Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams

Sirish Chandrasekaran
EECS Department, UC Berkeley
sirish@cs.berkeley.edu

Michael Franklin
EECS Department, UC Berkeley
franklin@cs.berkeley.edu

Abstract

This paper studies Data Stream Management Systems that combine real-time data streams with historical data, and hence access incoming streams and archived data simultaneously. A significant problem for these systems is the I/O cost of fetching historical data which inhibits processing of the live data streams. Our solution is to reduce the I/O cost for accessing the archive by retrieving only a reduced (summarized or sampled) version of the historical data. This paper does not propose new summarization or sampling techniques, but rather a framework in which multiple resolutions of summarization/sampling can be generated efficiently. The query engine can select the appropriate level of summarization to use depending on the resources currently available. The central research problem studied is whether to generate the multiple representations of archived data eagerly upon data-arrival, lazily at query-time, or in a hybrid fashion. Concrete techniques for each approach are presented, which are tied to a specific data reduction technique (random sampling). The tradeoffs among the three approaches are studied both analytically and experimentally.

1. Introduction

The queries that can be posed on a Data Stream Management System (DSMS) [1,3,5] can be distinguished into three classes (see Figure 1). The first consists of queries over archived disk data that is already present in the system before the query is posed. Traditional database literature has largely focused on supporting these *historical queries*. The second class consists of queries over live network data that enters the system after the query is posed. These *live queries* have been the subject of much recent research on continuous query (CQ) processing [14]. Neither CQ engines nor traditional databases, however have adequately addressed the problem of supporting queries that access a combination of disk *and* live data. This third class of *hybrid queries* is the focus of this paper. The following are representative examples of hybrid queries, based on the scenario of a freeway embedded with sensors that record information about passing cars.

Query 1 – ***Load on Freeway***: This is a single-stream query that accesses a window of data that begins in history, and continues into the future: "*Perform a running count on the increase in number of cars between Ashby exit and the Bay Bridge since the beginning of rush hour. Compute this value once every fifteen minutes till the end of rush hour.*"

Query 2 – ***How many Commuters***: This is a join query where the the live data is combined with different portions of history: "*For the cars that have been observed to pass Ashby exit, choose those that have been seen at the same exit at the same hour every day this last week. Count the total numbers that match this condition today.*"

As the number of such hybrid queries accessing different portions of the archive increases, so do the number of random disk accesses. Given the dramatic improvements in computation power and network bandwidth witnessed recently, the cost of interleaving random disk accesses with the processing of live network data has become substantial. These I/O costs can have a debilitating effect on the ability of a query processor to run multiple hybrid queries and cause it to fall increasingly behind in the processing of the live data.

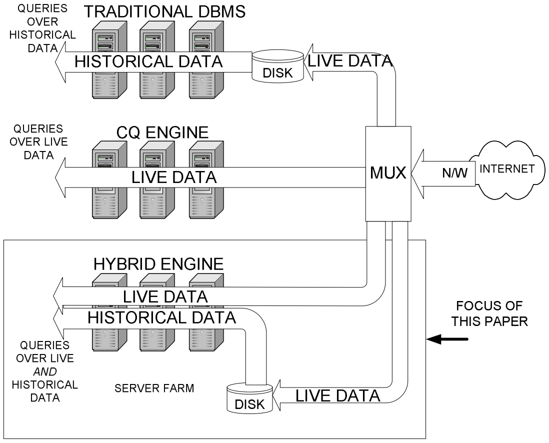


Figure 1: DSMS design following query classification

While it is tempting to reuse solutions developed in traditional DBMSs and in CQ engines to address this problem, the approaches developed in those systems to handle overload are either inapplicable or inadequate for workloads of hybrid queries. Historical queries do not typically have real time (or near real-time) requirements, allowing traditional databases the option of postponing some computation in the case of overload (e.g., data warehousing). This is, however,

This work was funded in part by the 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.

**Proceedings of the 30th VLDB Conference,
Toronto, Canada 2004**

not possible in a hybrid engine since query processing is coupled to the arrival of live data. Suspending a query only causes it to fall even further behind the live data. Further, in 24x7 applications, there might not even be a time of sufficiently reduced load at which to execute all the postponed queries. Indexing techniques combined with batched I/O can be used to support fast simultaneous inserts and reads of historical data [15,16]. However, even if the correct index is available for use by the query, it only pushes away the point at which a system is overloaded: it does not address what happens then.

Admission control at the network (also called load-shedding), on the other hand, has been proposed for CQ engines as a way to directly deal with overload [3,4]. Dropping the network data is, however, an unsuitable solution for hybrid-query workloads for two reasons. First, disk access is a more significant bottleneck for hybrid query workloads than the processing of network data. Dropping the network data therefore does not confront the root of the problem. Second, and more importantly, the dropped network data is lost forever, and unavailable for any future hybrid queries.

The key to immediately and precisely addressing overload in a hybrid query processor is to retrieve only a reduced version of the data on disk. A wealth of literature exists in the area of data reduction (henceforth abbreviated as DR) including sampling, summarization (such as aggregates, histograms) and compression. Rather than propose a new DR method, this paper focuses on the architectural issues of allowing a DSMS to exploit a variety of these pre-existing techniques to handle overload. In particular, we concentrate our efforts on designing mechanisms within the storage system to generate multiple resolutions of data reduced through random sampling or windowed aggregation. The query engine can select the appropriate level of summarization to use depending on the resources currently available.

By focusing our solution to the architectural issues concerning the storage system, rather than the DR methods or the query engine, we allow the DSMS considerable flexibility in handling overload. For example, the query engine can employ various responses to overload ranging from constant-rate data reduction to reduction based on the attribute-values of the live data or even the age of the historical data. Further, the DSMS is free to present applications with varying degrees of information about the fidelity of results ranging from nothing to sophisticated statistical guarantees.

1.1 Our solution: the 10,000 ft. view

In this section, we present an abstraction of the design of the hybrid query processing engine, and point out the portions of the system we modify to handle overload.

Figure 2 shows the three principal components of a hybrid query engine: the *network interface*, the *executor*, and the *disk*. The network interface reads the data off the network and is responsible for both making this data directly available to the executor, as well as writing it to disk. The queries run in the executor and can access the live data directly from the network interface and the historical data from the disk. In order to handle disk overload, we make the following modifications to the system.

First, all disk accesses (reads and writes) are controlled through a special access method called the *Overload-sensitive Stream Capture and Archive Reduction* access method *(OSCAR)*. OSCAR organizes the data on disk in a fashion that makes it possible to trade-off I/Os for the quality of data scanned. For example, OSCAR might store multiple versions of the stream on disk, with each version representing a different choice in the trade-off of quality and size of data. The OSCAR access method can then scan the appropriate version at query time.

The second modification to the system has to do with informing OSCAR about the degree of data reduction desired by the query for scans of historical data. To do this, we require all queries to associate each scan module over historical data with a *reduction User-Defined-Function (r-UDFs)*. As data scanned from the disk passes through the scan module at the leaf of the query plan, this r-UDF communicates back to OSCAR the degree of data reduction that is required. OSCAR uses this feedback to control the quality of data returned by the scan.

In this paper, we focus on the design of OSCARs for handling r-UDFs based on random sampling and windowed aggregation. Supporting other classes of r-UDFs is the subject of future work and discussed in Section 7.

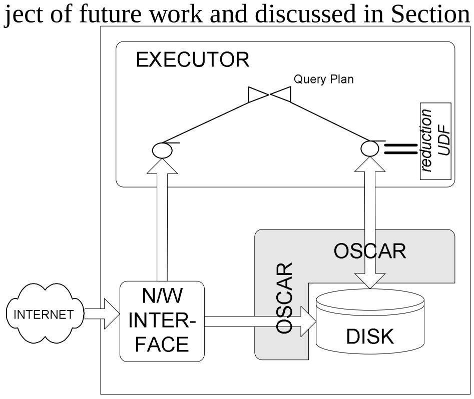


Figure 2: Our solution: the 10,000 ft. view

1.2 Contributions of this paper

In this section, we summarize the contributions of our paper.

Our first contribution is the identification of disk overload as a key problem in the support of hybrid queries that simultaneously access disk and network data, and the accompanying observation that load-shedding is an unsuitable solution to address this problem.

Second, we propose *Overload-sensitive Stream Capture and Archive Reduction* (OSCAR), an access method which can interface with a variety of user defined data reduction functions *(r-UDFs)* to retrieve reduced versions of the archive for correspondingly fewer I/Os.

Third, we present concrete OSCAR designs for r-UDFs based on random sampling and windowed aggregation. These designs differ as to when they perform most of their work generating the multiple resolutions of archived data. We describe the implementation of three OSCAR designs for r-UDFs based on random sampling in TelegraphCQ. The tradeoffs among these three designs are studied both analytically and experimentally, with an emphasis on the effects of a real file-system and operating system on the behavior of the solutions.

The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we describe OSCAR, the r-UDFs, and their interaction in more detail using an example. In Section 4, we present different OSCAR designs for both r-UDFs based on sampling and windowed aggregation. In Section 5, we describe the implementation of the mechanisms for sampling-based DR schemes in TelegraphCQ. Section 6 demonstrates and compares the performance benefits of our various storage schemes. We discuss future work in Section 7, and conclude in Section 8.

2. Related Work

Data reduction, admission control, and optimized disk access are all areas that have been extensively research by the database community. In this section, we discuss some representative and relevant work in each, discussing their applicability to overload in hybrid query workloads.

We cover the related work according to how they can be applied to the problem of disk overload.

Converting random to sequential I/O: The first important area of related work is the literature on fast inserts and reads in traditional databases. Saving random I/Os has always been a key theme of database systems research, and the goal of avoiding random I/Os in favor or sequential I/O has led to the proposal of various extent-based storage systems (e.g., [9,16]). Indexes [15] also can be used very effectively to reduce random I/O if the data is clustered. While it is difficult to ensure clustered organization of data in the presence of continually growing streams, the batching of writes of live data to disk offers an approximation. Both these approaches can reduce the impact of disk accesses; however, they suffer from some shortcomings. First, these solutions require periodic reorganization of the disk in order to make the above batches coalesce into larger ones. Second, given that it is not feasible to maintain indexes over every attribute on a high-throughput stream, and also that the queries might not be known ahead of time; the indexes maintained by the system might be inefficient for the query at hand. Finally, while these approaches increase the extent to which the system can scale, they still do not address the problem of overload, they only delay the problem.

Postponing query processing to a later time of lower load: This works for situations involving historical queries which do not typically have real time (or near real-time) requirements. A traditional database therefore has the option of postponing some computation in case of overload, for example, to the end of the business day. Examples of this can be seen in data-warehousing [13]. This is however not an option for the hybrid engine where query processing is coupled to the arrival of live data. Suspending a query only causes it to fall even further behind the live data. Further, for 24x7, high data-rate applications, it is unclear if there are any periods of lower load.

Shed load at the network: In CQ engines which neither archive nor revisit stream data, admission control at the network pipe has been proposed for dealing with overload [3,4]. As stated previously, however, this solution is not well suited to hybrid query workloads for the following reasons. For workloads involving hybrid queries, disk access is a more significant bottleneck than the main memory processing of live data: dropping data on the network pipe therefore does not directly address the current cause of overload, which is disk access. Second, and more importantly, the data dropped in the network is lost for ever, and cannot be queried by future hybrid queries.

Data Reduction (DR): There is a wealth of literature proposing different DR techniques [17]. Our work aims to complement these efforts by providing a framework where these techniques can be plugged into a DSMS. Since these techniques often require the special modifications to the executor, we restrict our focus to those based on sampling and windowed aggregation. These techniques have the dual advantage of requiring simpler enhancements to the executor, while still allowing a range of application-specific responses to the problem of overload. Other previous work, such as [8] has looked at ways to store the data on disk at multiple resolutions in order to tradeoff disk I/Os and accuracy. While we share the same goals, we are attempting to define an architectural solution that can encompass a wider range of DR techniques, and by extension, queries.

3. Our solution – the 100 ft. view

In this section, we expand on the overview of the solution we presented in Section 1.1, revealing more detail about OSCAR and the *r-UDF*. Figure 3 shows our solution for a specific disk organization and r-UDF.

OSCAR: We first discuss the most important component of our solution, the *Overload-sensitive Stream Capture and Archive Reduction* access method (OSCAR). OSCAR organizes the data on disk in a fashion that allows reduced versions to be retrieved in order to save I/Os. While the data should ideally be retrievable at any reduction level between 0 and 100% with matching savings in I/Os, in practice there can only be a finite number of reduction levels for which OSCAR can provide exactly matching I/O savings. In Figure 3, these levels are 25%, 50% and 75%. The disk therefore logically appears to have three additional copies of the stream containing 75%, 50%, and 25% of the original data respectively. The actual content of the disk is invisible to the executor and as we shall show in Section 4.1 for example, this logical view can be physically implemented using one, two or more copies of the stream on disk.

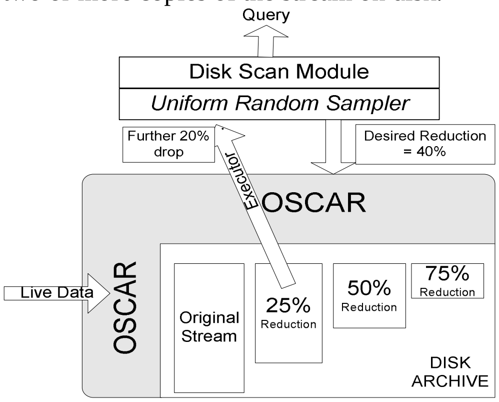


Figure 3: 100 ft. view of solution

In order to organize the data on disk and allow access at multiple levels of the reduction, OSCAR controls all disk accesses, both by the network interface and by the executor. It can therefore perform its modifications to the archive either *eagerly* when the network interface calls its *insert()* method to write the incoming data to disk, or *lazily* when the executor subsequently calls its *read()* method. In this paper, we present different approaches that perform most of their work either at data-arrival time or at query time, or split their work between the two phases. We also show how the approach chosen influences the physical implementation of the logical view of the disk.

The Reduction UDF*:* The second important part of our solution is the *r-UDFs*. Each physical scan over historical data is associated with an r-UDF that notifies OSCAR as to the degree of reduction required. The r-UDF is a piece of user code that takes in data as it is scanned from the disk and continuously returns values representing the *degree of reduction* desired in the scan. For example, an r-UDF that performs random sampling returns a value between 0 and 1 that indicates the fraction of data that can be dropped from the scan. To implement windowed aggregation, the r-UDF returns a time-interval that indicates the size of the window over which the aggregation is to be performed. The r-UDF shown in Figure 3 is a uniform random sampler that always returns 0.4; i.e., it drops 40% of the data passing through it. The r-UDF can be considerably more sophisticated. For example, an age-based r-UDF can determine the reduction level based on the age of the tuples in the section of the archive being scanned: it can thus demand increasingly higher fidelity for more recent data as in [11]. The r-UDF can also determine the level of reduction according to the current load on the system. Finally, the r-UDF can maintain internal state on the tuples that it has seen, and perform more sophisticated statistical analyses according to the desired approximation levels of the output (as in CONTROL [18]).

Putting it all together: A hybrid live/historical query is specified by associating each archived stream in the FROM clause of the query with one of the r-UDFs registered with the system. For example, the scan module over the archived stream in Figure 3 is associated with a uniform random sampler that drops 40% of all input. This level of reduction specified by the r-UDF (0.4) does not equal any of the levels of the reduction that OSCAR can directly satisfy (0.25, 0.5 and 0.75). OSCAR does the best it can and reads from the copy with 25% reduction. Though it will not save any more I/Os, the scan module can choose to drop 20% of this data, if it wishes to achieve the original target of 40% reduction ((100 – 20)% of (100 – 25)% = (100 – 40)% of original).

4. The OSCAR access method

In this section, we present various OSCAR designs that support r-UDFs based on sampling and windowed aggregation. In Section 5, we discuss the actual implementation of different OSCARs for sampling-based r-UDFs in TelegraphCQ.

4.1 OSCAR designs for random sampling

In this section, we present three different OSCAR designs to enable a variety of sampling-based r-UDFs to reduce overload caused by disk accesses in a hybrid query engine.

As shown in Figure 4, the three solutions perform their actions at different points in the system: OnWriteReplicate is an eager data-driven mechanism, performing most of its work at data arrival. OnReadModify lazily peforms most of its work modifying the archive at query time. The RandomizeThenSort method splits its work between data-arrival and query-execution.

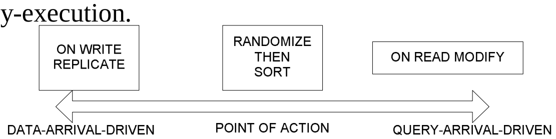


Figure 4: Different mechanisms for sampling-based DR

4.1.1 *OnWriteReplicate: The Eager Approach*

We first discuss an eager data-driven approach that exploits the fact that while random disk I/Os are expensive, disk space is not.

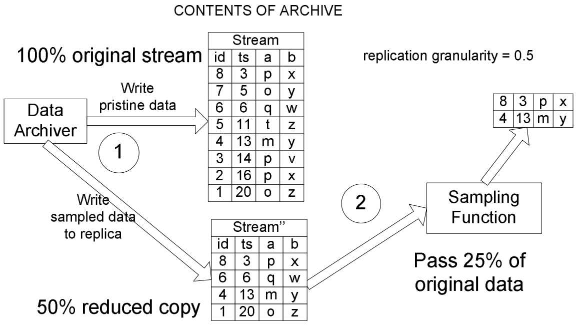


Figure 5: OnWriteReplicate in action

In this method, we always keep one pristine (non-reduced) copy of the stream on disk. This copy is referred to as the primary. In addition, there are a set of copies at increasing levels of reduction. The number and levels of reduction of these copies is specified at the time the stream is created, but can also be changed at any later time. OSCAR writes a pristine copy of the incoming data to the primary, while also randomly sampling this data at the different pre-determined levels to populate the various copies. Later, as the executor scans tuples, it continuously feeds the values returned by the r-UDF back to OSCAR. OSCAR uses these values to continuously identify (and if needed switch the scan to) the copy whose data granularity is closest to (and finer than) the desired coarseness.

Figure 5 shows an example of this technique in action. The stream has one copy (in addition to the primary) at a reduction level of 50%. At data arrival (step 1 in the figure), OSCAR populates the primary with all the incoming data, and the copy with half of this data. The r-UDF in the example specifies a uniform sample rate of 25% (i.e., a reduction level of 75%). Therefore, at query time (step 2), OSCAR reads data from the copy, rather than the primary. Note that this, however, only reduces I/Os by 50%, rather than 75%. The scan module can drop 50% of the data scanned from the copy to achieve the desired 75% reduction on the original.

4.1.2 *OnReadModify: The Lazy Approach*

In this section, we discuss a lazy, query-driven technique that performs most of its work at the executor.

One way to design this method is to maintain multiple copies of the stream at pre-determined levels and populate them on demand. This solution, however, is problematic. It increases the implementation complexity without offering significant new insight into the cost trade-offs with respect to the eager solution. To understand the increased implementation complexity, consider Figure 6, which shows the above approach in action for a stream with three copies. Different hybrid queries have caused the different copies to be filled arbitrarily. Each copy thus has sequences of extents of tuples followed by *holes* corresponding to the portions of the archive that have not yet been accessed by any hybrid queries.

Because of the arbitrary order in which these holes are filled, and the potentially unknown sizes of the tuples that will fill them, we can no longer use a simple heap file to store the copies; an extent-based storage structure must be used for each copy of the stream.

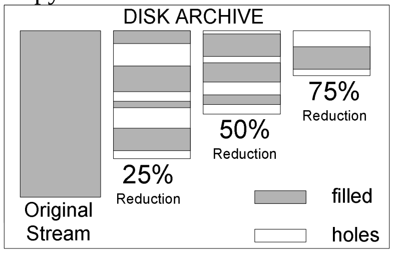


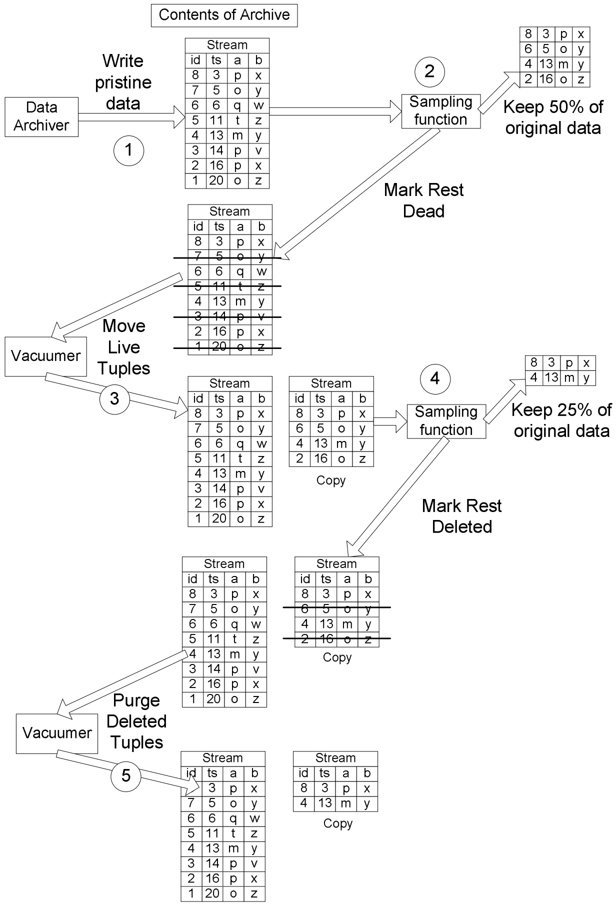
Figure 6: Effect of eager filling on pre-defined copies

A database like PostgreSQL does not have an inbuilt extent-based storage method, requiring one to be implemented from scratch. Further, the above solution would also require a managing data structure that continuously directs scan to the smallest possible copy while also filling these copies on demand. For these reasons, we have developed a solution that uses a simpler storage structure: a heap file for the original stream, and another for a reduced copy. The resulting design is different from the above mirror of the eager solution. We describe our solution below for the case of one r-UDF. In the presence of multiple r-UDFs, the solution is just repeated in parallel, with multiple copies.

In this technique, at data-arrival time, OSCAR simply streams a complete copy of the arriving data to disk. Later, at query time, OSCAR randomly marks some of the tuples that are scanned as *dead,* at the reduction rates returned by the r-UDF. These tuples do not make it to the query plan. A periodic *vacuumer* process later copies the remaining *live* tuples to a new location on disk. Future incoming tuples are written to both the original and to this copy. Subsequent disk accesses by the same or a different query using the same r-UDF are then directed to this new, smaller copy so they will incur fewer I/Os. As before, this will result in OSCAR marking some of the tuples in this copy as *dead*. Instead of moving them to yet another copy, the vacuumer just purges the tuples from this copy and compacts it in-place.

If the reduction level of any portion of this copy becomes too low for a new query using the same *r-UDF*, then the new query switches to scanning the original, with no further modifications to the archive. If the fraction of accesses that switch to the original crosses a preset threshold, then the copy can be deleted and the whole process repeated.

Figure 7 illustrates this approach. In Step 1, OSCAR writes all the data to disk. At query time (Step 2), let us assume that the sampling r-UDF requires a 50% reduction in data. These tuples are then marked dead in the original. At Step 3, the vacuumer copies the remaining live tuples to a new location on disk. A future query (shown in Step 4) uses the same r-UDF over this stream. At this time, however, the r-UDF requests a 75% reduction on the original data. This can be satisfied entirely by scanning only the copy. The vacuumer, however, had already removed 50% of the original data in constructing this copy. We therefore, only retrieve (25/50)*100 = 50% of the current copy. The remaining tuples are purged from the copy on a subsequent pass by the vacuumer.



Stream

id	ts	a	b
8	3	p	x
7	5	o	y
6	6	q	w
5	11	t	z
4	13	m	y
3	14	p	v
2	16	p	x
1	20	o	z

8	3	p	x
6	5	o	y
4	13	m	y
2	16	o	z

Stream

id	ts	a	b
8	3	p	x
~~7~~	~~5~~	~~o~~	~~y~~
6	6	q	w
~~5~~	~~11~~	~~t~~	~~z~~
4	13	m	y
~~3~~	~~14~~	~~p~~	~~v~~
2	16	p	x
~~1~~	~~20~~	~~o~~	~~z~~

Stream

id	ts	a	b
8	3	p	x
7	5	o	y
6	6	q	w
5	11	t	z
4	13	m	y
3	14	p	v
2	16	p	x
1	20	o	z

Stream

id	ts	a	b
8	3	p	x
6	5	o	y
4	13	m	y
2	16	o	z

Copy

8	3	p	x
4	13	m	y

Stream

id	ts	a	b
8	3	p	x
7	5	o	y
6	6	q	w
5	11	t	z
4	13	m	y
3	14	p	v
2	16	p	x
1	20	o	z

Stream

id	ts	a	b
8	3	p	x
~~6~~	~~5~~	~~o~~	~~y~~
4	13	m	y
~~2~~	~~16~~	~~o~~	~~z~~

Copy

Stream

id	ts	a	b
8	3	p	x
7	5	o	y
6	6	q	w
5	11	t	z
4	13	m	y
3	14	p	v
2	16	p	x
1	20	o	z

Stream

id	ts	a	b
8	3	p	x
4	13	m	y

Copy

Figure 7: OnReadModify in action

4.1.3 In RandomizeThenSort: The Hybrid Approach

The solutions described in the previous sections perform significant amounts of work either at data-arrival or query-time in order to enable sampled access to disk data. Depending on the query and data workloads, they might even exacerbate the degree of overload. The method we discuss in this section fixes this problem by spreading the burden across these two phases and improving the performance of both reads and writes.

In this technique, there is a single copy of the stream on disk, which is divided into separate "runs" or batches, with each "run" corresponding to a fixed number of blocks. The data source writes the arriving tuples uniformly at random to one of the blocks in the current run. When any of the blocks in the current run fills up, the entire set of the blocks in the run is flushed to disk and a new run is created for inserting new tuples. At query time, when OSCAR begins to scan a new run of blocks, it uses the latest value returned by the r-UDF to only read from a corresponding fraction of the blocks in this new run. Since the tuples within each block are sorted by timestamp, a merge sort is employed to retrieve the original order of tuples across the different blocks in the run.

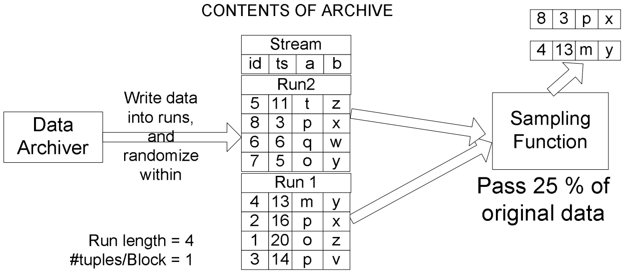


Figure 8: RandomizeThenSort in action

Figure 8 shows an example of this technique in action. The stream is defined to have runs of size 4 blocks each (In the above figure, we assume one tuple per block for ease of visualization). The r-UDF specifies a 75% level of reduction. Therefore, OSCAR reads exactly block from each run.

4.1.4 Comparison of the solutions

Having discussed our three basic algorithms, we briefly compare them analytically. In Section 6, we will compare them experimentally. Table 1 summarizes the analytically computed I/O costs of three OSCAR designs and a strawman that performs all the sampling in the executor, does not modify the archive, and saves no I/Os.

As can be seen, OnWriteReplicate, always maintains the original data, and in fact, pays a higher insertion cost by maintaining extra copies in the hope of saving I/Os at query time. It however pays a low cost on every read.

OnReadModify, on the other hand, performs most of its work on the first time it scans an archive. Both its original write costs, and subsequent read costs are restricted to the minimum possible. It does however require additional work by the vacuumer. The fact that it removes tuples from the copy can be an asset. If the solution is modified to allow the vacuumer to purge tuples from the original itself, it can be used to limit the size of the archive.

RandomizeThenSort is a hybrid solution that has both low write and read costs, with the #I/Os at write time being the same as that for the OnReadModify solution, and the #I/Os at read time very close to the minimum possible #I/Os.

4.1.5 Analytical Plots

In order to better visualize the formulae in Table 1, we use them to analytically compute and plot the write and read rates for the various OSCAR designs.

First, we parametrize the eager and hybrid OSCAR designs as follows. We assume that OnWriteReplicate has four copies in addition to the original stream at reduction levels of 20%, 40%, 60% and 80%. This determines the function *g(r)* in the table. For the hybrid solution, we assume a run length (*R*) of a hundred blocks.

To determine the values of the other parameters in the analytical formulae, we ran some tests on our implementation of OSCAR in TelegraphCQ. We measured the size on disk of a stream (based on the strawman described above) containing 20 million tuples to be 108,000 blocks of 8192-bytes each. Since the size of each index entry is only 20 bytes, we approximate I/S to be zero. Therefore, B = 108,000 blocks. We also measured the time it took to populate the strawman with these tuples to be 430 seconds.

Based on the above parameters, Figure 9 displays a generated plot that shows the analytically computed time for the different OSCAR designs to write the same stream. As can be seen, the hybrid solution, RandomizeThenRead, pays the same cost as the strawman, requiring very little work at data insertion time. The lazy solution takes the same time as the strawman and hybrid solution prior to vacuuming. After being vacuumed, however, it contains an additional copy that also receives all the incoming data, doubling the write cost. The eager solution is computed to take the most time, since the sum of the sizes of its copies is twice the size of the original stream. The cost of writing the stream for the eager solution is thus thrice that of the strawman.

Mechanism	**Cost of Writes**	**Cost of reads**
OnWrite Replicate	$\sum_{i=1}^{n} B_j(1+\frac{I}{S})$	$\sum_{i=1}^{B} g(f(i))$
OnRead Modify	No copy: $B(1+\frac{I}{S})$ 1 copy: $2*B(1+\frac{I}{S})$	1^{st} Read: $B+\sum_{i=1}^{B}\lceil 1-f(i)\rceil$ 2^{nd} Read: B Post Vac.: $\sum_{i=1}^{B} f(i)$
Randomize ThenSort	$B(1+\frac{I}{R*S})$	$\sum_{j=1}^{B/R}\lceil R*f'(j)\rceil$
Strawman	$B(1+\frac{I}{S})$	B

B = # blocks of stream in archive
I = size of index entry
S = size of a block
R = run-length
n = # copies of stream
		f(i) = 1 – min(reduction level for tuples in i^{th} block)
		f'(j) = 1 – min(reduction level for tuples in j^{th} run)
		g(r) = 1 – (reduction level amongst copies closest to r)

Table 1: I/O costs of storage mechanisms

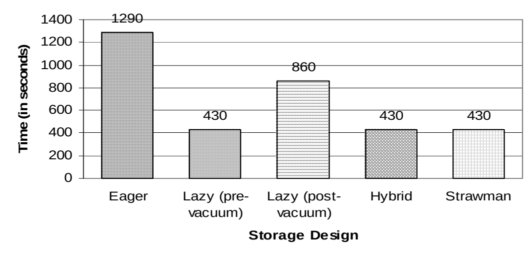


Figure 9: Time to insert 20 million tuples

Next, we ran a *count(*)* query over the entire strawman archive; this took 20 seconds. Based on this measurement, we computed the time it would take to query the data for different values of the reduction level (*f(i)*) for the different OSCAR designs. Figure 10 shows the plot for these generated values.

In the figure, the first and second reads for the lazy design are assumed to occur before the vacuuming. In order to reduce the visual clutter, we group together designs that have very similar or identical performance. The performance of the strawman is independent of the reduction level. As can be seen, the first query pays a high price for OnReadModify, the lazy solution. The second query performs as well as the strawman. After vacuuming, the lazy solution matches the hybrid solution, and has the best possible performance. The cost of the eager solution is a step function, since the OnWriteReplicate solution can only offer savings commensurate to the granularities of its pre-stored copies. It has much lower cost than the pre-vacuuming lazy solution. It still has comparable, but slightly higher cost relative to the post-vacuum version of the lazy solution.

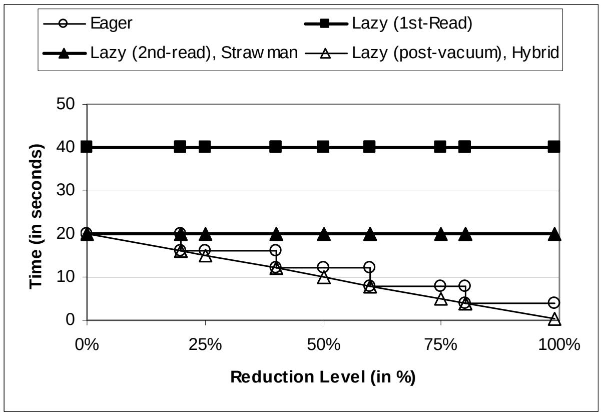


Figure 10: Time to query 20 million tuples for different reduction levels

The analysis of the performance of the system when the write and the reads occur simultaneously is as follows. Since the disk is the bottleneck, the available disk bandwidth will be parcelled out between the queries and the insertion process at the rate at which they access the disk. Therefore, the total time for insertion will increase in proportion to the number of disk accesses by the queries for a given query-completion rate. As we shall see in Section 6, however, reality is quite different: the disk elevator algorithm used in Linux (and the ext3 file system) removes contention between the writes and reads, by guaranteeing each a minimum latency. Further, the operation system unfairly schedules the write process more if it does not realize that the queries are performing sequential I/O.

In this analysis, the hybrid OSCAR design emerges as the best solution because it offers both low insert and reduces I/Os at exactly the rate demanded by the r-UDF.

4.2 OSCARs for windowed aggregation

In Section 4.1 we described storage schemes for sampling-based r-UDFs. In this section, we will discuss a mechanism for supporting *summary-based* techniques such as grouped averages.

We propose solutions here similar to the lazy (OnReadModify) and eager (OnWriteReplicate) methods described in Section 4.1. Like those other techniques, these two work by storing multiple copies of the data at different degrees of summarization. The main difference is that unlike sampling, decisions about the reduction are now made on a per-window rather than per-tuple basis. This observation leads us to our key insight in supporting windowed-aggregation: the r-UDFs in this case can be abstracted away as a query on the window of tuples being summarized. The results of this query can then be written to the replicas.

To drive the rest of the discussion, we use the following example of an r-UDF over a stream of stock-ticker data.

Example windowed aggregation r-UDF: For each hour of data, return a tuple per distinct stock symbol containing the average value of that symbol over that hour. This r-UDF is a tumbling-window [1] grouped-average query.

4.2.1 The Eager Approach

As before, OSCAR writes one complete copy of the data on disk, and other reduced versions of the data. The number of these other versions is specified at stream creation time, as are windowed-aggregation queries that populate them. Copies can also be added and removed at a later time.

As an example, consider the stock ticker and r-UDF presented above. Let us also assume that the stream is created with two additional copies: the first (copy#1) stores the hourly averages and the other (copy#2), the daily averages of the stock data, both grouped by company symbol.

OSCAR writes the individual records to the master copy of the stream on disk. In addition, it populates these two additional copies using either triggered historical queries, or tumbling window continuous queries. The trigger approach requires one purely historical copy for each of the copies that have to be populated. Copy#1, is therefore populated by a grouped average query that is triggered every hour and executes over the previous hour of data. Copy#2, is populated by a similar query that is triggered every day. The CQ approach, on the other hand has two permanently running grouped average continuous queries, one with a window that tumbles every hour, and the other with a window that tumbles every day. In either case, a new query that uses the r-UDF described above (which requires hourly averages) can scan Copy#1 to reduce overload.

4.2.2 The Lazy Approach

As with OnReadModify, there is a single copy of the complete data on disk, and one additional reduced copy per r-UDF. At data-arrival OSCAR simply writes the complete data to disk. At query time as data is scanned from disk, it is passed to the through the tumble query in the r-UDF. The result of this DR query is passed on to the user query. These results are also written back to disk in a new temporary location on disk. Each of these tuples is also encoded with information about the time window to which it corresponds. A vacuumer process can later combine the data from the original with the tuples in this temp location to create a new copy, replacing those original tuples that were scanned through the r-UDF with the summarized version in the temp copy. Future queries can then access this new copy. As with OnReadModify, modifications to this copy are made in place. Again, as in the case of OnReadModify, the reduction level of this copy might be too low for a certain query, in which case the original must be scanned.

In summary, the design of eager and lazy OSCAR mechanisms for r-UDFs based on windowed aggregation is essentially the same as that for random-sampling. They only differ in their details, to accommodate the different properties of the input and output to these r-UDFs.

5. Implementation in TelegraphCQ

We implemented the above OSCAR designs for sampling-based r-UDFs in TelegraphCQ, the CQ extension to PostgreSQL v7.3. The implementation closely follows the description in Section 4.1; therefore we only describe our experiences that were specific to building within TelegraphCQ (and PostgreSQL). In the rest of this section, we discuss the implementation of hybrid queries in TelegraphCQ, the custom vacuumer and free space map we built for OnReadModify, the implementation of stream copies and the special tuple format used for OnReadModify.

5.1 Running hybrid queries in TelegraphCQ

We first describe the changes we make to TelegraphCQ to run hybrid queries. These include the addition of an access method that retrieves archived streams, and modifications to the symmetric join operator to allow the executor to combine network and disk data properly.

The access method underlying OSCAR consists of an append-only heap file clustered by timestamp, and a sparse BTree on the timestamp attribute. This design allows both efficient archiving and scans by hybrid queries. Most streaming systems assume that incoming streams are in timestamp-order (modulo a maximum finite skew that can be handled through a reorder buffer). The append-only timestamp-clustered heap file therefore allows for very efficient inserts. Further, hybrid queries such as those shown in Section 1 require access, in increasing timestamp order, to all tuples with timestamp greater than a certain historical value. The Btree can be used to efficiently locate the starting point, after which a sequential scan on the heap file can be used to retrieve further tuples.

The second change we made to TelegraphCQ to support hybrid queries was to its executor, which uses an eddy and SteMs[19] to implement main-memory symmetric joins. Since TelegraphCQ only supports sliding window band joins, the SteMs only maintain a limited main memory window on the each input stream. In order to extend this mechanism to disk data, we force the eddy to coordinate the rate of data access from the network and the disk for queries that access both the network and disk. This prevents the disk scan from outstripping the rate of arrival of live data during periods of low data arrival rate.

5.2 Changes to PostgreSQL

We now discuss implementation details relating to PostgreSQL components and data structures.

The vacuumer: While PostgreSQL v7.3 has two vacuum modes, both are inapplicable in the streaming scenario, and we had to implement the vacuumer for OnReadyModify from scratch.

The *full* vacuuming mode requires a lock on the entire relation. This is clearly unsuitable for unbounded streams, since it blocks read and write accesses to the stream for a potentially unbounded time. Further, the full vacuumer destroys the original ordering of the tuples during compaction of the relation heap files. While this does not matter for relations, we care about preserving the ordering of tuples in a stream according to timestamp, as described in Section 5.1.

The other vacuum mode, called *lazy*, does not compact the relation: it only removes the dead tuples in-place. The uncompacted heap file is of no use to us as it cannot be used to save any I/Os upon future scans. Also, this vacuum mode performs no index maintenance (PostgreSQL removes index entries for tuples as soon as they are marked dead in the executor). We found that index deletion operation as performed in PostgreSQL is very expensive; invoking it in the fast path during query time in the executor therefore severely impacts the performance of the system. Hence, we move all index maintenance to the vacuumer.

The Free Space Map: A key data-structure in implementing vacuuming is the free space map (FSM) which records the available space on each page. PostgreSQL has a free space map, which however, is soft-state, and even worse, possibly stale. We therefore implement our own free space map that uses fast Judy arrays [7] to record for each stream which pages have atleast one dead tuple (these need to be processed by the vacuumer), and how much free space is available on each page (to determine which pages to compact).

5.3 Miscellaneous implementation details

Stream copies: The multi-copy streams in the lazy and eager solutions are implemented in a straightforward manner. Each copy is stored in an append-only heap file, and has an index on the timestamp attribute as described in Section 5.1. Creation, destruction, write and read calls to the stream are conducted through wrapper functions that are responsible for managing the copies.

Tuple format for OnReadModify: Finally, we discuss the unique tuple attribute requirements of OnReadModify. Recall from Section 4.1.2, that the copy stream might not exist at a uniform level of reduction. Rather, different portions of the stream might be reduced to different levels, according to the run-time reduction levels demanded by the r-UDF. To allow OSCAR to decide if the copy can be used or not to satisfy a certain query, we add an implicit attribute to each tuple in the stream, called *samplerate* that indicates the level of reduction for each tuple. This overhead of 25% to the 32 byte header of a TelegraphCQ tuple can be reduced even further if we sample all the tuples in a block at the same rate and store that granularity in the disk block header.

6. Performance

In this section, we test our central thesis that reducing I/Os at the disk scans is an essential tool for hybrid engines if they wish to support high insert rates in the presence of numerous hybrid queries. We describe the setup for the experiments in Section 6.1. In Section 6.2, we examine the insertion times for the different OSCAR designs for sampling-based r-UDFs in the absence of any queries. In Section 6.3, we look at the query-times for these schemes in the absence of concurrent insertions. In Section 6.4, we study the performance of simultaneous queries and data archiving. We summarize our results in Section 6.5.

6.1 Experimental Setup

In this section, we first explain our data and query workload and then describe the hardware and software environment for our experiments.

6.1.1 *Data and Query Workloads*

In addition to the three OSCAR designs described in this paper for sampling-based r-UDFs, we also measure the performance of a strawman in which all the sampling takes place in the executor. We tried three variants of our hybrid RandomizeThenSort solution with run-lengths of 10, 100 and 500 blocks respectively. For the OnWriteReplicate solution, we replicated the original stream into four additional copies at reduction levels of 20%, 40%, 60% and 80%.

OnWriteReplicate, OnReadModify and RandomizeThenSort are referred to as Eager, Lazy and Hybrid in the plot legends. Suffixes are used to indicate different states or parametrizations for the eager and hybrid solutions. We now describe the tuples that populate the streams and the queries over them.

Input Data: For the strawman and for the eager and hybrid OSCAR designs the input tuples have the schema "(float8 *timestamp*, int4 *userattr*)", where *userattr* is a dummy attribute. As explained in Section 5.3, the input tuples to OnReadModify have an additional *samplerate* attribute of type float8. The value of this attribute is set to 1 on entry (indicating pristine data). The *timestamp* attribute for tuples of all streams is set by the network interface to the time of entry.

Queries: Since our experiments focus on OSCAR, and not the executor, we use very simple count(*) landmark queries (e.g. *Query 1* in Section 1) rather than hybrid join queries (e.g. *Query 2* of Section 1). As we shall see in Section 6.4, however, the behavior of the OS affects the processing of live data in a way that is also applicable to join queries.

6.1.2 *Execution Environment*

Machine and OS: All the experiments were run on a Linux box (kernel 2.4.18) with a 1390 MHz Pentium III processor with 512 KB cache size. The machine had 512 MB RAM and 2 GB swap space. The underlying file system was ext3.

One Seagate Cheetah 36ES disk using the SCSI Ultra 160 interface was dedicated to our experiments. The response time of the disk is 5.2 ms, its peak data transfer rate is 320MBps, and it has an internal buffer of 4MB. The observed raw throughput through the file system is much lower at about 60MBps.

PostgreSQL Settings: We initialized PostgreSQL to have a buffer pool with 4096 frames. The size of each frame (and also disk pages) is 8K bytes. The buffer replacement policy is LRU. We modified our vacuumer to be invoked on demand, rather than execute periodically. This allows us to generate repeatable observations. We also disabled logging because we only had access to one disk to store our database. This is acceptable because most real systems write log records to a separate disk.

6.2 Inserting into the Archive

In this section, we present an experiment that measures the insertion-performance into streams for the three OSCAR designs and the strawman.

To perform this test, we measured the time it took to insert 20 million tuples into a stream for each OSCAR design. Data was generated online by a C program and piped to the data archiver. The total overhead generated by this data generator was less than a half second, and is negligible. At the end of the insertion of the 20 million tuples, we flush the buffer pool and sync the filesystem buffers to ensure that we measure the complete cost to write and disk the relation.

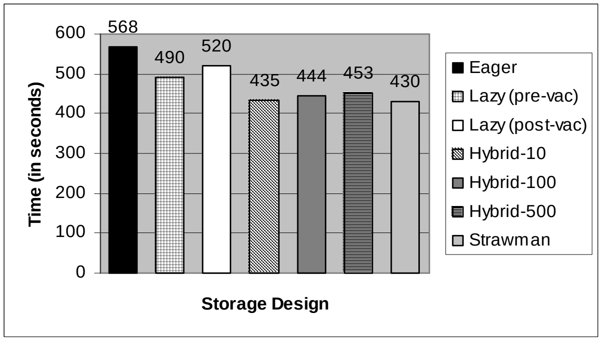


Figure 11: Insertion Times for 20M tuples

Figure 11 shows the time taken to insert the data into the various schemes. The three RandomizeThenRead solutions (marked as Hybrid in the legend) write the stream at about the same rate as the strawman. The slight differences between the three can be attributed to the different amounts of wasted space in the different runs of the streams. As the run size increases, so does the wasted space, increasing the size of the stream that must be written to disk. OnReadModify ('Lazy' in the plot) is a little slower than the strawman and hybrid solutions since the extra samplerate attribute results in increased total #I/Os. As the tuple payload increases, this overhead will disappear, and OnReadModify will approach the performance of the strawman. The cost of replication in OnWriteReplicate ('Eager') is sub-linear in the total size of the original stream and its copies. This is due to the amortization of the overhead of the processing in the wrapper clearing house. From these results, it might seem that the strawman is better than the eager and lazy OSCAR designs. As we shall see in the next section, however, the strawman performs poorly at query time under overload overload that OSCAR is intended to solve.

In terms of absolute bandwidth, the insertion rates are much lower than can be supported by the disk. It might thus seem that the network interface is the bottleneck, not the disk. As we shall see shortly, however, with multiple queries, disk accesses rapidly becomes a problem.

6.3 Querying the Archive

In this section, we present an experiment that measures the time taken for a *count(*)* query over the entire archive for the three OSCAR designs, and also the strawman. We repeat this test for different reduction levels to study the response of the different designs to overload. Figure 12 shows the results of the experiment.

The strawman uniformly takes around 20 seconds because it scans the same number of blocks irrespective of the reduction level. The size of each tuple is around 50 bytes; the strawman therefore runs near the peak 60MBps disk bandwidth available at the application level.

For each of the reduction levels, the first query on the OnReadModify stream (Lazy-1st read in plot) has the greatest cost of all the schemes. This is because it pays a write cost in addition to the read cost for all buffers it dirties. Thus, even at 99% reduction, the method continues to pay almost twice the cost of the strawman, since it dirties at least one tuple in each page. On the second pass through the OnReadModify stream, the write cost does not have to be paid again, and the cost of this solution drops almost to that of the strawman. After the vacuuming has been performed, the copy is smaller than the original by a fraction equal to the reduction level. The cost of subsequent scans drop dramatically, by the same fraction.

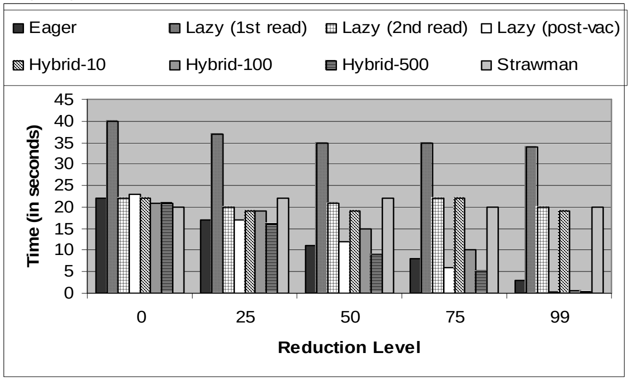


Figure 12: Query time for different reduction levels

The cost of the query for the OnWriteReplicate design drops with the reduction level. It is slightly more expensive than the post-vacuuming version of OnReadModify, since the method can only reduce I/Os according to the granularities of the pre-stored copies.

The different hybrid alternatives offer the most interesting results. Based on the earlier theoretical analysis, we would expect all the hybrid solutions to have improving query times as the reduction level increases. This improvement is, however, always realized only for the hybrid design with run-length of 500. For some of the reduction levels, Hybrid-100 also shows improvement in performance. Hybrid-10, however, is never better than the strawman. The reason for this has to do with the pre-fetching and buffering in the file system. For the hybrid design with run-length 10, and for low reduction levels for Hybrid-100, the blocks that are read are so close on disk that the file-system ends up performing a sequential read, instead of only picking exactly the required blocks. As a result, in these cases, the hybrid solution performs no better than the strawman.

6.4 Putting Insertions and Queries together

In this section, we test our central hypothesis that disk accesses have a significant impact on the insertion process in a hybrid stream query processor.

To conduct this test, we measure the time taken to insert 20 million tuples into a stream in the presence of simultaneous *count(*)* queries over N other streams which each have previously had 20 million tuples inserted into them. We then vary N to measure the effect of an increasing number of simultaneous queries. Since query rates are much higher than insertion rates (see Sections 6.2 and 6.3), the queries complete much before the insertions are completed. To maintain a constant number of simultaneous queries we start a new query on a stream, once the previous one completes.

Measuring the insertion rate by itself, however, does not tell us anything, since the insertion-performance can be artificially inflated if the queries are not scheduled at all. Therefore, for each of the above readings, we also measure the corresponding number of queries that are completed in the time it took to insert the 20 M tuples.

Figures 13 through 16 show these measurements for 25% and 75% reduction (we also measured these values for reduction levels of 0%, 50% and 99%; the trends are similar). Figures 13 and 15 show the insertion time and the total queries completed in that time for varying number of simultaneous queries, for 25% reduction. Figure 14 and 16 show the corresponding figures for 75% reduction.

Let us first analyze Figures 13 and 15. In the absence of any archived queries, the strawman can populate the stream in about 430 seconds. As we increase the number of queries, the insertion-time increases, but once we reach 3 archived queries, it stabilizes at around 1500 seconds. Looking at the corresponding numbers in Figure 15, we see that the total number of completed queries increases from 13 for a single query to about 25 for two queries, and then stabilizes at around 30. This number does not increase, or decrease significantly with increasing number of simultaneous queries.

This pattern is also observed for all the OSCAR designs. This curious phenomenon is explained by looking at the disk-elevator algorithm used by the system. The algorithm sets separate latency limits for reads and writes. This means that hybrid queries cannot affect the write process beyond a certain limit. Initially, the insertion process is not disk-bound (see Section 6.2). With increasing number of queries, however, the disk bandwidth available to the insertion process reduces, reducing its performance. However, once the disk subsystem is saturated for reads, the write process cannot be affected further. Therefore, the write throughput stays constant after a certain number of simultaneous queries. Further, since the same read bandwidth has to be shared amongst the concurrent queries, no matter how many of them there are, the total number of completed queries also stays fixed.

Figures 13 and 15, however, also suggest that for the strawman, if the total number of simultaneous queries is increased past the early-thirties, the system cannot run them at the rate of data arrival. This is because with all queries running at the same rate, no single query will actually be able to reach completion in the time it takes to insert the 20 M tuples. This therefore represents a bound on the number of simultaneous hybrid queries in system before it starts to fall behind.

The OnReadModify stream (‘Lazy’ in the figures) performs marginally poorer than the strawman solution prior to vacuuming, both with respect to the insertion times and the number of completed scans. This is because the sizes of the tuples are larger than in the other solutions, as explained in Section 6.1.1. The real savings with this mechanism kicks in after the stream has been vacuumed. The size of the archive is reduced by a quarter and the total number of completed scans increases correspondingly. Since the overhead of the network interface is amortized between the original and the new copy, the write performance for the lazy solution is largely unaffected from the pre-vacuum version.

The OnWriteReplicate method (‘Eager’) has higher insertion times than the others, as expected. But again, the cost of the extra writes is sub-linear in the size of the stream, and the increase in insertion time is not significant. The number of completed queries is almost as good as the post-

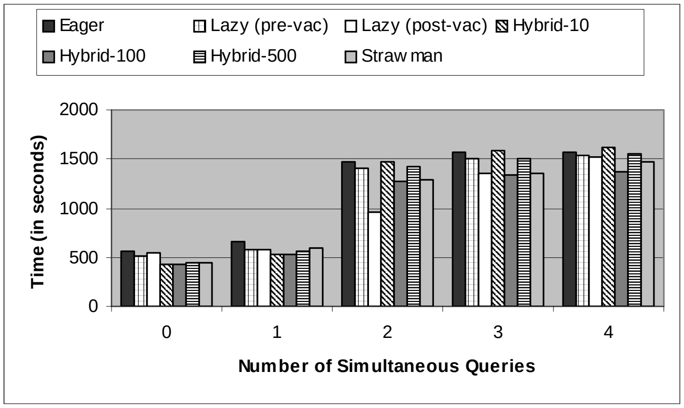


Figure 13: Insertion time for 20M tuples, 25% reduction

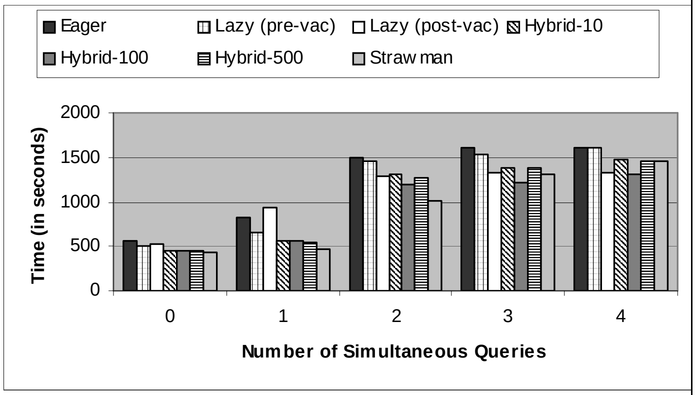


Figure 14:Insertion time for 20M tuples, 75% reduction

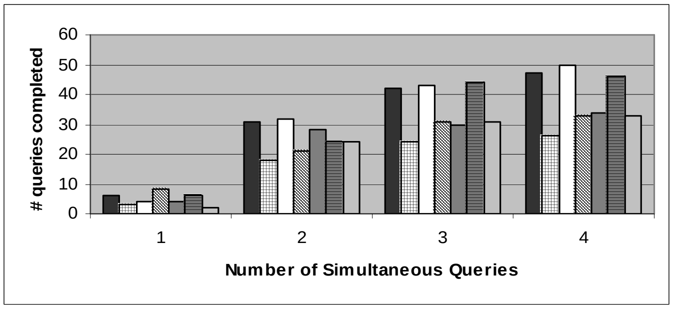


Figure 15: #queries completed within 20M tuples insertion, 25% reduction

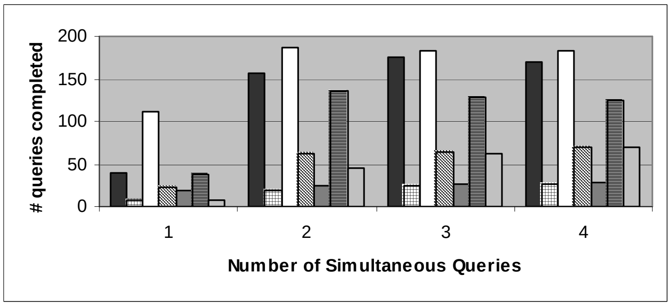


Figure 16: #queries completed within 20M tuples insertion, 75% reduction

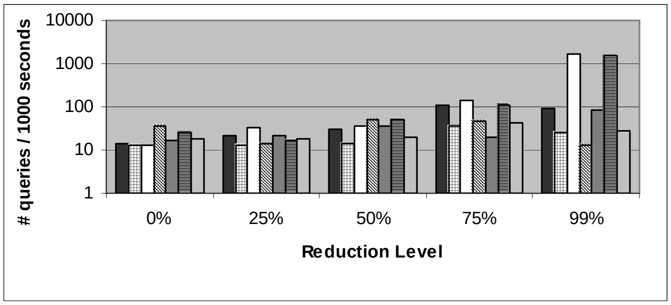


Figure 17: Queries completed/1000 sec, 2 simultaneous queries

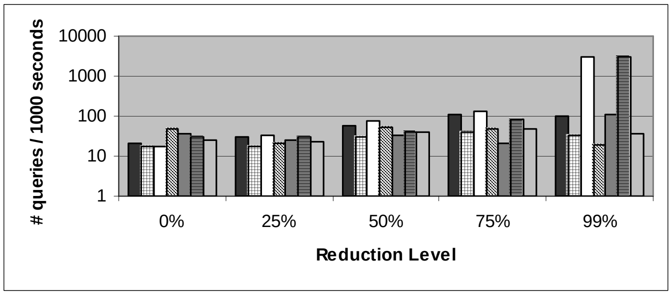


Figure 18: Queries completed/1000 sec, 4 simultaneous queries

vacuumed lazy solution. It is slightly lower, since this solution can only offer performance improvement in steps according to the reduction levels of its pre-stored copies.

All the hybrid solutions have insert performance approximately as good as the strawman. At query time, however, the hybrid solutions display interesting behavior. As explained in Section 6.3, at 25% reduction, Hybrid-10 and Hybrid-100 are essentially performing sequential scans because of pre-fetching and buffering by the file system. Their performance numbers are therefore similar to the strawman. Hybrid-500 on the other hand, does save some physical I/Os, leading to higher queries completed.

Now, let us look at Figures 14 and 16. These show the insertion-time and number of queries completed in the meanwhile, for 75% reduction. The insertion rates in Figure 14 are largely the same as in Figure 13 because of the above-mentioned effect of the disk scheduler. In Figure 16 (note that the scale is different from Figure 14), the number of completed queries increases only slightly for the strawman, since its actions are independent of the reduction level. The slight increase corresponds to the available bandwidth till the disk saturates. The same is true for the OnRead-Modify solution.

The post-vacuum version of OnReadModify, and On-WriteReplicate have significantly higher completed queries than in Figure 14 because of the greater reduction level. Of the two, the eager version is slightly better since it tracks the reduction level exactly, while the eager solution only offers reduction at preset rates.

As before, the hybrid design offers the most interesting results. At 75% reduction, the hybrid-10 continues to perform sequential scan (since it still reads 3 out of every 10 contiguous blocks). Hybrid-100 is, however, reading fewer blocks now. Instead of showing improved performance, the number of completed queries actually decreases. The reason for this lies with the OS process scheduler. When the OS guesses that a query is not performing sequential I/O (in our system this happens if two blocks that are accessed consecutively are more than 8 blocks apart), it gets scheduled less often. The number of physical I/Os saved by Hybrid-100 is not sufficient to overcome this scheduling bias. Hybrid-500, however, does save enough physical I/Os (since it skips 375 blocks at a time) and has more completed queries as compared to the 25% reduction.

We also ran the above tests for reduction levels of 0%, 50%, and 99%. Figures 17 and 18 (note the log-scale on the y-axis) give a flavor of the results, showing the number of queries completed per 1000 seconds for the different reduction levels. The eager solution stops providing improved performance once the reduction level is less than its smallest copy. The lazy solution and Hybrid-500 continue to support more queries with greater reduction levels.

6.5 Summary of Results

In this section, we summarize the results of our performance study. The behavior of the file-system prevents us from directly proving our central hypothesis that the processing of live data suffers with increasing number of hybrid live/historical queries. On the other hand, the results shown in Figures 15 through 18 demonstrate the performance benefits of the different OSCAR designs for hybrid queries in the presence of overload.

The strawman, which samples only in the executor, has excellent write performance, but does poorly when overloaded. Among the OSCAR designs, the hybrid solution offers the most benefit. It has low write costs and offers excellent I/O savings with different reduction levels provided the run-length is suitably large to prevent interference from the file-system and OS process scheduler. The lazy approach has even better response to overload than the hybrid solution, but only after it is vacuumed. The eager solution performs well as long as the desired reduction levels are close to those of the pre-stored copies.

These performance results underscore an additional important point: the OS kernel and its various policies have a tremendous effect on the benefits, or lack thereof, of storage level solutions. For example, pre-fetching and buffering in the file system and the OS process scheduler cause our hybrid OSCAR design to show widely different behavior depending on the size of the runs. On a more global level, the disk scheduling algorithm on the platform we used for our experiments limited the interference between read and write accesses to disk.

7. Future Work

There is much interesting work to be done in the area of supporting hybrid stream queries. One avenue for further work in the area of overload handling is the extension of the framework presented in this paper to include more classes of data reduction. Another is to provide overload handling for index access methods just as OSCAR does for scan-based disk access. We are currently engaged in studying a framework for making run-time decisions as to which indexes are updated on data-arrival, and handling the resulting “holes” in other indexes that do not get built. Other challenges involving hybrid query include intelligent buffer management that can exploit the predictable access patterns of long-running windowed queries. Finally, current techniques [20,21,5] for shared computation of stream queries rely on all queries processing the same tuple at the same time. This is not possible for hybrid queries that access different portions of the archive; a new soluton is therefore needed.

8. Conclusion

Random disk accesses in a DSMS that processes hybrid queries can have a detrimental effect on the performance of a system. Under sufficiently high load, the system might fall further and further behind in its processing of the stream. We therefore propose that such systems should support means to deliver reduced versions of historical data to these queries to ease the disk load on the system. In this paper we propose the *Overload-sensitive Stream Capture and Data Reduction (OSCAR)* access method to achieve this. We discuss different designs of OSCAR to handle data reduction based on random sampling and windowed aggregation. We describe its implementation in TelegraphCQ for doing this, and present experimental results validating our thesis.

9. References

[1] Carney et al., “Monitoring Streams - A New Class of Data Management Applications”, In VLDB 2002

[2] Madden et al. “Continuously Adaptive Continuous Queries”, In SIGMOD 2002

[3] Motwani et al. “Query Processing, Approximation, and Resource Management in a Data Stream Management System”. In CIDR 2003.

[4] Tatbul et al., “Load Shedding in a Data Stream Manager”, In VLDB 2003.

[5] Chandrasekaran et al. “TelegraphCQ: Continuous Dataflow Processing for an Uncertain World”. In CIDR 2003.

[6] TelegraphCQ source code: http://telegraph.cs.berkeley.edu/

[7] Judy Arrays sourceforge project: http://judy.sourceforge.net/

[8] Lazaridis I. and Mehrotra, S.. “Progressive Approximate Aggregate Queries with a multi-resoltion tree structure”. In SIGMOD 2001.

[9] Roesenblum et al, “The Design and Implementation of a LogStructured File System”, ACM TOCS 1991

[10] Manku and Motwani, “Approximate Frequency Counts over Data Streams”, In VLDB 2002.

[11] Bronimann et al, “Efficient DR Methods for On-Line Association Rule Discovery”.

[12] R. Read et al. A multi-resolution relational data model. VLDB 1992

[13] S. Chaudhuri and U. Dayal: An Overview of Data Warehousing and OLAP Technology. In SIGMOD Record (1): 65-74 (1997)

[14] L. Golab and M. T. Özsu: “Issues in data stream management”. SIGMOD Record(2): 5-14 (2003)

[15] P. Muth et al.. “The LHAM Log-Structured History Data Access Method”. VLDB J. 8(3-4): 199-221 (2000)

[16] M. Overmars: “The design of dynamic data structures”. LNCS 1983

[17] Barbara et al. “The new jersey data reduction report”. Data Engineering Bulletin, September 1996

[18] J M. Hellerstein, et al. Informix under CONTROL: Online Query Processing. Data Min. Knowl. Discov. 4(4): 281-314 (2000)

[19] V. Raman et al. Using State Modules for Adaptive Query Processing. In ICDE 2003.

[20] Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In SIGMOD 2000.

[21] S. Chandrasekaran and M. Franklin. Streaming Queries over Streaming data. In VLDB 2002