Paper Number 1059

Efficient Snapshot Differential Algorithms for Data Warehousing

Wilburt Juan Labio, Hector Garcia-Molina

Abstract

Detecting and extracting modifications from information sources is an integral part of data warehousing. For unsophisticated sources, in practice it is often necessary to infer modifications by periodically comparing snapshots of data from the source. Although this *snapshot differential problem* is closely related to traditional joins and outerjoins, there are significant differences, which lead to simple new algorithms. In particular, we present algorithms that perform (possibly lossy) compression of records. We also present a *window* algorithm that works very well if the snapshots are not "very different." The algorithms are studied via analysis and an implementation of two of them; the results illustrate the potential gains achievable with the new algorithms.

1 Introduction

Warehousing is a promising technique for retrieval and integration of data from distributed, autonomous and possibly heterogeneous information sources [Squ95]. A warehouse is a repository of integrated information that is available for queries. As relevant information sources become available or as relevant information sources are modified, the new information is extracted from the sources, translated to the data model of the warehouse, and integrated with the existing warehouse data. In this paper, we focus on the detection and the extraction of the modifications to the information sources.

The detection and extraction of modifications depends on the facilities at the source. If the source is sophisticated, say a relational database system with *triggers*, then this process is relatively easy. In many cases, however, the source does not have advanced facilities available for detecting and recording modifications (e.g. *legacy* sources). If this is the case there are essentially three ways to detect and extract modifications [IC94]:

1. The application running on top of the source is altered to send the modifications to the warehouse.
2. A system log file is parsed to obtain the relevant modifications (as done in the IBM Data Propagator [Gol95]). Since log files are used for recovery, this approach may not require any modification to the application.

3. The modifications are inferred by comparing a current source snapshot with an earlier one. Typically, the snapshots used are the same ones generated for backup, so this approach may not require modification to the application either. We call the problem of detecting differences between two source snapshots the *snapshot differential* problem; it is the problem we address in this paper.

Although the first two methods are usually preferred, both methods have limitations and disadvantages. The first method requires that existing code be altered. In most cases, however, the code is so shopworn that additional modifications are problematic. Since the modifications are recorded as they happen, this method also entails extra processing on top of normal operations. The second method also has its difficulties. For instance, it is often the case that DBA privileges are required to access the log, so site administrators are reluctant to provide access. Moreover, log files often have a format that is hard to decipher and DBMS vendors are usually not willing to disclose it. It may also be the case that the source does not even have (or need) a log. The third method is used in practice when the other methods do not apply. Some commercial products, such as the Prism Warehouse Manager [IC94], provide support for all the methods. However, as far as we know, there are no published papers detailing the algorithms used by the commercial systems.

We stress that we are *not* arguing in favor of snapshot differentials as the best solution for reporting modifications to a warehouse. It clearly does not scale well: as the volume of source data grows, we have to perform larger and larger comparisons. We are saying, however, that it is a solution we are stuck with for the foreseeable future (until sophisticated database systems become universal), and because differentials are such inherently expensive operations it is absolutely critical that we perform them as efficiently as possible. In this paper we will present very efficient differential algorithms; they perform so well because they exploit the particular semantics of the problem.

1.1 Problem Formulation

We view a source snapshot as a file containing a *set* of distinct records. The file is of the form $\{R_1, R_2, ...R_n\}$ where R_i denote the records. Each R_i is of the form $< K, B >$, where K is the key and B is the rest of the record representing one or more fields. Without loss of generality, we refer to B as a single field in the rest of the paper. (In [LGM95] we extend the algorithms presented in this paper to the case where records do not have unique keys.)

For the snapshot differential problem we have two snapshots, F_1 and F_2 (the later snapshot). Our goal is to produce a file F_{OUT} that also has the form $\{R_1, R_2, ...R_n\}$ and each record R_i has one of the following three forms.

1. $< Update, K_i, B_j >$

2. $< Delete, K_i >$

3. $< Insert, K_i, B_i >$

The first form is produced when a record $< K_i, B_i >$ in file F_1 is updated to $< K_i, B_j >$ in file F_2. The second form is produced when a record $< K_i, B_i >$ in F_1 does not appear in F_2. Lastly, the third form is produced when a record $< K_i, B_i >$ in F_2 was not present in F_1. We refer to the first form as *updates*, the second as *deletes* and the third as *inserts*. The first field is only necessary in distinguishing between updates and inserts. It is included for clarity in the case of deletes.[1]

Conceptually, we have represented snapshots as sets because the physical location of a record within a snapshot file may change from one snapshot to another. That is, records with matching keys are not expected to be in the same physical position in F_1 and F_2. This is because the source is free to reorganize its storage between snapshots. Also, insertions and deletions may also change physical record positions in the snapshot.

The snapshot differential can be performed at the source itself. That is, a snapshot is taken periodically and stored at the source site. A daemon process then performs the snapshot differential periodically and sends the detected modifications to the warehouse. The snapshot differential can also be performed at an intermediate site (as is done in the *WHIPS* [HGMW$^+$95] data warehousing system we are building in Stanford). That is, the source sends the full snapshots to an intermediate site where the snapshot differential process is performed. In any case, the exact procedure for sending these modifications to the data warehouse is implementation dependent. One way of sending the information is to produce the file F_{OUT} in its entirety. After it is produced, a message is sent to the data warehouse (using TCP/IP say) for each record in F_{OUT}. Based on the form of the record, either an update, insert or delete message may be sent.

It is important to realize that there is no unique set of modifications that captures the difference between two snapshots. At one extreme, a deletion can be reported for each record in F_1 and an insertion can be reported for each record in F_2. Obviously, this can be wasteful. We capture this notion of wasted messages by defining *useless delete-insert pairs* and *useless insert-delete pairs*. A useless insert-delete pair is a message sequence composed of $< Insert, K_i, B_i >$ followed (not necessarily immediately) by $< Delete, K_i >$, produced when the two snapshots both have a record $< K_i, B_i >$ or when the earlier snapshot has $< K_i, B_j >$ and the later one has $< K_i, B_i >$. A useless insert-delete pair introduces a correctness problem. When the insert is processed at the warehouse, it will most likely be ignored since a record with the same key already exists. Thus, when the delete is processed, the record with the key K_i will be deleted from the warehouse. On the other hand, a useless delete-insert pair (which is composed of the opposite sequence) does not compromise the correctness of the warehouse. However, it introduces overhead in processing messages since either no modifications were needed (when the two snapshots both have a a record $< K_i, B_i >$) or the modification could have been reported more succinctly by $< Update, K_i, B_i >$.

Since useless pairs are not an effective way of reporting changes, one may be tempted to require snapshot differential algorithms to generate *no* useless pairs. However, strictly forbidding useless delete-insert pairs turns out to be counterproductive! Allowing the generation of "some" useless delete-insert pairs gives the differential algorithm significant flexibility and leads to solutions that

[1]In some applications, we may also want to filter out some modifications that we know in advance not to be of interest to the warehouse (e.g., only cancer patient data is collected at the warehouse). However, for simplicity, we assume that all of the modifications are relevant to the warehouse.

can be very efficient in some cases. We return to these issues later when we quantify the savings of "flexible" differential algorithms over algorithms that do not allow useless delete-insert pairs. Thus, in this paper we do allow useless delete-insert pairs, with the ultimate goal of keeping their numbers relatively small.

However, we do want to avoid useless insert-delete pairs since they may compromise correctness. Useless insert-delete pairs can be eliminated by batching the deletes together and sending the deletes first to the warehouse for processing. In essence, we have transformed the insert-delete pairs into delete-insert pairs. This method also amortizes the overhead cost of sending the modifications over the network. Another method for eliminating useless insert-delete pairs is to record the modifications detected in a file. A second pass can then be performed over the file to eliminate the useless pairs altogether. Since the size of the file is probably much smaller than the snapshots, this pass will not be too expensive. We assume for the rest of the paper that all useless insert-delete pairs are eliminated by one of the methods just outlined.

1.2 Differences with Joins

The snapshot differential problem is closely related to the problem of performing a join between two relations. In particular, if we join $F1$ and $F2$ on their common K attribute on the condition that their B attributes differ, we can obtain the update records required for the differential problem. However, the join does not capture the unmatched deleted and inserted records. An outerjoin, however, can generate the inserts and deletes, although the resulting tuples will not be in the desired format (they will have all fields of both relations, some with null values).

Still, join and outerjoin are so closely related to the differential problem that the traditional, *ad hoc*, join algorithms ([ME92],[HC94]) can be adapted to our needs. Indeed, in Section 3 we show these modifications. However, given the particular semantics and intended application of the differential algorithms, we can go beyond the ad hoc solutions and obtain new and more efficient algorithms. The three main ideas we exploit are as follows:

- As discussed earlier, some useless delete-insert pairs are acceptable. In the context of outerjoins, a useless delete-insert pair is equivalent to "reporting" two tuples as "dangling" when they actually have matching keys. Traditional outerjoin algorithms do not have useless delete-insert pairs. The extra flexibility we have allows algorithms that are "sloppy" (but very efficient) in matching records.

- For some data warehousing applications, it may be acceptable to miss a few of the modifications, especially if these "errors" are very infrequent. For example, if the warehouse is used for statistical analysis or data mining, missing one sales record out of billions may be acceptable. Thus, for differentials we can use probabilistic algorithms that may miss some differences (with arbitrarily low probability), but that can be much more efficient. Again, traditional algorithms are not allowed any "errors," must be very conservative, and must pay the price.

- Snapshot differentials are an on-going process running at a source (or intermediate source).

This makes it possible to save some of the information used in one differential to improve the next iteration. Traditional join algorithms typically do not take advantage of data structures created during other joins (other than existing general purpose indexes).

1.3 Outline

The rest of the paper is organized as follows. Section 2 briefly reviews related research in the literature. We then present how the ad hoc join algorithms can be extended to perform differentials in Section 3.1. We present the record compression techniques to reduce snapshot size in Section 3.2 and show how these techniques can be used with the ad hoc outerjoin algorithms. In Section 4, we introduce our *window* algorithm, representing a second class of efficient differential algorithms. The algorithms are analytically compared in Section 5.1; we report on the implementation and evaluation of some of the algorithms in Section 5.2. We conclude the paper in Section 6.

2 Related Work

Snapshots were first introduced in [AL80]. Snapshots were then used in the system R* project at IBM Research in San Jose [Loh85]. The data warehouse snapshot can be updated by maintaining a log of the modifications to the database. This approach was defined to be a *differential refresh* strategy in [KR87]. If snapshots were sent periodically, this was called the *full refresh* strategy. [KR87] proposed two logging methods for the *differential refresh* strategy. In this paper we only consider the case where the source strategy is *full refresh*. [Lea86] also presented a method for refreshing a snapshot that minimizes the number of messages sent when refreshing a snapshot. The method requires annotating the base tables with two columns for a tuple address and a timestamp. We cannot adopt this method in data warehousing since the sources are autonomous.

Reference [CRGMW96] investigates algorithms to find differences in hierarchical structures (e.g., documents, CAD designs). Our focus here is on simpler, record structured differences, and on dealing with very large snapshots that may not fit in memory.

There has also been recent complementary work on copy detection of files and documents. Tools have been created to find similar files in a file system [MW94]. Copy detection mechanisms for documents have been proposed in an attempt to safeguard intellectual property on the Internet ([BDGM95], [SGM95]). These mechanisms ultimately provide as output the extent of the similarity of two files. The snapshot differential problem is concerned with detecting the specific differences of two files as opposed to measuring how different (or similar) two files are. Also related are [BGMF88] and [FWJ86], which propose methods for finding differing pages in files. However, these methods can only detect a few modifications and assume that no insertions or deletions have taken place.

The snapshot differential problem is also related to text comparison, for example, as implemented by UNIX *diff* and DOS *comp*. However, the text comparison problem is concerned with the order of the records. That is, it considers a *sequence* of records, while the snapshot differential problem is concerned with a *set* of records. Reference [HT77] outlines an algorithm that finds the longest common subsequence (LCS) of the lines of the text, which is used in the UNIX *diff*. Report [LGM95] takes a closer look at how this algorithm can be adopted to solve the snapshot differential problem, although the solution is not as efficient as the ones presented here.

The methods for solving the snapshot differential problem proposed here are based on ad hoc joins which have been well studied; [ME92] and [Sha86] are good surveys on join processing. The snapshot differential algorithms proposed here are used in the data warehousing system *WHIPS*. An overview of the system is presented in [HGMW+95]. After the modifications of multiple sources are detected, the modifications are integrated using methods discussed in [ZGMHW95].

Note that there are also cases wherein knowledge of the semantics of the information maintained at the warehouse helps make change detection simpler. For instance, if the warehouse keeps a history of all the information contained at the source, then it makes sense to simply pass complete snapshots to the warehouse. We have an outline of these special cases in report [LGM95].

3 Using Compression

In this section we first describe existing, ad hoc, join algorithms but we do not cover all the known variations and optimizations of these algorithms. We believe that many of these further optimizations can also be applied to the snapshot differential algorithms we present.

After extending the ad hoc algorithms to handle the differential problem, we study record compression techniques to optimize them. In the sections below, we denote the size of a file F as $|F|$ blocks and the size of main memory as $|M|$ blocks. We also exclude the cost of writing the output file in our cost analysis since it is the same for all of the algorithms.

3.1 Outer Join Algorithms

The basic sort merge join first sorts the two input files. It then scans the files once and any pair of records that satisfy the join condition are produced as output. The algorithm can be adapted to perform an outerjoin by identifying the records that do not join with any records in the other file during the scan. This can be done with no extra cost when two records are being matched: the record with the smaller key is guaranteed to have no matching records.

Since differentials are an on-going process running at a source, it is possible to save the sorted file of the previous snapshot. Thus, the algorithm only needs to sort the second file, F_2. This can be done using the multiway merge-sort algorithm. This algorithm constructs runs which are sequences of blocks with sorted records. After a series of passes, the file is partitioned into progressively longer runs. The algorithm terminates when there is only one run left. In general, it takes $2 * |F| * log_{|M|}|F|$ IO operations to sort a file with size $|F|$ ([Ull89]). However, if there is enough main memory ($|M| > \sqrt{|F|}$), the sorting can be done in $4 * |F|$ IO operations (sorting is done in two passes). The second phase of the algorithm, which involves scanning and merging the two sorted files, entails $|F_1| + |F_2|$ IO operations for a total of $|F_1| + 5 * |F_2|$ IO operations.

Algorithm 3.1
Input $F_{1\ sorted}$, F_2
Output F_{out} (the snapshot differential), $F_{2\ sorted}$
Method
(1) $F_{2\ runs} \leftarrow SortIntoRuns(F_2)$
(2) $r_1 \leftarrow$ read the next record from $F_{1\ sorted}$
(3) $r_2 \leftarrow$ read the next record from $F_{2\ runs}$; $F_{2\ sorted} \leftarrow Output(< r_2.K,\ r_2.B >)$
(4) **while** $((r_1 \neq NULL) \wedge (r_2 \neq NULL))$
(5) **if** $((r_1 = NULL) \vee (r_1.K > r_2.K))$ **then**
(6) $F_{out} \leftarrow Output(< Insert,\ r_2.K,\ r_2.B >)$
(7) $r_2 \leftarrow$ read the next record from $F_{2\ runs}$; $F_{2\ sorted} \leftarrow Output(< r_2.K,\ r_2.B >)$
(8) **else if** $((r_2 = NULL) \vee (r_1.K < r_2.K)$ **then**
(9) $F_{out} \leftarrow Output(< Delete,\ r_1.K >)$
(10) $r_1 \leftarrow$ read the next record from $F_{1\ sorted}$
(11) **else if** $(r_1.K = r_2.K)$ **then**
(12) **if** $(r_1.B \neq r_2.B)$ **then**
(13) $F_{out} \leftarrow Output(< Update,\ r_2.K,\ r_2.B >)$
(14) $r_1 \leftarrow$ read the next record from $F_{1\ sorted}$
(15) $r_2 \leftarrow$ read the next record from $F_{2\ runs}$; $F_{2\ sorted} \leftarrow Output(< r_2.K,\ r_2.B >)$
$\diamond$

Figure 1: Sort Merge Outerjoin as a Snapshot Differential Algorithm

The *IO* cost can be reduced further by just producing the sorted runs (denoted as $F_{2\ runs}$) in the first phase. This improved algorithm is shown in Figure 1. Step (1) produces the sorted F_2 runs, at a cost of only $2 * |F_2|$ *IO*s. (File F_1 has already been sorted at this point.) The sorted F_2 file, needed for the next run of the algorithm, can then be produced while matching $F_{2\ runs}$ with F_1. In producing the sorted F_2 file (steps 3, 7, 15), we read into memory one block from each run in $F_{2\ runs}$ (if the block is not already in memory), and select the record with the smallest K value. The merge process (steps 4 through 15) now costs $2 * |F_2| + |F_1|$ *IO*s. Thus, when sort merge join is used as a snapshot differential algorithm, the total cost incurred is $|F_1| + 4 * |F_2|$ *IOs.*

Another method that we discuss here is the partitioned hash join algorithm. In the partitioned hash join algorithm, the input files are partitioned into buckets by computing a hash function on the join attribute. Records are matched by considering each pair of corresponding buckets. First, one of the buckets is read into memory (the smaller one) and an in-memory hash table is built (assuming the bucket fits in memory). The second bucket is then read and a probe into the in-memory hash table is made for each record in an attempt to find a matching record in the first bucket. A more detailed discussion of the partitioned hash algorithm is found in Appendix A where we also show that the *IO* cost incurred is $|F_1| + 3 * |F_2|$.

3.2 Compression Techniques

Our compression algorithms reduce the sizes of records and the required *IO*. Compression can be performed in varying degrees. For instance, compression may be performed on the records of a file by compressing the whole record (possibly excluding the key field) into n bits. A block or a group of blocks can also be compressed into n bits. There are also numerous ways to perform compression such as computing the check sum of the data, hashing the data to obtain an integer or simply omitting fields in a record that are not important in the comparison process. Compression can also be *lossy* or *lossless*. In the latter case, the compression function guarantees that two different uncompressed values are mapped into different compressed values. Lossy compression functions do not have this guarantee but have the potential of achieving higher compression factors. Henceforth, we assume that we are using a lossy compression function. We ignore the details of the compression function and simply refer to it as $compress(x)$.

There are a number of benefits from processing compressed data. First of all, the compressed intermediate files, such as the buckets for the partitioned hash join, are smaller. Thus, there will be fewer *IO* when reading the intermediate files. Moreover, the compressed file may be small enough to fit in memory. Even if the compressed file does not fit entirely in memory, some of the join algorithms may still benefit. For example, the compressed file may result in buckets that fit in memory which improves the matching phase of the partitioned hash join algorithm. Another algorithm which may benefit from compression is hybrid hash join (discussed in Appendix A). For this algorithm, more in-memory buckets can be kept during the bucketizing phase if compression is used.

Compression is not without its disadvantages. As mentioned earlier, a lossy compression function may map two different records into the same compressed value. This means that the snapshot differential algorithm is probabilistic and may not be able to detect all the modifications to a snapshot. We now show that this can occur with a probability of 2^{-n}, where n is the number of bits

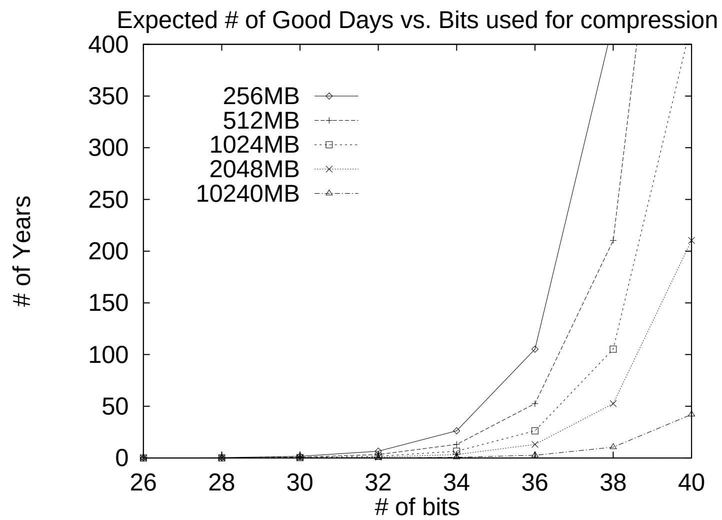

Figure 2: $N_{good\ days}$ for Different File Sizes

for the compressed value. Assume that we are compressing an object (which may be the B field, or the entire record, or an entire block, etc.) of b bits ($b > n$). There are then 2^b possible values for this object. Since there are only 2^n values that the compressed object can attain, there are $2^b/2^n$ original values mapped to each compressed value. Thus for each given original value, the probability that another value maps to the same compressed value is $((2^b/2^n) - 1)/2^b$, which is approximately 2^{-n} for large values of b. For sufficiently large values of n, this probability can be made very small. The expression 2^{-n}, henceforth denoted as E, gives the probability that a *single* comparison is erroneous. For example, if the B field of the record $< K, B >$ is compressed into a 32-bit integer, the probability that a single comparison (of two B fields) is erroneous is 2^{-32} or approximately $2.3 * 10^{-10}$. However, as we compare more records, the likelihood that a modification is missed increases. To put this probability of error into perspective, let us suppose we perform a differential on two 256 MB snapshots daily. We now proceed to compute how many days we expect to pass before a record modification is missed. We first compute the probability (denoted as p_{day}) that there is no error in comparing two given snapshots (that is, there is no error in one day). Let us suppose that the record size is 150 bytes which means that there are approximately 1,789,570 records for each file.

$$p_{day} = (1 - E)^{records(F)} = (1 - 2.3 * 10^{-10})^{1,789,570} = 0.99979169 \tag{1}$$

Using this probability, we can compute the expected number of days (denoted as $N_{good\ days}$) before an error occurs.

$$N_{good\ days} = (1 - p_{day}) * \sum_{1 \leq i} i * p_{day}^{i-1} = \frac{1}{1 - p_{day}} = 2,430\ days \tag{2}$$

This comes out to be 2,430 days, or more than 6.7 years! We believe that for some types of warehousing applications, such as data mining, this error rate will be acceptable.

It is evident from the equations above that as the number of records increases, the expected number of days before an error occurs goes down. This is shown more clearly in Figure 2. The graph shows that a 10 GB file will encounter more errors than a 256 MB file. However, as the number of bits used for compressing the B field is increased, the the expected number of years before an error occurs can be made comfortably large even for large files.

For the algorithms we will present here, we consider two ways of compressing the records. For both compression formats, we do not compress the key, and we denote the compressed B field as b. The first format is simply compress a record $< K, B >$ into $< K, b >$. For the second form, the only difference is that a pointer is appended forming the record $< K, b, p >$. The pointer p points to the corresponding disk resident uncompressed record. The use of the pointer will be explained when we describe the algorithms. We use u to represent the ratio of the size of the original record to that of the compressed record (including the key and pointer, if any). So, if an uncompressed file is size $|F|$, the compressed size will be $|F|/u$ blocks long.

3.3 Outerjoin Algorithms with Compression

We now augment the sort merge outerjoin with compression. We assume that the compressed sorted F_1 file was produced in the previous differential (denoted as $f_{1\ sorted}$, with a size of $|F_1|/u$). For this

Algorithm 3.2
Input $f_{1\ sorted}$, F_2
Output F_{out} (the snapshot differential), $f_{2sorted}$
Method
(1) $F_{2\ runs} \leftarrow SortIntoRuns(F_2)$
(2) $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$ (other r_1 reads later on are also from $f_{1\ sorted}$)
(3) $r_2 \leftarrow$ read the next record from $F_{2\ runs}$; $f_{2\ sorted} \leftarrow Output(< r_2.K,\ Compress(r_2.B) >)$
(4)-(6) See Algorithm 3.1
(7) $r_2 \leftarrow$ read the next record from $F_{2\ sorted}$; $f_{2\ sorted} \leftarrow Output(< r_2.K,\ Compress(r_2.B) >)$
(8)-(11) See Algorithm 3.1
(12) **if** $(r_1.b \neq Compress(r_2.B))$ **then**
(13)-(14) See Algorithm 3.1
(15) $r_2 \leftarrow$ read the next record from $F_{2\ sorted}$; $f_{2\ sorted} \leftarrow Output(< r_2.K,\ Compress(r_2.B) >)$
$\diamond$

Figure 3: Sort Merge Outerjoin Enhanced with the $< K,\ b >$ Compression Format

algorithm, we use the $< K,\ b >$ compression format. The modified sort merge algorithm is shown in Figure 3. Note that only the steps that differ from Algorithm 3.1 are shown explicitly. Steps (3), (7) and (15) now first compress the B field before producing an output into $f_{2\ sorted}$ (which is needed in the next differential). Also, when detecting the updates in step (12), the compressed versions of the B field are compared.

The sorting phase of the algorithm incurs $2 * |F_2|$ *IOs* (since it generates only the sorted runs as in Algorithm 3.1). The matching phase (steps (4) onwards) incurs $|F_2| + |f_1|$ *IOs* since the two files are scanned once. Lastly, the sorted $f_{2\ sorted}$ must be produced for the next differential, which costs $|f_2|$ *IOs*. The total cost is then $|f_1| + 3 * |F_2| + |f_2|$ *IOs*.

Greater improvements may be achieved by compressing not only the first snapshot but also the second snapshot before the files are matched. When the second snapshot arrives, it is read into memory and *compressed* sorted runs are written out. In essence, the uncompressed F_2 file is read only once. The problem introduced by compressing the second snapshot is that when insertions and updates are detected, the original uncompressed record must be obtained from F_2. In order to find the original (uncompressed) record, a pointer to the record must be saved in the compressed record. Thus, for this algorithm, the $< K, b, p >$ compression format must be used. The full algorithm is shown in Figure 4. Step (5a) (step(12a)) shows that when an insertion (update) is detected, the pointer p of the current record is used to obtain the original record in order to produce the correct output.

Step (1) of Algorithm 3.3 only incurs $|F_2| + |f_2|$ *IOs* instead of $2 * |F_2|$ *IOs*. Steps (4) through (15) incur $|f_1| + |f_2| + U + I$ *IOs*, where U and I are the number of updates and insertions found. An additional $|f_2|$ *IOs* are needed to write out the sorted f_2 file. As a result, the overall cost is $|f_1| + |F_2| + 3 * |f_2| + U + I$. The savings in *IO* cost is significant especially if there are few updates and inserts. Moreover, we are also assuming that each access using the pointer p requires a random *IO*. This can be optimized by recording all the pointers that need to be accessed. After the

Algorithm 3.3
Input $f_{1\ sorted}$, F_2
Output F_{out} (the snapshot differential), $f_{2\ sorted}$
Method
(1) $f_{2\ runs} \leftarrow SortIntoRuns \circ Compress(F_2)$
(2) $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$
(3) $r_2 \leftarrow$ read the next record from $f_{2\ runs}$; $f_{2\ sorted} \leftarrow Output(< r_2.K,\ r_2.b,\ r_2.p >)$
(4) **while** $((r_1 \neq NULL) \wedge (r_2 \neq NULL))$
(5) **if** $((r_1 = NULL) \vee (r_1.K > r_2.K))$ **then**
(5a) $r_{full} \leftarrow$ read tuple in F_2 with address $r_2.p$
(6) $F_{out} \leftarrow Output(< Insert,\ r_2.K,\ r_{full}.B >)$
(7) $r_2 \leftarrow$ read the next record from $f_{2\ runs}$; $f_{2\ sorted} \leftarrow Output(< r_2.K,\ r_2.b,\ r_2.p >)$
(8) **else if** $((r_2 = NULL) \vee (r_1.K < r_2.K)$ **then**
(9) $F_{out} \leftarrow Output(< Delete,\ r_1.K >)$
(10) $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$
(11) **else if** $(r_1.K = r_2.K)$ **then**
(12) **if** $(r_1.b \neq r_2.b)$ **then**
(12a) $r_{full} \leftarrow$ read tuple in F_2 with address $r_2.p$
(13) $F_{out} \leftarrow Output(< Update,\ r_2.K,\ r_{full}.B >)$
(14) $r_1 \leftarrow$ read the next record from $f_{1\ sorted}$
(15) $r_2 \leftarrow$ read the next record from $f_{2\ runs}$; $f_{2\ sorted} \leftarrow Output(< r_2.K,\ r_2.b,\ r_2.p >)$
$\diamond$

Figure 4: Sort Merge Outerjoin Enhanced with the $< K,\ b,\ p >$ Compression Format

differential is performed, these recorded pointers are used to produce the inserts and the updates. By sorting the pointers, the cost of probing the original snapshot is lessened since the *IO* operations are no longer random.

The partitioned hash outerjoin is augmented with compression in a very similar manner to the sort merge outerjoin. We assume that the compressed bucket files for the first snapshot (denoted collectively as f_1) were produced in the previous differential. We show in Appendix B that the overall cost is reduced to $|f_1| + 3 * |F_2| + |f_2|$ *IOs* if the buckets are compressed after the matching phase. If the buckets are compressed before the matching phase (and using the $< K, b, p >$ compression format), we also show in Appendix B that the overall cost is $|f_1| + |F_2| + 2 * |f_2| + I + U$ *IOs*.

The performance gains can even be greater if the compression factor u is high enough such that all of the buckets of F_1 fit in memory. In this case, all the buckets for F_1 are simply read into memory ($|f_1|$ *IOs*). The file F_2 is then scanned, and for each record in F_2 read, the in-memory buckets are probed. The compressed buckets for F_2 can also be constructed for the next differential during this probe. The overall cost of this algorithm is only $|f_1| + |F_2| + |f_2|$ *IOs*. Note that the cost is independent of the number of updates and inserts unlike the algorithm discussed previously. Unfortunately, this optimization cannot be used for the sort merge outerjoin because constructing the compressed sorted file for F_2 cannot be done by just scanning through F_2 once.

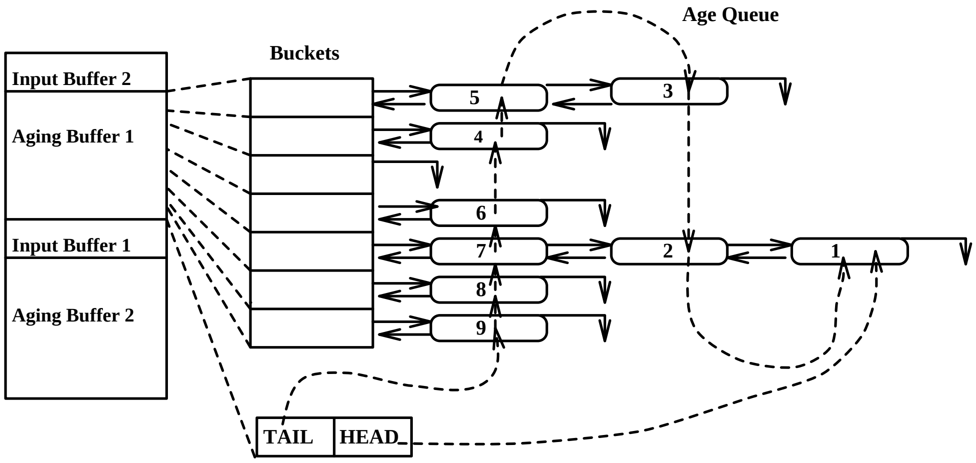

Figure 5: The *window* Algorithm Data Structures

4 The Window Algorithm

In the previous section, we described algorithms that compute the differential of two snapshots based on ad hoc join algorithms. We saw that the snapshots are read multiple times. Since the files are large, reading the snapshots multiple times can be costly. We now present an algorithm that reads the snapshots exactly *once*. This new algorithm assumes that matching records are physically "nearby" in the files. As mentioned in Section 1, matching records cannot be expected to be in the same position in the two snapshots, due to possible reorganizations at the source. However, we may still expect a record to remain in a relatively small area, such as a block, cylinder, or track. This is because file reorganization algorithms typically rearrange records within a physical sub-unit. The *window* algorithm takes advantage of this, and of ever increasing main memory capacity, by maintaining a moving window of records in memory for each snapshot. Only the records within the window are compared in the hope that the matching records occur within the window. Unmatched records are reported as either an insert or a delete, which can lead to useless delete-insert pairs. As discussed in Section 1, a small number of these may be tolerable.

For the window algorithm, we divide available memory into four distinct parts as shown in Figure 5. Each snapshot has its own *input buffer* (*input buffer 1* is for F_1) and *aging buffer*. The input buffer is simply the buffer used in transferring blocks from disk. The aging buffer is essentially the moving window mentioned above.

The algorithm is shown in Figure 6 and we now proceed to explain each step. Steps (1) and (2) simply reads a constant number of input block of records from file F_1 and file F_2 to fill *input buffer 1* and *input buffer 2*, respectively. This process will be done repeatedly by steps (9) and (10). Before the input buffers are refilled, the algorithm guarantees that they are empty. Steps

Algorithm 4.1
Input F_1, F_2, n (number of blocks in the input buffer)
Output F_{out} (the snapshot differential)
Method
(1) $Input\ Buffer_1 \leftarrow$ Read n blocks from F_1
(2) $Input\ Buffer_2 \leftarrow$ Read n blocks from F_2
(3) **while** $((Input\ Buffer_1 \neq EMPTY) \wedge (Input\ Buffer_2 \neq EMPTY))$
(4) Match $Input\ Buffer_1$ against $Input\ Buffer_1$
(5) Match $Input\ Buffer_1$ against $Aging\ Buffer_2$
(6) Match $Input\ Buffer_2$ against $Aging\ Buffer_1$
(7) Insert contents of $Input\ Buffer_1$ into $Aging\ Buffer_1$
(8) Insert contents of $Input\ Buffer_2$ into $Aging\ Buffer_2$
(9) $Input\ Buffer_1 \leftarrow$ Read n blocks from F_1
(10) $Input\ Buffer_2 \leftarrow$ Read n blocks from F_2
(11) Report records in $Input\ Buffer_1$ as deletes
(12) Report records in $Input\ Buffer_2$ as inserts
$\diamond$

Figure 6: Window Algorithm

(4) through (6) are concerned with matching the records of the two snapshots. In Step (4), the matching is performed in a nested loop fashion. This is not expensive since the input buffers are relatively small. The matched records can produce updates if the B fields differ. The slots that these matching records occupy in the buffer are also marked as free. In step (5), the remaining records in *input buffer 1* are matched against *aging buffer 2*. Since the aging buffers are much larger, the aging buffers are actually hash tables to make the matching more efficient. For each remaining record in *input buffer 1*, the hash table that is *aging buffer 2* is probed for a match. As in step (4), an update may be produced by this matching. The slots of the matching records are also marked as free. Step (6) is analogous to step (5) but this time matching *input buffer 2* and *aging buffer 1*. Steps (7) and (8) clear both input buffers by forcing the unmatched records in the input buffers into their respective aging buffers. The same hash function used in steps (4) and (5) is used to determine which bucket the record is placed into. Since new records are forced into the aging buffer, some of the old records in the aging buffer may be displaced. These displaced records constitute the deletes (inserts) if the records are displaced from *input buffer 1* (*input buffer 2*). The displacement of old records is explained further below. The steps are then repeated until both snapshots are processed. At that point, any remaining records in the aging buffers are output as inserts or deletes.

In the hash table that constitutes the aging buffer there is an embedded "aging" queue, with the head of the queue being the oldest record in the buffer, and the tail being the youngest. Figure 5 illustrates the aging buffer. Each entry in the hash table has a timestamp associated with it for illustration purposes only. The figure shows that the oldest record (with the smallest timestamp) is at the head of the queue. Whenever new records are forced into the aging buffer, the new records are placed at the tail of the queue. If the aging buffer is full, the record at the head of the queue is displaced as a new record is enqueued at the tail. This action produces a delete (insert) if the buffer in question is *aging buffer 1* (*aging buffer 2*).

Since files are read once, the *IO* cost for the *window* algorithm is only $|F_1| + |F_2|$ *regardless* of memory size, snapshot size and number of updates and inserts. Thus the window algorithm achieves the *optimal IO* performance if compression is not considered. However, the *window* algorithm can produce useless delete-insert pairs in Steps 6 and 7 of the algorithm. Intuitively, the number of useless delete-insert pairs produced depends on how physically different the two snapshots are.

To quantify this difference, we define the concept of the *distance* of two snapshots. We want the distance measure to be symmetric and independent of the size of the file. While the reason for the first property is obvious, the reason for the second is more subtle. If the measure is not independent of the size of the file, we may end up with a measure that is unbounded. For instance, if the distance of two snapshots is defined to be the sum of the absolute value of the differences in positions of matching records, this sum may become arbitrarily large for large snapshots. Moreover, such a measure can be misleading since two small snapshots that are in opposite order will have a small distance measure when intuitively they should have a large distance.

The equation below exhibits the two desired properties.

$$distance = \frac{\sum_{R_1 \epsilon F_1, R_2 \epsilon F_2, match(R_1,R_2)} |pos(R_1) - pos(R_2)|}{max(records(F_1), records(F_2))^2/2} \tag{3}$$

The function *pos* returns the physical position of a record in a snapshot. The boolean function *match* is true when records R_1 and R_2 have matching keys. The function *records* returns the number of records of a snapshot file. F represents the larger of the two files. Thus, this equation sums up the absolute value of the difference in position of the matching records and normalizes it by the maximum distance for the given snapshot file sizes. The maximum distance between two snapshots is attained when the records in the second snapshot are in the opposite order (the first record is exchanged with the last record, the second record with the second to the last, and so on) relative to the first snapshot. If $records(F_1) = records(F_2)$, it is easy to see that in the worst case the average displacement of each record is $records(F)/2$, and hence the maximum distance is $records(F) * records(F)/2$. If the files are of different sizes, using the larger of the two files gives an upper bound on the maximum distance. Our *distance* metric will be used in the following section to evaluate the window algorithm.

5 Performance Evaluation

5.1 Analytical IO Comparison

We have outlined in the previous section algorithms that can compute a snapshot differential: performing sort merge outerjoin (SM), performing a partitioned hash outerjoin (PH), performing a sort merge outerjoin with two kinds of record compression ($SMC1$, $SMC2$), performing partitioned hash outerjoin with two kinds of record compression ($PHC1$, $PHC2$) and using the *window* algorithm (W). $SMC1$ denotes sort merge outerjoin with a record compression format of

	Variable Description	Default Values
M	Memory Size	32 MB
B	Block Size	16K
F	File Size	256 MB or 1024 MB
R	Record Size	150 bytes
$records(F)$	Number of Rows	1,789,569 or 7,158,279
r	Compressed Record Size	10 or 14 bytes
u	Compression Factor	15 or 10
$U+I$	Number of Inserts and Updates	1% of $records(F)$
IO	Number of IOs	N/A
X	Intermediate File Size	N/A
E	Probability of Error	N/A

Figure 7: List of Variables

Algorithm	IO_{256} (%savings)	IO_{1024} (%savings)	X_{256} (MB)	X_{1024} (MB)	Probability of Error (E)
SM	81,920	327,680	16384	65,536	0
$SMC1$	51,336 (37%)	205,346 (37%)	16,384	65,536	$2.3 * 10^{-10}$
$SMC2$	40,833 (50%)	163,333 (50%)	1,639	6,554	$2.3 * 10^{-10}$
PH	65,536 (20%)	262,144 (20%)	16,384	65,536	0
$PHC1$	18,568 (77%)	205,346 (37%)	16,384	65,536	$2.3 * 10^{-10}$
$PHC2$	19,660 (76%)	156,779 (52%)	1,639	6,554	$2.3 * 10^{-10}$
W	32,768 (60%)	131,072 (60%)	0	0	0

Figure 8: Comparison of Algorithms

$< K, b >$ (similarly for $PHC1$); $SMC2$ uses the record compression format $< K, b, p >$ (similarly for $PHC2$). In this section, we will illustrate and compare the algorithms in terms of IO cost, size of intermediate files, and the probability of error. Due to space limitations, this is not a comprehensive study, but simply an illustration of potential differences between the algorithms in a few realistic scenarios.

Figure 7 shows the variables that will be used in comparing the algorithms. We assume that the snapshots have the same number of records. The number of records ($records(F)$) are calculated using F/R, where R is the record size (150 bytes). The compressed record size is 10 bytes for the $< K, b >$ format and 14 bytes for the $< K, b, p >$ format. This leads to compression factors of 15 and 10 respectively.

Figure 8 shows a summary of the results computed for the various algorithms. The two columns

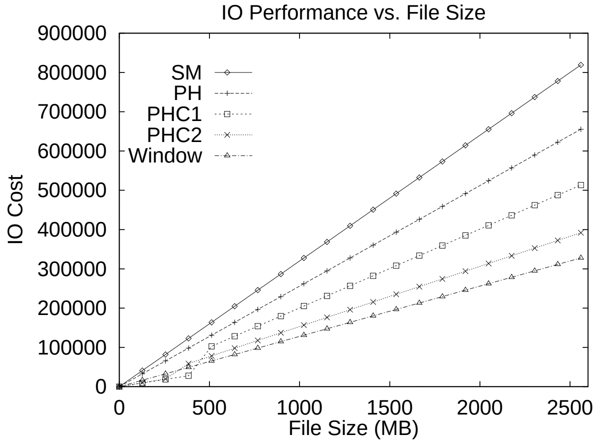

Figure 9: IO Cost Comparison of Algorithms

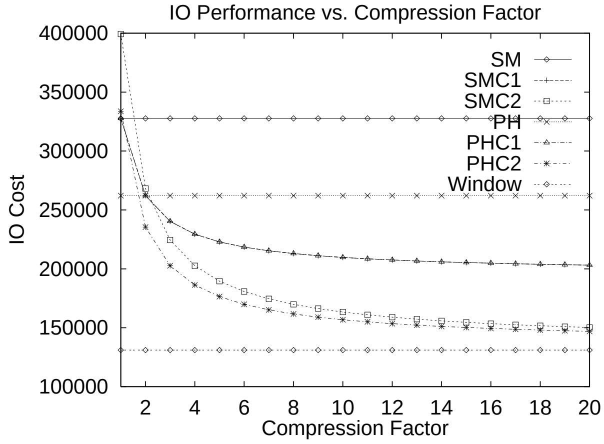

Figure 10: IO Cost and Compression Factor

labeled IO_{256} and IO_{1024} show the IO cost incurred in processing 256 MB and 1024 MB snapshots for the different algorithms. Using the sort merge outerjoin as a baseline, we can see that the partitioned hash outerjoin (PH) reduces the IO cost by 20%. Compression using the $< K, b >$ record format achieves a 37% reduction in IO cost over sort merge using $SMC1$, and a 50% reduction using $SMC2$. For the 256 MB file, the compressed file fits in memory which enables the $PHC1$ and $PHC2$ algorithms to build a complete in-memory hash table, as explained in Section 3.3. The reduction in IO cost for these two algorithms, in this case, surpasses even that of the window algorithm.

However, when the larger file is considered, the compressed file no longer fits in the 32 MB memory. Thus the $PHC1$ and $PHC2$ algorithms achieve more modest reductions in this case (37% and 52% respectively). Other than these two algorithms, the reductions achieved by the other algorithms are unchanged even with the larger file.

Figure 9 shows how the algorithms compare when the size of the snapshots is varied over a range. The values of other parameters are unchanged. Note that we have not plotted $SMC1$ and $SMC2$ since their plots are almost indistinguishable from $PHC1$ and $PHC2$ respectively beyond a file size of 500 MB. Also note the discontinuity in the graph for $PHC1$ and $PHC2$. $PHC1$ is able to build an in-memory hash table if the file is smaller than 500 MB (and files smaller than 320 MB for $PHC2$). If the partitioned hash join algorithms are able to build an in-memory hash table, they can even outperform the window algorithm.

Clearly, the IO savings for compression algorithms depend on the compression factor. Figure 10 illustrates that when the compression factor is low, the algorithms with compression perform worse than PH (even worse than SM in case of $SMC1$ and $SMC2$). The other point that this graph illustrates is that the benefits of compression are bounded (which is to be expected from the IO cost equations). Thus, going beyond a factor of 10 in this case does not buy us much.

The performance of the compression algorithms that use the pointer format (algorithms $PHC2$ and $SMC2$) depend on the number of updates and inserts. If $U + I$ is higher than what we have

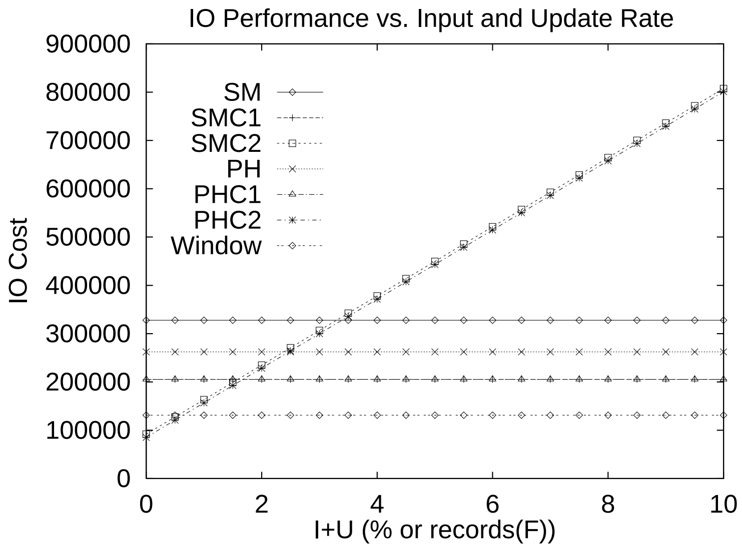

Figure 11: IO Cost and Varying Update and Insertion Rates

assumed, $PHC1$ and $SMC1$ outperform $PHC2$ and $SMC2$. Figure 11 shows the performance of the algorithms with different $U+I$. This shows that $PHC2$ and $SMC2$ are only useful for scenarios with relatively few modifications between snapshots (less than say 2 percent of the records). By manipulating the IO cost equations, it is not hard to show that if $U+I$ is greater than 1.7%, $PHC1$ and $SMC1$ incur less IO than $PHC2$ and $SMC2$.

The next two columns in Figure 8 (X_{10} and X_{25}) examine the size of the intermediate files. In the case of the SM, PH, $SMC1$ and $PHC1$ algorithms, uncompressed intermediate files need to be saved. In the case of the $SMC2$ and $PHC2$ algorithms, the compressed versions of these files are constructed, which leads to a more economic disk usage. The *window* algorithm, on the other hand, does not construct any intermediate files.

The last column (labeled E) illustrates the probability of a missed matching record pair. Note that both record compression formats result in the same probability of error although the two formats have different compression factors. This is because the B field is compressed into a 32 bit integer for both formats.

In closing this section, we stress that the numbers we have shown are only illustrative. The gains of the various algorithms can vary widely. For example, if we assume very large records, then even modest compression can yield huge improvements. On the other hand, if we assume very large memories (relative to the file sizes), then the gains become negligible.

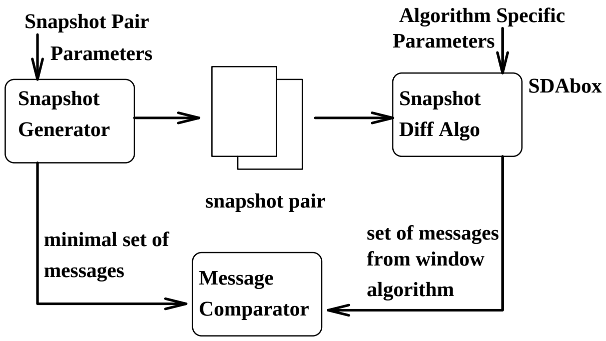

Figure 12: The Evaluation System

	Snapshot Parameters	Default Values
R	Size of B field Size of Record	150 bytes 156 bytes
F	Number of Records File Size	650,000 100 MB
	$displacement_{avg}$	50,000 records
U	Number of Updates	20% of $records(F)$
	Window Parameters	Default Values
AB	Aging Buffer Size	8 MB
IB	Input Block Size	16K

Figure 13: List of Parameters

5.2 Evaluation of Implemented Algorithms

In *WHIPS*, we have implemented the sort merge outerjoin and the *window* algorithm to compute the snapshot differentials. We have also built a snapshot differential algorithm evaluation system, which we used to study the effects of the snapshot pair distance on the number of useless delete-insert pairs that is produced by the *window* algorithm. We will also use the evaluation system to compare the actual running times of the *window* algorithm and the sort merge outerjoin algorithm. The evaluation system is depicted in Figure 12.

The snapshot generator produces a pair of synthetic snapshots with records of the form $< K, B >$. The snapshot generator produces the two snapshots based on the following parameters: size of the B field, number of records, average record displacement ($displacement_{avg}$) and percentage of updates. The first snapshot is constructed to have ordered K fields with the specified number of records and with the specified B field size. Figure 13 shows the default snapshot pair parameters.

Conceptually, the second snapshot is produced by first copying the first snapshot. Each record R_j in the second snapshot is then swapped with a record that is, on average (uniformly distributed from 0 to $2 * displacement_{avg}$), $displacement_{avg}$ records away from R_j. Based on the specified percentage of updates, some of the records in the second snapshot are modified to simulate updates. Insertions and deletions are not generated since they do not affect the number of useless delete-insert pairs produced. Notice that $displacement_{avg}$ is not the distance measure between snapshots. It is a generator parameter that indirectly affects the resulting distance. Thus, after generating the two snapshots, the actual distance of the two snapshots is then measured.

The two snapshots are then passed to the snapshot differential algorithm (in the SDA_{BOX}) being tested. Note that any of the previous algorithms discussed can be plugged into the SDA_{BOX}. In the experiments that we present here we focus on the *window* and the sort merge outerjoin algorithms. Algorithm specific parameters are also passed into the SDA_{BOX}. By varying the aging buffer size and the input buffer size parameters passed into the SDA_{BOX}, we can study how these parameters affect the *window* algorithm. Figure 13 also shows the default *window* parameters. These were used unless the parameter was varied in an experiment.

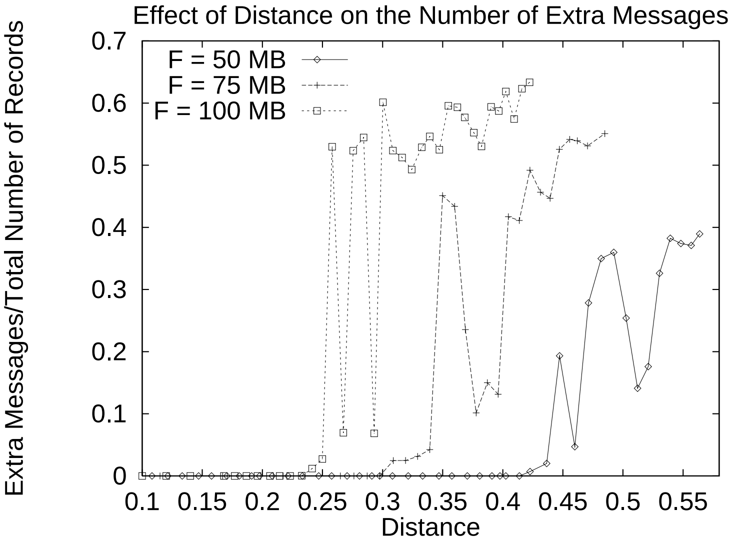

Figure 14: Effect of Distance on the Number of Extra Messages

After the snapshot differential algorithm is run, the output of the algorithm is compared to what was "produced" by the snapshot generator. Since the snapshot generator synthesized the two snapshots, it also knows the minimal set of differences of the two snapshots (which is the set of records of the first snapshot that it modified to produce the second). The message comparator can then check for the correctness of the output and count the number of extra messages.

The experiments we conducted enable us to evaluate, given the size of the aging buffer, and the size and the distance of the snapshots, how well the *window* algorithm will perform in terms of the number of extra messages produced (see Section 1.1). In the first experiment, we varied the $displacement_{avg}$ (and indirectly the distance) and measured the number of extra messages produced. This experiment was performed on three pairs of snapshots whose sizes ranged from 50 MB to 100 MB. Figure 14 shows that, as expected, as the distance of the snapshots increases beyond the capacity of the aging buffer, the number of extra messages increases. As the number of extra messages sharply rises, the graphs exhibit strong fluctuations. This is because the synthetic snapshots were produced randomly and only one experiment was done for each distance. (Only one experiment was done for each distance since it is hard to create two or more synthetic snapshot pairs with exactly the same distance.) For each snapshot size, there is a critical distance ($distance_{crit}$) which causes the *window* algorithm to start producing extra messages with the given aging buffer size.

For a system designer, it is helpful to translate $distance_{crit}$ into a critical average *physical* displacement. For instance, if the designer knows that records can only be displaced within a cylinder and the designer can only allocate 8 MB to each aging buffer, it is useful to know if the *window* algorithm produces few useless delete-insert messages in this scenario. We now capture this notion by first manipulating the definition of distance (equation (3) in Section 4) to show

File Size	$records(F)$	$distance_{crit}$	$displacement_{crit\ MB}$
50 MB	162,500	0.44	5.11
75 MB	325,000	0.34	7.91
100 MB	650,000	0.24	11.2

Figure 15: $distance_{crit}$ and $displacement_{crit\ MB}$

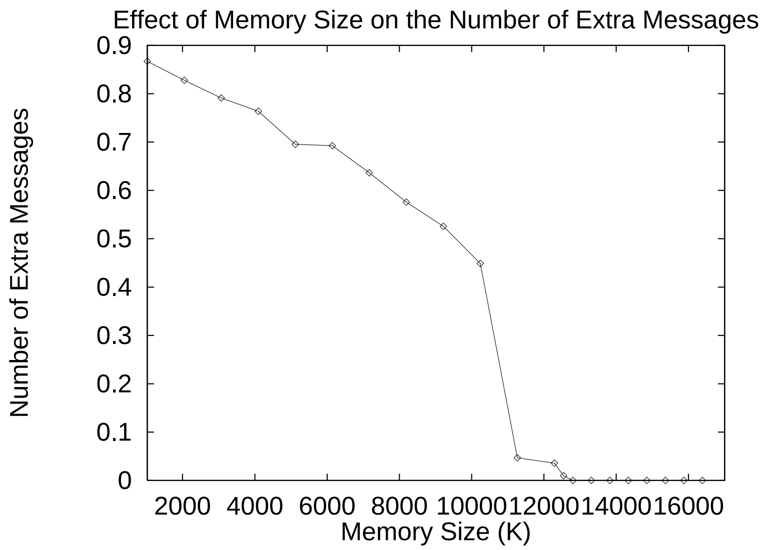

Figure 16: Effect of the Memory Size on the Number of Extra Messages

that $distance_{crit}$ of the different snapshot pairs can be translated into a critical average physical displacement (in terms of MB). Since there are no insertions nor deletions in the synthetic snapshot pair, we can define a critical average *record* displacement (denoted as $displacement_{crit}$) which is related to $distance_{crit}$ as shown in equation (4).

$$distance_{crit} = \frac{\sum_{R_1 \epsilon F_1, R_2 \epsilon F_2, match(R_1,R_2)} |pos(R_1) - pos(R_2)|}{records(F)^2/2} = \frac{records(F) * displacement_{crit}}{records(F) * records(F)/2} \quad (4)$$

$$displacement_{crit\ MB} = displacement_{crit} * R = distance_{crit} * (records(F)/2) * R \quad (5)$$

Using the size of the record (R), we can translate the $distance_{crit}$ into a critical average physical displacement (denoted as $displacement_{crit\ MB}$ which is in terms of MB) using equation (5). Figure 15 shows the result of the calculations for the different snapshot pairs. The $distance_{crit}$ of the snapshot pairs are estimated from Figure 14. This table shows, for example, that the *window* algorithm can tolerate an average physical displacement of about 11.2 MB given an aging buffer

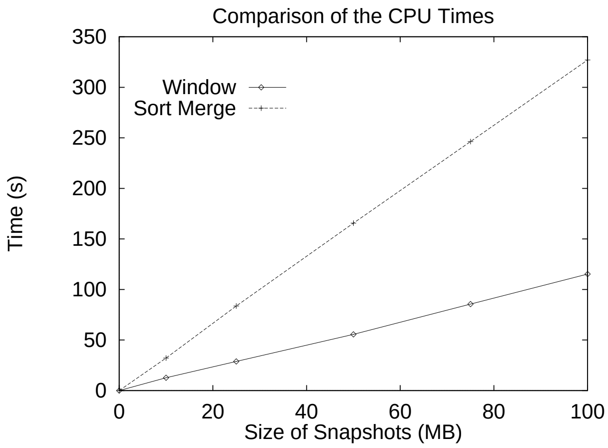

Figure 17: Comparison of the CPU Times

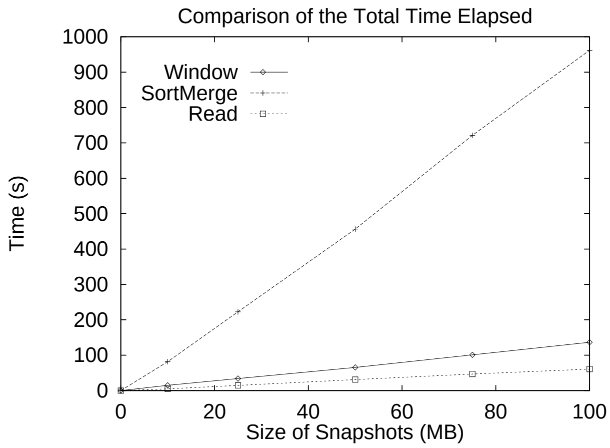

Figure 18: Comparison of the Total Times

size of only 8 MB to compare 100 MB snapshots. Thus, if a system designer knows that the records can only be displaced within, say a page (which is normally smaller than 11.2 MB), then the designer can be assured that the *window* algorithm will not produce excessive amounts of extra messages.

In the next experiment, we focus on the 100 MB snapshots. Using the parameters listed in Figure 13, we varied the size of the aging buffer from 1.0 MB to 16 MB. The $displacement_{avg}$ was set at 50,000 with a resulting distance of 0.34, which is well above the $distance_{crit}$. Figure 16 shows that once the size of the aging buffer is at least 12.8 MB, no extra messages are produced. This is to be expected since we showed previously (Figure 15) that the tolerable $displacement_{crit\ MB}$ for the 100 MB file is 11.2 MB. Using the same snapshot pair, we also varied the input block size from 8 K to 80 K. The variation had no effect on the number of extra messages and we do not show the graph here. Again, this is to be expected, since the size of the aging buffer is much larger than the size of the input block. Thus, even if the input block size is varied, the window size stays the same. We also varied the record size (keeping the size of the snapshot constant) and this showed no effect on the number of extra messages produced.

Lastly we compared the CPU time and the clock time (which includes the *IO* time) that the *window* algorithm consumes to that of the sort merge outerjoin based algorithm. We ran the simulations on a DEC Alpha 3000/400 workstation running UNIX. We used the UNIX *sort* utility in the implementation of the sort merge outerjoin. (UNIX *sort* may not be the most efficient, but we believe it is adequate for the comparisons we wish to perform here.) We used the same input block size for both the *window* and the sort merge outerjoin algorithms (16 K). The $displacement_{avg}$ of the two snapshots was set so that the resulting distance was 0.05 (within the $distance_{crit}$ for all file sizes). The analysis in the previous section illustrated that the *window* algorithm incurs fewer *IO* operations than the sort merge outerjoin algorithm. Figure 17 shows that the *window* algorithm is also significantly less CPU intensive than the sort merge based algorithm. As expected then,

Figure 18 shows that the *window* algorithm outperforms the sort merge outerjoin in terms of clock time. Moreover, Figure 18 also shows that the CPU time is a small fraction of the clock time in the *window* algorithm. Thus, the *IO* comparisons of Section 5.1 are indeed useful.

6 Conclusion

We have defined the snapshot differential problem and discussed its importance in data warehousing. The algorithms we have proposed are "extensions" of traditional join algorithms, but take advantage of the flexibility allowed for snapshot differentials. All of our proposed algorithms are relatively simple, but we view this as essential for dealing efficiently with large files. In summary, we have the following results:

- By augmenting the outerjoin algorithms with record compression, we have shown that very significant savings in *IO* cost can be attained. We have also illustrated that the probability that an error will occur if compression is used can be made negligible.
- We have introduced the *window* algorithm which works extremely well if the snapshots are not too different. Under this scenario, this algorithm outperforms the join based algorithms and its running time is comparable to simply reading the snapshots once. We have defined the concept of snapshot pair distance to characterize quantitatively the scenarios where the algorithm is applicable. We have also defined $displacement_{crit\ MB}$ which can be of use to the system designer.

We have incorporated the *window* and the sort merge outerjoin algorithms into the initial *WHIPS* Warehouse prototype at Stanford. The production version of the algorithm takes as input a "format definition" that describes the record format of the snapshots and identifies the key field(s). The format allows for complex value fields (e.g., lists), but the *window* algorithm will consider the entire record as a single field. We also plan to implement a post-processor that filters out useless delete-insert pairs before they are sent to the warehouse. (If the number of output records is small, this filtering could be done very efficiently in memory.) The differential algorithm and the warehouse itself are implemented within the Corba distributed object framework, using ILU, an implementation from Xerox PARC [CJS+94]. Thus, the modifications will be sent to the warehouse by remote procedure calls on its "insert record," "delete record," and "update record" methods. For our system demonstrations, we use the *window* algorithm to extract modifications from a legacy source that handles financial account information at Stanford. In the future, we will use the algorithm to compare file dumps of company information obtained from various Dialog databases ([Ser94]).

References

[AL80] M.E. Adiba and B.G Lindsay. Database snapshots. In *Proceedings of the International Conference on Very Large Databases*, Montreal, Canada, October 1980.

[BDGM95] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the ACM SIGMOD Annual Conference*, San Francisco, CA, May 1995.

[BGMF88] D. Barbara, H. Garcia-Molina, and B. Feijoo. Exploiting symmetries for low-cost comparison of file copies. In *Proceedings of the International Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[CJS+94] A. Courtney, W. Janssen, D. Severson, M. Spreitzer, and F. Wymore. Inter-language unification, release 1.5. Technical Report ISTL-CSA-94-01-01, Xerox PARC, May 1994.

[CRGMW96] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD Annual Conference*, Montreal, Canada, June 1996.

[FWJ86] W.K. Fuchs, K. Wu, and Abraham J. Low-cost comparison and diagnosis of large remotely located files. In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, January 1986.

[Gol95] Rob Goldring. Ibm datapropagator relational application guide. *IBM White Paper*, 1(1), 1995.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[HC94] L. Haas and M. Carey. SEEKing the truth about ad hoc join costs. Technical report, IBM Almaden Rsearch Center, 1994.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin*, June 1995.

[HT77] J.W. Hunt and Szymanski T.G. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5), 1977.

[IC94] W.H. Inmon and E. Conklin. Loading data into the warehouse. *Tech Topic*, 1(11), 1994.

[KR87] B. Kahler and O. Risnes. Extending logging for database snapshot refresh. In *Proceedings of the International Conference on Very Large Databases*, Brighton, England, September 1987.

[Lea86] B.G. Lindsay and et al. A snapshot differential refresh algorithm. In *Proceedings of the ACM SIGMOD Annual Conference*, Washington DC, May 1986.

[LGM95] W.J. Labio and H. Garcia-Molina. Comparing very large database snapshots. Technical Report STAN-CS-TN-95-27, Computer Science Department, Stanford University, June 1995.

[Loh85] G.M Lohman. Query processing in R*. In *Query Processing in Database Systems*, Berlin, West Germany, March 1985.

[ME92] P. Mishra and M. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1), 1992.

[MW94] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the winter USENIX Conference*, January 1994.

[Ser94] Dialog Information Services. Dialog pocket guide 1994. *Dialog Information Services*, 1(1), 1994.

[SGM95] N. Shivakumar and H. Garcia-Molina. Scam: A copy detection mechanism for digital documents. In *Proceedings of the 2nd International Conference in Theory and Practice of Digital Libraries*, Austin, Texas, June 1995.

[Sha86] L. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3), 1986.

[Squ95] C. Squire. Data extraction and transformation for the data warehouse. In *Proceedings of the ACM SIGMOD Annual Conference*, San Francisco, CA, May 1995.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, Rockville, MD, 1989.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD Annual Conference*, San Francisco, CA, May 1995.

A Analysis of the Partitioned Hash Join Algorithm

In this section, we obtain the *IO* cost formula for the partitioned hash algorithm. In the partitioned hash join algorithm, the input files are partitioned into buckets by computing a hash function on the join attribute. The matching phase is performed by considering each pair of corresponding buckets. The smaller bucket is read into memory and an in-memory hash table is built. The second bucket is then read and a probe into the in-memory hash table is made for each record in an attempt to find a matching record in the first bucket. Matching tuples are merged and produced as output. Creating the buckets incurs $2 * |F_1| + 2 * |F_2|$ *IOs* and the matching phase and merging phase incur $|F_1| + |F_2|$ *IOs*, assuming the buckets fit in memory. This assumption has a main memory requirement of $|M| > \sqrt{min(|F_1|, |F_2|)}$. If the buckets do not fit in memory, additional repartitioning needs to be done. In general the *IO* cost is $2 * log_N(|F_1|/|M|) * (|F_1| + |F_2|)$ with repartitioning (where N is the number of buckets) [Gra93]. For the rest of the analysis, we assume that the buckets do fit in memory. In a similar manner to the sort merge join algorithm, the buckets of the later snapshot can be saved for the next snapshot differential process. Thus the total *IO* cost incurred is $|F_1| + 3 * |F_2|$ since only the second snapshot needs to be partitioned into buckets.

It may also be the case that there is enough memory to keep some of the hash buckets from being written to disk when creating the buckets. This modified partitioned hash join is called the hybrid hash join. We do not consider the hybrid hash join in this paper. [LGM95] also extends the hybrid hash join to perform an outerjoin.

Unlike the sort merge algorithm, the partitioned hash join algorithm does not detect the deletions and insertions without additional data structures. For each pair of buckets, two arrays of flags must be allocated to keep track of the records that have not been matched. Let us assume that an in-memory hash table is built for a bucket of file F_1 (denoted as B_{F_1}), and a portion b_{F_2} of the corresponding F_2 bucket is read from disk. We define an array of flags for B_{F_1} and another array for b_{F_2} which indicate whether a record has been matched or not. All of the flags are initialized to indicate that all the records are unmatched. Whenever two matching (same K field) records are found, the corresponding flags are marked as matched. After b_{F_2} is processed, the unmatched records in b_{F_2}, as identified by the flags, are identified as inserted records. The next b_{F_2} portion is then read from disk and the flag array for b_{F_2} is reinitialized. After processing the last b_{F_2} portion for that F_2 bucket, the array of flags for B_{F_1} is consulted to identify the unmatched records that constitute deleted records. It is easy to see that the *IO* cost of the partitioned hash join algorithm is not altered with this modification (given that the array of flags fit in memory).

B Augmenting the Partitioned Hash Join with Compression

In this section, we augment the partitioned hash algorithms with compression. We assume that the compressed bucket files for the first snapshot was produced in the previous snapshot differential. When the second snapshot arrives, the buckets are created as in the previous section, incurring $2 * |F_2|$ *IOs*. The corresponding buckets are matched by reading the smaller bucket (which is most likely a bucket in f_1) into main memory. An in-memory hash table is constructed and the algorithm proceeds in a similar fashion to the partitioned hash outerjoin explained in Appendix A. The only difference is that the compressed B fields are compared when searching for an update. In addition, the records in b_{F_2} are compressed and written into a bucket file. After processing all of the F_2 buckets, the set of compressed buckets that comprise f_2 is also complete and ready for the next snapshot differential. The matching phase incurs $|f_1| + |F_2|$ *IOs* to read in the buckets and $|f_2|$ to write out the buckets for the next snapshot differential. Therefore, the overall cost is $|f_1| + 3 * |F_2| + |f_2|$ *IOs*.

Like the sort merge outerjoin, greater performance gains can be made by compressing the buckets of F_2 before the matching phase. Similarly, the $< K, b, p >$ compression format is used. In this case, only $|F_2| + |f_2|$ *IO* operations are needed to bucketize F_2 into a set of compressed buckets denoted as f_2. The matching phase is similar except that pointers must be followed to find the inserted and updated records. As a result, the overall *IO* cost is $|f_1| + |F_2| + 2 * |f_2| + I + U$. As in the sort merge outerjoin, we can also argue that the probes on F_2 through p can be recorded and can be done more efficiently after processing f_2.