time changes. Each time such an event occur, for each server, we roll a five way dice. Hence, each server changes speed with a probability of $1/5$. Anytime, the server network rates are changed, the whole priority queue is reordered.

The policies for admitting and removing views is as we have discussed in the previous sections. In our simulations, we first run the simulator once to find the standard deviation of all formulas used. Then, we run the simulator the second time, this time normalizing the values produced by these function by their standard deviations.

Each view request is first searched for in the priority queue. If the view is found, then the total time spent handling the request is incremented by the priority queue search time plus the disk transfer time. The priority of the view is updated based on the fact that a hit occurred and then the view is repositioned in the priority queue based on its new priority, and the cost of this operation is added to the overall cost. If the view is a miss, CAVES increments the time by the priority queue search time, plus the time to compute the view in the server and the time to transfer it over the network. We assume that the query server first computes the views completely and then sends them over the network. This allows us to simplify the simulation model. If the disk in the CAVES system is busy due to a read or a write operation, then then the view does not get stored and gets dropped. Otherwise if the view meets the entry criteria the view gets put in the cache based on priority. If there is not enough room in the storage other views get removed until it will fit. The disk is busy during this time period so any view requests that are missed does not get serviced. Also the time it would take for every view request to be processed without the CAVES is computed.

4.1 Experimental Setup

In our experiments, we tested a number of functions, the functions $OF = \frac{cc}{size*(now-timein+c_1)}$, $LFU = hits$, and $FNF = \frac{1}{nr}$ provided the best results. Also, since these functions do not have any factors in common, they provided a good test case. We will refer to OF , LFU , FNF as f_1 , f_2 , f_3 respectively. We ran 6800 simulation experiments using these functions. All simulations are 10^5 second runs (= 27.78 hours) with five query types and five associated servers. Each server has network rate ranging between 0.1 Kbytes/sec to 1000 Kbytes/sec, with a mean of 100 Kbytes/sec. The disk speeds used for all the servers were 2.5 Mbytes/sec and for CAVES was 5 M/sec. The changes in the standard deviation for network speed are only applied to servers serving query types 1 and 3. The tables below shows all different query types and the six workloads we have experimented with, and the remaining parameters.

QType	# Desc	Size (K)	Complexity	FF	W-0	W-1	W-2	W-3	W-4	W-5
0	1000	10 – 500	3 – 10	1 – 2	20	6	10	10	10	40
1	500	175 – 200	90 – 100	2 – 2	20	47	5	40	10	20
2	500	175 – 200	90 – 100	2 – 2	20	47	5	10	40	20
3	200	10 – 50	50 – 100	2 – 3	20	0	40	40	40	10
4	100	10 – 50	100 – 200	4 – 5	20	0	40	0	0	10

Parameter	Range
Mean arrival time for requests	One query every 1,5, and 10 sec
Mean time for network rate changes	100, and 1000 sec
Standard deviation for network rate changes	2, 5, and 10 K/sec
Storage space	1280,2560, and 5120
Weights for f_1, f_2, f_3	$w_0 = \langle 0, 0, 1 \rangle, w_1 = \langle 0, 1, 0 \rangle, w_2 = \langle 1, 0, 0 \rangle, w_3 = \langle 0, .2, .8 \rangle,$ $w_4 = \langle 0, .4, .6 \rangle, w_5 = \langle 0, .6, .4 \rangle, w_6 = \langle 0, .8, .2 \rangle, w_7 = \langle .2, 0, .8 \rangle,$ $w_8 = \langle .4, 0, .6 \rangle, w_9 = \langle .6, 0, .4 \rangle, w_{10} = \langle .8, 0, .2 \rangle, w_{11} = \langle .2, .8, 0 \rangle,$ $w_{12} = \langle .4, .6, 0 \rangle, w_{13} = \langle .6, .4, 0 \rangle, w_{14} = \langle .8, .2, 0 \rangle,$ $w_{15} = \langle .2, .2, .6 \rangle, w_{16} = \langle .2, .4, .4 \rangle, w_{17} = \langle .2, .6, .2 \rangle,$ $w_{18} = \langle .4, .2, .4 \rangle, w_{19} = \langle .4, .4, .2 \rangle, w_{20} = \langle .6, .2, .2 \rangle$

4.2 Results

In this section, we present our performance results across a wide variety of parameters. We begin our discussion with an examination of how the different weight cases effect our three major performance metrics across all five workloads. The first major performance metrics is speedup. Speedup is defined to be the overall execution of the workload without CAVES divided by the execution time of a given workload with CAVES. The second metric is hit-ratio and is defined to be the number of “hits” within the view storage system divided by the total number of accesses or requests. The last view storage system metric is “average views in cache” (AVINC). This metric denotes the number of views in the cache averaged over time.

From the outset of this work, it has been our hypothesis that a configurable storage management system that supports dynamic change rules would be of considerable performance benefit over a static storage management system. The following workload experimental data is our first proof that this would be the case.

Figure 4 plots speedup as a function of weight cases across all five workload cases. Here, we observe that the best weight case in terms of speedup varies across different workloads. For example, if we look at workload 2, which uses mostly query type 2 and 3, we observe the best range is weight cases 16 and above, however if we used those cases on workloads 0, 1 or 2 we would clearly not be achieving the optimal performance. Moreover, we observe that weight cases 10 through 15 yield good performance for workload 2, but very poor performance for workload 4, in fact the worse performance of all.

Adding to these results are the hit-ratio results presented in Figure 5. In this figure, the hit-ratio is shown as function of the weight cases across all workloads. What is interesting here is that for workloads 2 and 4, the hit-ratio is actually inversely proportional to real performance (i.e., speedup). Observe that weight cases 10 through 15 yield the highest hit-ratio, yet are not the best performance cases reported in Figure 4. Clearly, for workload 4, the best hit-ratio yields the lowest performance. The explanation for this behavior is that because all views are not of the same size, or complexity, hits may not be a good measure of performance, as done with typical processor caching systems. However, hits do play an important role in the storage management system and should not be ignored. As shown in the speedup results, it was the weight cases that utilized a weighted combination of f_1, f_2 and f_3 that yielded the best performance (recall, that $f_2 = hits$).

AVINC results are shown in Figure 6. Like the hit-ratio, we find that the AVINC performance metric not to be a

good predictor of performance for similar reasons.

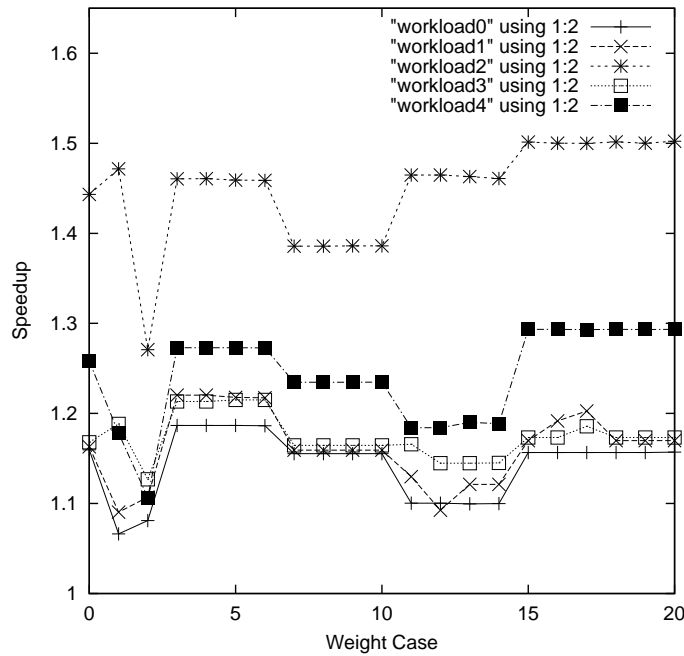


Figure 4: Comparison of speedup as a function of weight cases across all workload cases.

In this next set of performance graphs, we present the performance metrics as a function of weight cases across all cache sizes tested. In Figure 7, we observe that the best weight case does not vary with cache size. These findings are underscored by the hit-ratio and AVINC results shown in Figures 8 and 9 respectively. This means that changes to the weights is largely independent of cache size and in fact a good weight selection for a given workload may work well for many different cache sizes. This is a good finding since it allows the view storage system to optimize and self-tune the weight without regard to cache size.

Like cache size, we observe a similar behavior with mean request time with regard to when weights should be changed. In Figures 10, 11 and 12, we show the speedup, hit-ratio and AVINC results, respectively. Here, we see that the same weight case provides the best performance across all the request time cases. These results suggest that mean request time is not a factor that should be considered when deciding on when to change weight factors in the view storage management system.

The final performance data series considers the effect of the performance metrics as a function of weight cases on the change-turn-around time (CTAT) parameter. Here, we see a performance picture where CTAT does impact when weight changes should be made. In Figure 13, we see that the best speedup for CTAT 0 is under weight cases 11 through 14, however the best weight cases for CTAT 1 are 15 and above. Moreover, we observe that for CTAT 0, cases 15 and above yield significantly lower performance for CTAT 0. So while CTAT 1 could use CTAT 0's weights that opposite does not appear to be true.

If we examine, hit-ratio and AVINC, as shown in Figures 14 and 15, we find another example of where these two parameters do not predict performance. Again, this phenomenon is attributed to large size and more complex views having a great impact to be stored than smaller, easier to re-create views.

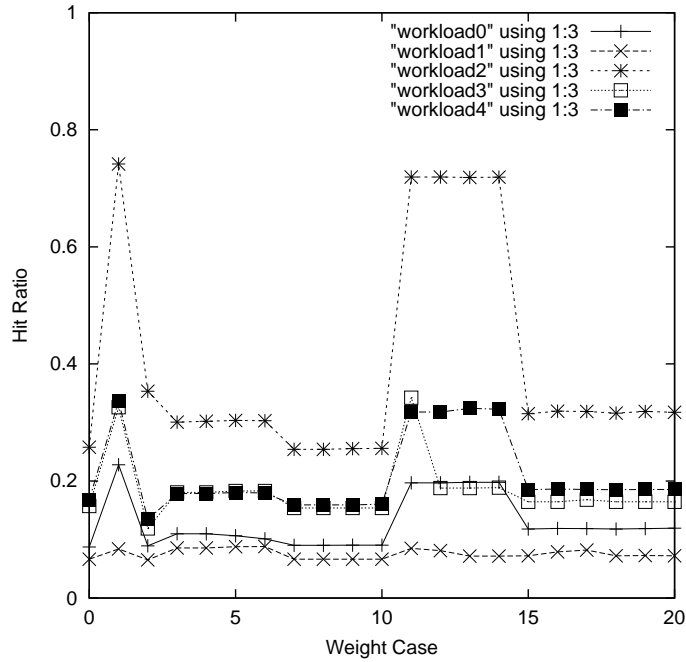


Figure 5: Comparison of hit-ratio as a function of weight cases across all workload cases.

5 Related Work

There has been a great deal of research in cache management (primary storage) methods for database systems. Such methods provide simple and hard-and-fast criteria for deciding which pages to keep in memory given their usage and size. Examples of these methods include [8, 12, 16]. A related area of research focuses on determining which pages will likely be referenced in the future and prefetching those pages. For example, Kraiss and Weikum [10] provide a comprehensive framework that decides how to move objects vertically between primary, secondary and tertiary storage systems based on a probabilistic model and usage statistics. While the CAVES system can exploit all these results, CAVES must extend them to develop storage management policies that take into account cost factors associated with a distributed architecture and dynamic parameters. As a result, the problem that CAVES addresses is different from the traditional cache management problems addressed in the papers above.

To address these different requirements, the CAVES system uses client-side caching of views. Client-side data caching and reuse is a commonly used query optimization method [3, 4, 5, 15]. In most cases, the cached data is modeled at the tuple level or at a single relation level. However, mediated systems, multimedia applications, and some data mining applications may store more complex views. See, for example, [2, 17, 18, 19]. Use of view definitions to derive alternate query plans such as query rewriting has also been an active area of research. See, for example, [6, 11, 13, 14]. While the CAVES system exploits this previous work, the integration of these algorithms with the associated storage management policies as required by CAVES has not been explored in the literature yet.

To our knowledge, there is no unifying theoretical framework for specifying dynamic view replacement policies based on a general notion of precedence. Without such a framework, it is not possible to develop a principled storage management system that handles multiple policies and that takes into account both static and dynamic factors. The CAVES system is a significant step in this direction. The two research projects other than CAVES that come closest to doing this are the cache investment strategies of Franklin and Kossmann [7] and the DynaMat system of Kotidis and Roussopoulos [9].

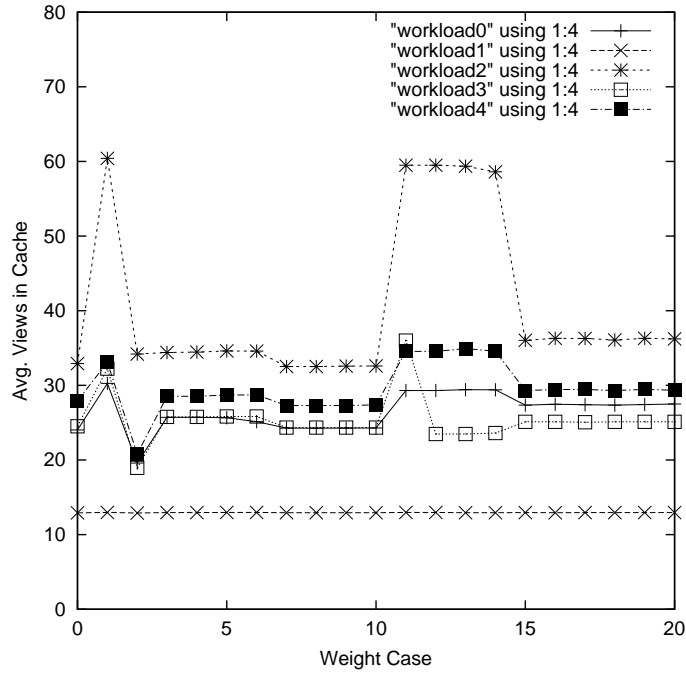


Figure 6: Comparison of average views in cache as a function of weight cases across all workload cases.

Franklin and Kossmann [7] demonstrate that caching of views in a distributed system can have a significant positive impact on query processing performance. This result is achieved with a simulation study using a variety of static and history-based caching policies. The study does not include dynamic policies that vary between client applications and query types as is the case in the CAVES system, however. DynaMat [9] uses a "goodness" measure to materialize views at multiple levels to optimize query processing for a data warehouse. It uses the incoming queries to dynamically alter the views that are materialized. DynaMat is designed for the restricted domain of a data cube in a data warehouse. It uses a fixed "goodness" measure for admitting a view into the cache and managing the cache. The CAVES system is not focused on a particular domain of applicability and emphasizes the use of dynamic measures of goodness. Nevertheless, the DynaMat system demonstrates that significant performance benefits are possible. Neither the work of Franklin and Kossmann nor DynaMat consider using simulation to dynamically tune the view management system while it runs, as is done in the CAVES system.

6 Conclusions and Future Work

In this paper, we have discussed a general architecture for disk bound storage management. We have shown that our method provides significant performance improvements over a variety of workloads with varying demands on the network and computational resources of distributed servers. We have shown that our architecture integrates view reuse methods with storage management. We have introduced the notion of dynamic change rules on storage management protocols. These rules allow the system to optimize its performance based on the runtime system parameters. We have shown with experimental results that the optimal storage management methods may vary drastically based on various factors. The key findings from the study include:

- Dynamic weight change rules appear to have a significant performance benefit.

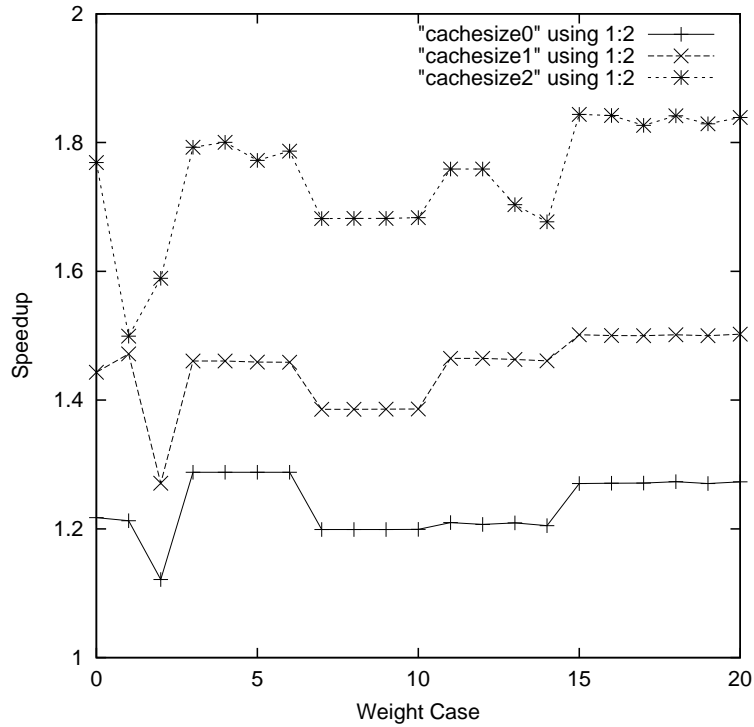


Figure 7: Comparison of speedup as a function of weight cases across all cache size cases.

- Hit counts and/or hit-ratio are important parameters for storage system, but should be considered in conjunction with other factors to yield constantly good storage system performance.
- Cache size does not appear to impact weight change decisions and in fact appear to be uniformly good across a wide range of cache sizes.
- Like cache size, mean request time does not appear to impact weight change decisions.
- Finally, change-turn-around-time (CTAT) *does* appear to impact weight changes and should be considered in the dynamic decision process.

We are currently working on automated methods for generating simulation experiments from within CAVES and developing methods to interpret the results of these experiments. These results will allow us to discover new order formulas that might provide performance gains, optimal cutoff points for dynamic change rules and new dynamic weight change rules. There are many other factors that may have a profound effect on performance such as the cost and availability of a view reuse method, the percentage view use. Such methods are vital for applications that process large data sets such as map data, data cubes, etc.. We are planning to run tests on both simulated data as well as real-life applications to develop customized storage management protocols for different scenarios. In addition, we are currently planning to parallelize the simulation model and expand its capabilities to include many clients as well as support a hierarchy of servers. This parallel simulation model will allow us to investigate enterprise level configurations, as well as rapidly explore the state space of a wide range of parameters.

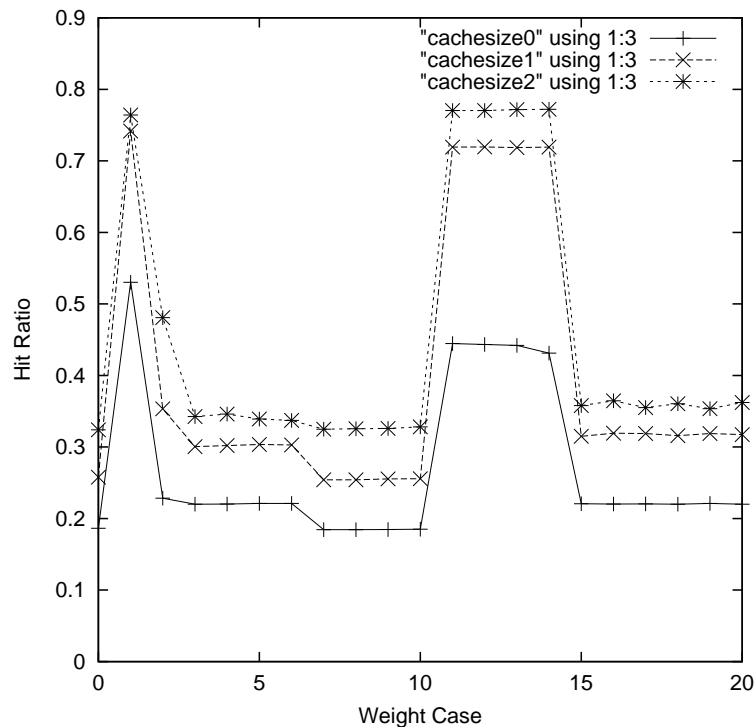


Figure 8: Comparison of hit-ratio as a function of weight cases across all cache size cases.

References

- [1] T. Blum, D. Keislar, J. Wheaton and E. Wold. "Audio Databases with Content-based Retrieval", in Proceedings 1995 IJCAI workshop on Intelligent Multimedia Information Retrieval, 1995, Montreal, Canada.
- [2] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. "The TSIMMIS Project: Integration of Heterogeneous Information Sources.", in Proceedings of IPSJ Conference, Tokyo, Japan, October 1994.
- [3] C. Chen and N. Rousopoulos. "The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching", In Proceedings of the International Conference on Extending Database Technology, 1994.
- [4] M.D. Dahlin, R.Y. Wang, T.E. Anderson and D.A. Patterson. "Cooperative Caching: Using Remote Client Memory to Improve System Performance", in First Symposium on Operating Systems Design and Implementation, 1994, pp. 267-280.
- [5] S. Dar, M. Franklin, B. Jonsson, D. Srivastava and M. Tan. "Semantic Data Caching and Replacement", in Proceedings of the 22nd VLDB Conference, 1996.
- [6] O. Duschka and M. Genesereth. "Answering Recursive Queries Using Views", in Proceedings of the Sixteenth ACM Symposium on Principles of Database Systems, 1997.
- [7] M. J. Franklin and D. Kossmann, "Cache Investment Strategies", to appear in Transactions on Database Systems, December 2000.

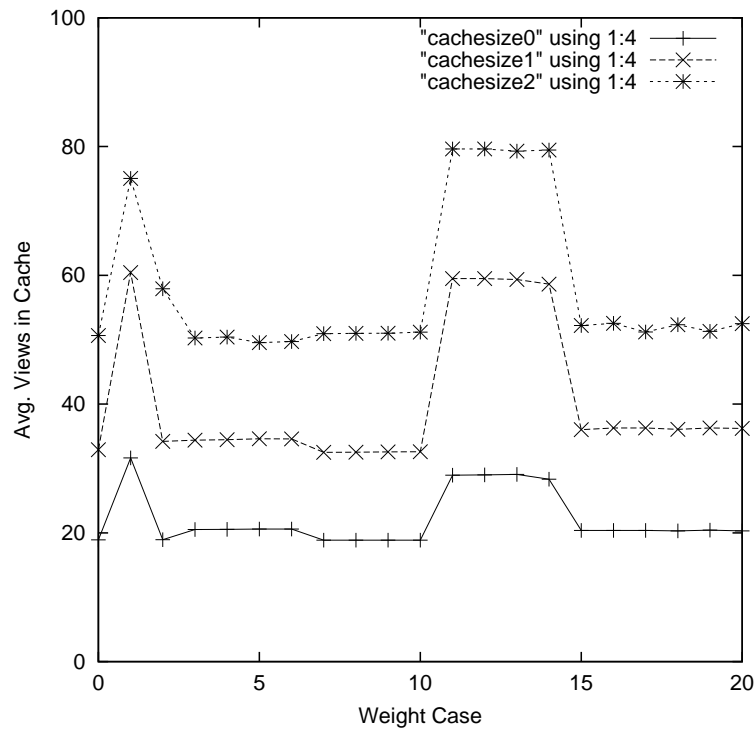


Figure 9: Comparison of average views in cache as a function of weight cases across all cache size cases.

[8] J. Gray and F. Putzolu, "The 5-Minute Rule for Trading Memory for Disk Accesses and the 10-Byte Rule for Trading Memory for CPU Time", in Proc. of the SIGMOD Conference on Management of Data, 1987, pp. 395-398.

[9] Y. Kotidis and N. Roussopoulos, "DynaMat: A Dynamic View Management System for Data Warehouses", in Proceedings of the ACM SIGMOD '99 Conference, ACM Press, 1999, pp. 371-382.

[10] A. Kraiss and G. Weikum. "Integrated Document Caching and Prefetching in Storage Hierarchies Based on Markov-chain Predictions", VLDB Journal, 1998.

[11] A. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. "Answering Queries Using Views", in Proceedings of the Fourteenth ACM Symposium on Principles of Database Systems, pages 95-104, 1995.

[12] E.J. O'Neil, P.E. O'Neil and G. Weikum. "The LRU-K Page Replacement Algorithm for Database Disk Buffering" in Proceedings of the SIGMOD Conference on Management of Data, 1993, pp. 297-306.

[13] X. Qian. "Query folding". In Proceedings of the Twelfth International Conference on Data Engineering , 1996.

[14] A. Rajaraman, Y. Sagiv, and J. D. Ullman. "Answering queries using templates with binding patterns". In Proceedings of the Fourteenth ACM Symposium on Principles of Database Systems, pages 105-112, 1995.

[15] N. Rousopoulos, C.M. Chen, S. Kelley, A. Dellis, and Y. Papakonstantinou. "The Maryland ADMS Project: Views R Us". IEEE Data Engineering Bulletin, 18(2), 1995.

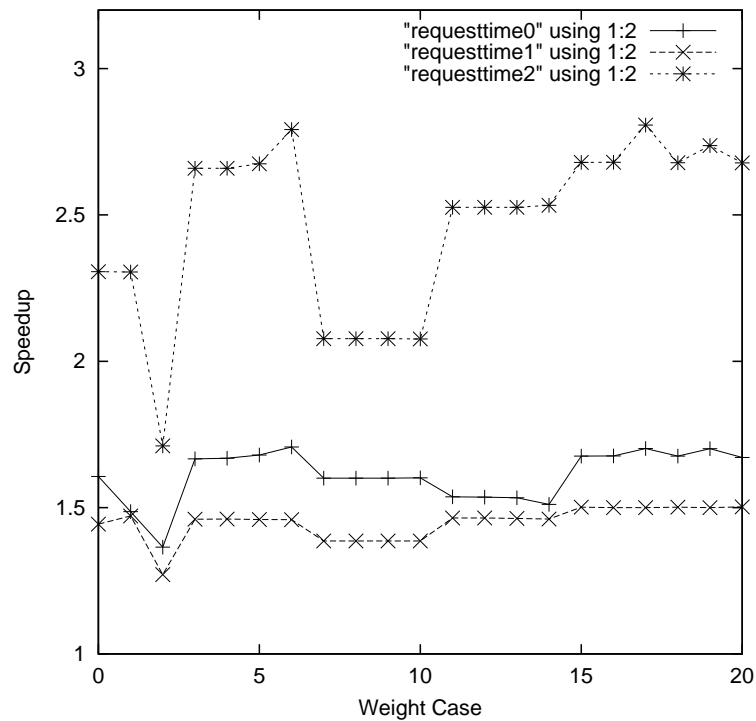


Figure 10: Comparison of speedup as a function of weight cases across all request time cases.

- [16] P. Scheuermann, J. Shim, and R. Vingralek. "WATCHMAN: A Data Warehouse Intelligent Cache Manager", in Proceedings of the 22nd VLDB Conference, 1996.
- [17] J. D. Ullman. "Information Integration Using Logical Views", in Proceedings of the Sixth International Conference on Database Theory, 1997.
- [18] G. Wiederhold. "Mediators in the Architecture of Future Information Systems", IEEE Computer, March 1992, pps 38–49.
- [19] J.W. Wong, K.A. Lyons, D. Evans, R.J. Velthuys, G.v. Bochmann, E. Dubois, N.D. Georganas, G. Neufeld, M.T. Özsu, J. Brinskelle, A. Hafid, N. Hutchinson, P. Iglinski, B. Kerherve, L. Lamont, D. Makaroff, and D. Szafron, "Enabling Technology for Distributed Multimedia Applications", IBM Systems Journal, 36(4), 1997, pages 489-507.

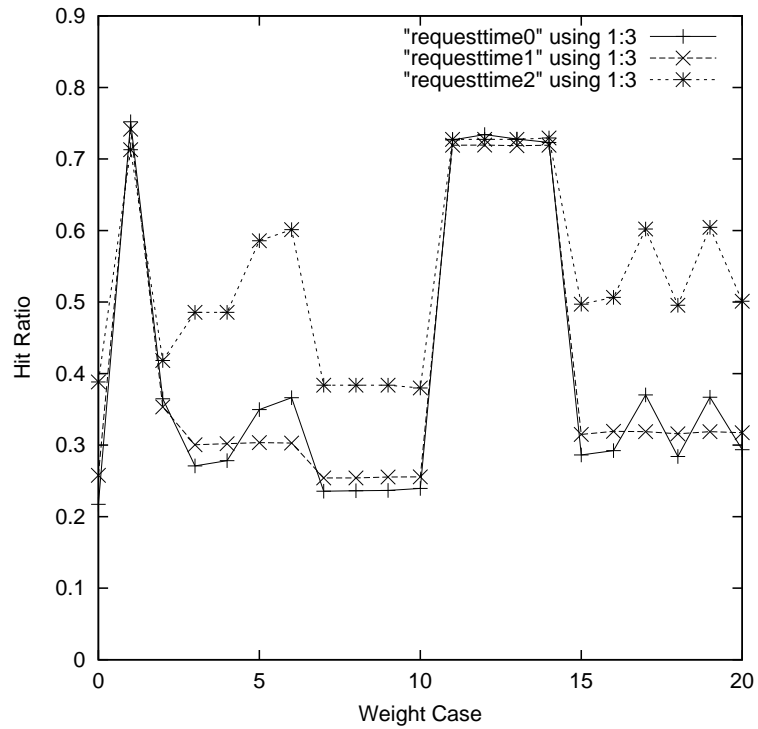


Figure 11: Comparison of hit-ratio as a function of weight cases across all request time cases.

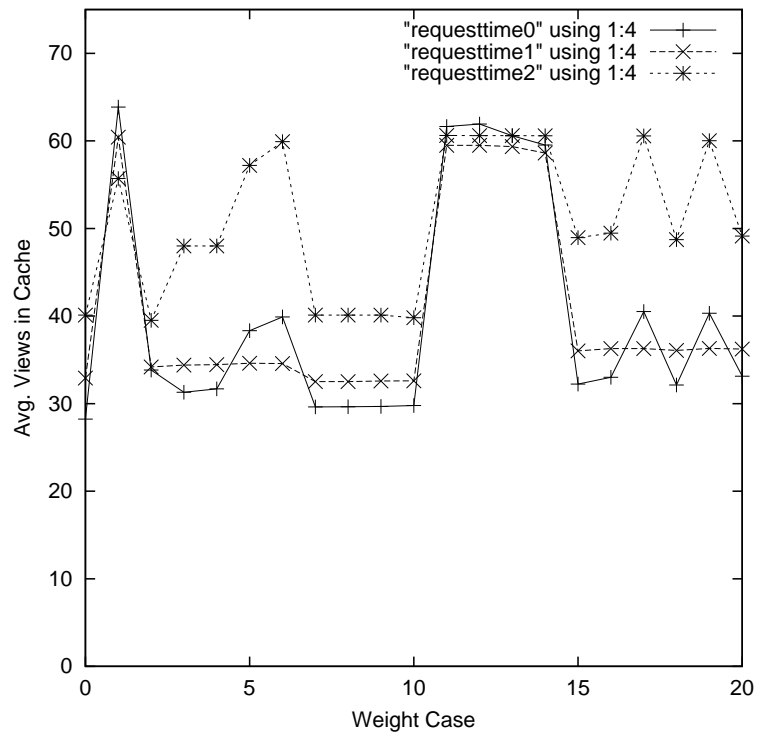


Figure 12: Comparison of average views in cache as a function of weight cases across all request time cases.

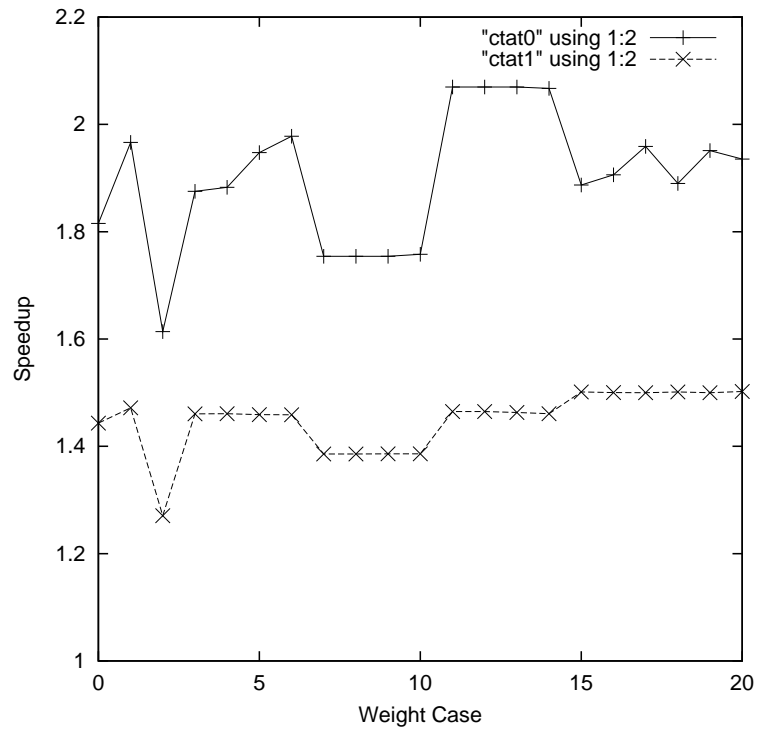


Figure 13: Comparison of speedup as a function of weight cases across all change-turn-around-time (CTAT) cases.

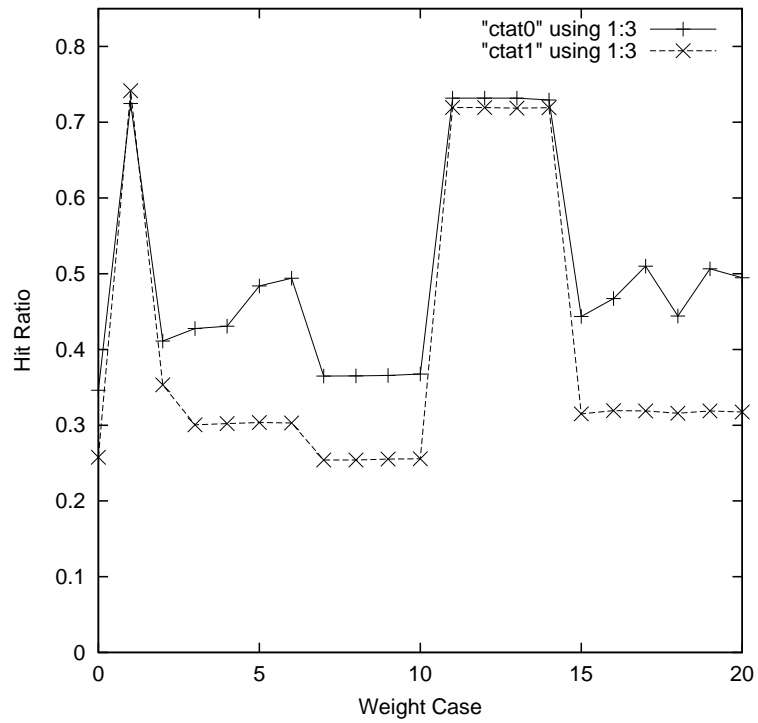


Figure 14: Comparison of hit-ratio as a function of weight cases across all CTAT cases.

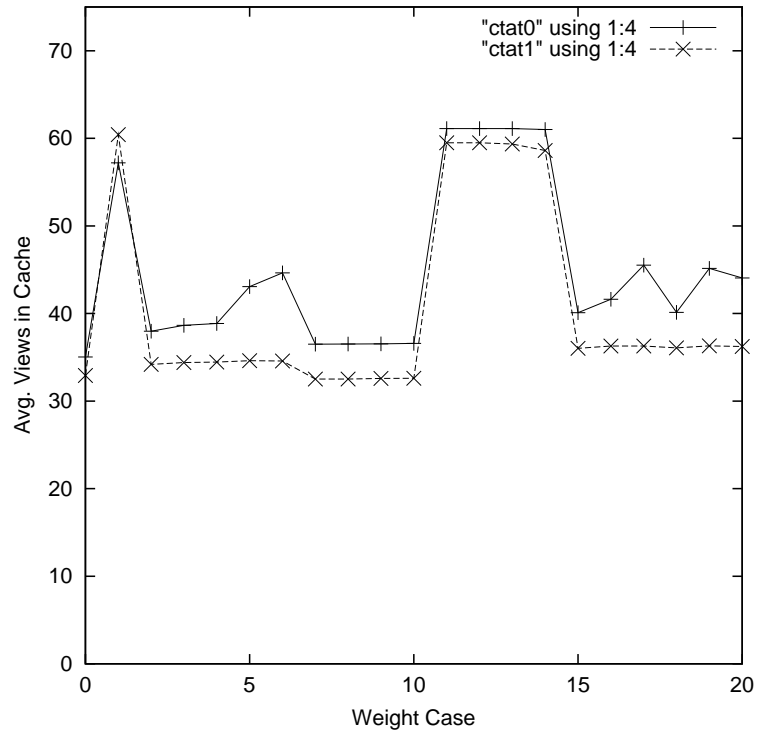


Figure 15: Comparison of average views in cache as a function of weight cases across all CTAT cases.