Comparing Very Large Database Snapshots

Wilburt Juan Labio, Hector Garcia-Molina
Department of Computer Science
Stanford University, Stanford CA 94305
{*wilburt, hector*}*@cs.stanford.edu*

October 31, 1995

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. We call this problem the *snapshot differential problem*. We show that this is closely related to outerjoins. In this paper we extend the traditional join algorithms to perform outerjoins. We then make the outerjoin algorithms more efficient by using compression techniques. We also examine how text comparison algorithms can be used to solve the *snapshot differential problem*.

1 Introduction

Warehousing is a technique for retrieval and integration of data from distributed, independent and possibly heterogeneous information sources. A data warehouse is a repository of integrated information that is available for queries. As relevant information is available from a source or as relevant information is modified, the new information is extracted from the source, translated to the data model, and integrated with the existing data of the data warehouse.

The information sources may range from object oriented databases, relational databases to legacy systems. Even with the advent of the more popular relational database systems, there is still a proliferation of legacy systems. Consequently, legacy systems are still very important. One of the complications in integrating legacy systems into a data warehouse is detecting the modifications made to the system. Legacy systems do not have the capability of continuously monitoring the data and performing actions when certain modifications are made (unlike say Sybase triggers). The legacy system data are usually exported by taking snapshots of the data periodically to be used as information sources of the data warehouse. The problem of detecting the modifications made to the legacy system data is reduced to finding the difference between two snapshot files. We call this the *snapshot differential* problem. The *snapshot differential* problem also arises when there are relational or object-oriented databases that have restricted access. For instance, GenBank is a genome database that may be queried using only one type of form submitted through the

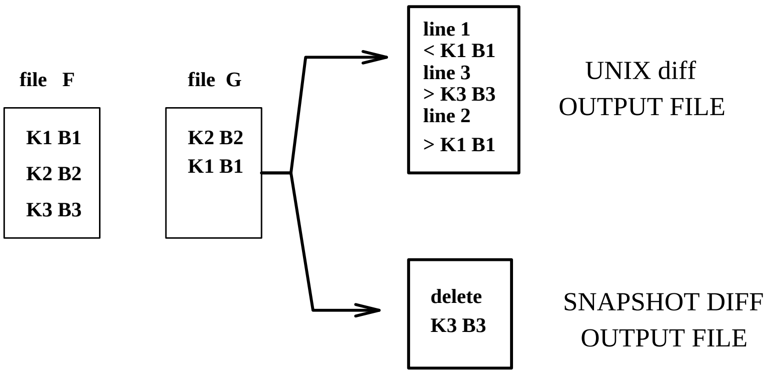

Figure 1: Comparison of *snapshot differential* output and UNIX *diff* output

Internet. For it to become an information source of a data warehouse, either periodic snapshots of the database must be taken (which are available on-line) or daily updates need to be incorporated.

The *snapshot differential* problem, is not unlike the problem of finding the difference between two text files (which is obtained using common utility programs such as the UNIX *diff*). The figure above illustrates some of the similarities and differences between the two problems. When the UNIX *diff* is run on files F and G, the output file indicates which lines have to be deleted (those prefixed by $<$) and inserted (those prefixed by $>$) into file F in order to make it identical to file G. If files F and G are considered to be snapshots, the *snapshot differential* of files F and G is just the row $< K3, B3 >$. The reason for this discrepancy is that the UNIX *diff* program is sensitive to the ordering of the lines in the file while the *snapshot differential* is not. Thus, the *snapshot differential* problem cannot be solved by a simple application of text file comparison programs.

The *snapshot differential* problem is also similar to the problem of performing joins between two relations. When a join is performed between two relations, the tuples of the two relations are matched. If the matching tuples satisfy the join condition, then the tuples are combined to form a tuple of the output relation. However, the *snapshot differential* problem is not only concerned about the matching rows but it is also concerned about the unmatched rows. For example, in Figure 1, the row $< K3, B3 >$ in file F does not match with any rows in file G but it is a part of the *snapshot differential* output. This implies that the *snapshot differential* problem cannot be solved by simple application of the join algorithms either.

Even though the *snapshot differential* problem cannot be solved by simply applying the methods used in text file comparison and joining relations, the problem is similar enough to these two problems so that it may be possible to derive *snapshot differential* algorithms from these methods. Moreover, these derived algorithms may lend themselves to optimizations that were not applicable to the two problems mentioned. For example, since the two snapshots pertain to the same data source, the *snapshot differential* problem can be likened to joining a relation with itself which may have useful implications for *snapshot differential* algorithms. On the other hand, conventional join algorithms were designed to join two different relations (with at least one pair of compatible fields). In this paper, we derive *snapshot differential* algorithms that are based on the methods for comparing files and joining relations. These derived algorithms are then further optimized by taking advantage of the specific properties of the snapshots.

In the next section, we present the problem formulation, introduce some terminologies, and elaborate on the differences (and similarities) of the three problems mentioned. We present and categorize algorithms that can solve the *snapshot differential* problem in section 3. We then identify possible optimizations of the algorithms discussed based on assumptions that can be made on the two snapshot files in section 4. In section 5, we identify possible performance metrics and compare the various algorithms suggested based on these metrics. We investigate the applicability of the new algorithms to the related problem of performing outer joins in section 6. Lastly, we present some conclusions and future directions in section 7.

2 Problem Formulation

2.1 Some Definitions

We assume the snapshots to be files that have the form $\{R_1, R_2, ...R_n\}$ where R_i denotes the rows and each row R_i is of the form $< K, B >$, where K is the key and B is the rest of the row representing one or more fields. Without loss of generality, we refer to B as a single field in the rest of the paper. The problem of finding the difference between two snapshots is given two snapshots, F_1 and F_2 (the later snapshot), produce a file F_{OUT} that also has the form $\{R_1, R_2, ...R_n\}$ and each row 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 differences of the two snapshots are captured succinctly with these three forms of rows. The first form is produced when two rows, R_i of file F_1 and R_j of file F_2, have matching keys (K_i and K_j) but have different B fields. The row R_i of the earlier snapshot has been modified to row R_j. Note that the two matching rows may be in different positions of their respective files. The second form is produced when there exists a row $< K_i, B_i >$ in file F_1 (the earlier snapshot) but there are no rows with a matching key in file F_2. In other words, this row has been deleted from the earlier snapshot. Lastly, the third form is produced when there exists a row $< K_i, B_i >$ in file F_2 but there are no rows with matching key in file F_1, which implies that the row has been inserted. In later sections, we will refer to the first form as *updates*, the second as *deletes* (or *deletions*) and the third as *inserts* (or *insertions*). The first field is only necessary in distinguishing between the *updates* and the *inserts*. It is included for clarity in the case of *deletes*. These three forms (without the first field in case of *deletes*) represent the minimal information needed to update the data warehouse. That is when a row is deleted from the earlier snapshot, we need to convey to the data warehouse only the key of the row that has been deleted and nothing more. On the other hand, if a row is inserted into the new snapshot, we need to convey the whole row to be included in the data warehouse. In the case of an updated row, we need to convey new value of the modified fields of the matching rows and the key of either two rows (since the two keys are identical for a match).

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 row in F_{OUT}. Based on the form of the row, either an *update*, *insert* or *delete* message may be sent. Another reasonable way of sending the modifications is to send the whole file F_{OUT} as one message. However, the size of the file may be too large for the network to accommodate and smaller chunks of the file may be sent instead. We assume, for the rest of the paper, that one message is sent for each row in F_{OUT} and that the message may be one of the three types mentioned.

The *snapshot differential* problem, as defined above, requires that the minimum number of differences are found. That is, it does not allow an output of say $< Delete, K_i >$, followed by $< Insert, K_i, B_i >$. If an algorithm similar to the text comparison algorithms is used, then it is possible that these kinds of output are produced. However, the output can be filtered to produce an output that conforms to the specification of the *snapshot differential* problem. Thus, when we discuss some of the algorithms in the next section, we temporarily relax the constraint that the minimum number of differences are found and investigate ways to remedy the situation.

2.2 Similarities with Joins and Outer Joins

The *snapshot differential* problem is similar to the problem of performing a join between two relations. An example of a join operation is shown below in SQL.

select $F1.J$, $F2.J$
from $F1$, $F2$
where $F1.J = F2.J$ **and** $F1.C \neq F2.C$

The join operation matches tuples from two different relations that satisfy a join condition. In the example above, the join condition is that the C fields must match. If a pair of tuples match, new tuples of possibly different format are produced (based on the matching tuples). The *snapshot differential* problem involves matching rows from the two snapshots with the same K field (as defined in section 2.1). In fact, finding the differences between the two snapshots involves performing a conceptually identical operation to the SQL operation above. More specifically, the rows from the two snapshots with matching K fields but with different B fields are used to produce the *updates*. In order to find the *deletions* and *insertions*, the rows of one file that do not match with any rows from the other file have to be reported. These rows that do not have matching rows have been called "dangling" rows.

The outer join was defined to handle these "dangling" tuples as well. The outer join performs a corresponding innerjoin and concatenates the keys of the "dangling" tuples of both relations to the result appended with nulls. In contrast, the *snapshot differential* problem appends the keys of "dangling" rows in the case of *deletes* and *inserts*, and it also appends the modified fields for *inserts*. Apart from this difference, the *snapshot differential* problem can be considered to be an outer join. It then follows that algorithms for the outer join can be used in solving the *snapshot differential* problem. Clearly, high performance join algorithms can be adapted to outerjoins. Unfortunately, we have not encountered any discussion of outerjoin algorithms in literature.

On the other hand, there is a plethora of conventional join algorithms in literature. In solving the *snapshot differential* problem, only the *ad hoc* join algorithms can be exploited since the "snapshot differential" process can only start at the moment the second snapshot arrives. In other words, the absence of the second snapshot makes it impossible to build pre-computed access structures across the two files (such as join indices). We now proceed to apply the general structure of *ad hoc* join algorithms to the *snapshot differential* problem.

All the *ad hoc* join algorithms in literature can be decomposed into three stages:

1. Partitioning stage
2. Matching stage
3. Merging stage

The first stage partitions the data so that less work will be done in subsequent stages. In the case of a simple nested-loop join, no partitioning is done. The matching stage is concerned with finding the tuples that match based on a certain criteria (the join condition). Upon finding the tuples that match, the tuples are merged into a certain format in the last stage.

The *snapshot differential* problem can also be broken down into these three stages. The partitioning stage can be done to simplify the work to be done in subsequent stages. In fact, the partitioning techniques that have been used in join algorithms (e.g. sorting, hashing) are also applicable to the new problem. We will elaborate on this in the next section. The *snapshot differential* problem has a more difficult matching stage because not only does the problem require producing rows that match, but it also requires producing rows from each file that do not match with any rows from the other file. This may seem trivial at first, but some of the join algorithms are not able to detect these "dangling" rows. The matching rows are merged in the merging phase to produce a row of the form $< Update, K_i, B_j >$. The "dangling" rows in file F_1 file are appended to produce a row of the form $< Insert, K_i, B_i >$. Lastly, the "dangling" rows in F_2 file are produced as output ($< Delete, K_i >$ row form). In the next section, we modify specific *ad hoc* join algorithms to solve the *snapshot differential* problem.

2.3 Similarities with Text Comparison

There are a number of text comparison programs available on different platforms. For example, in most UNIX platforms, the *diff*, *bdiff*, and *comm* programs compare text files. In DOS, a similar program named *comp* can be used. For conciseness, we will use the UNIX *diff* as a representative of this class of programs.

The UNIX *diff* program compares two arbitrary files. Although *diff* will be able to detect the differences between the two snapshot files, it does not solve the *snapshot differential* problem. As illustrated in Figure 1, the UNIX *diff* produces three differences between the two files. A solution to the *snapshot differential* problem only has one difference between the two files. There are a number of ways that the UNIX *diff* can be modified to satisfy this requirement and we elaborate on this in the next section. It is interesting to note that if messages are sent to update the data warehouse (as described in section 2.1) based on the UNIX *diff* output, the final state of the warehouse will still be correct. However, unnecessary messages might be costly especially if the network link connecting the information source and the data warehouse is slow; and there may be a lot of unnecessary messages if the later snapshot does not maintain any semblance of the order of the rows of the earlier snapshot.

Another problem that arises when using the UNIX *diff* to compare two database snapshots is that the *diff* program uses lines as the unit of comparison when the unit of comparison of the *snapshot differential* problem is a row. A row in a snapshot corresponds to a line if and only if the database dump routine uses the carriage return as the row delimiter. However, the unit of comparison of the *longest common subsequence* (LCS) algorithm, upon which the UNIX *diff* is based, is arbitrary. Thus, we will assume that the problem just mentioned is not intrinsic and proceed to analyze the possibility of using *diff* in the next section.

3 Solutions

We present several solutions to the *snapshot differential* problem in this section. Although the algorithms are presented in turn, it will be evident that the algorithms can be grouped into categories based on certain characteristics. Thus, the algorithms are categorized at the end of this section.

The discussions of the different algorithms refer back to the definitions stated in section 2. In addition, we denote the size of file F as $B(F)$ blocks or $R(F)$ rows. We also assume that the key K is unique for each file. We will relax this constraint in section 4.

3.1 Nested-Loop Join Algorithm

The nested-loop join algorithm is as follows: for each item of file F_1 (the outer file), scan the entire file F_2 (the inner file) and find matches. A number of improvements can be made to this naive nested-loop join. First of all, if the keys are unique, the scan of file F_2 can be terminated prematurely once a match is found. Moreover, the entire file F_2 can be scanned once for each block of F_1. Thus, the nested-loop join requires on the average $B(F_1) + B(F_1) * B(F_2)$ *IOs* to find the matching rows of the two files. The matching rows satisfying the join condition are produced as output.

In order to solve the *snapshot differential* problem, an algorithm needs to detect the updated rows of F_1, deleted rows of F_1 and the inserted rows of F_2. The nested-loop join algorithm, as described above, can *only* detect the updated rows of F_1. In order to detect the other two kinds of rows, additional structures need to be incorporated in the algorithm. An array A_{OUTER} which records the rows of outer file F_1 that have not been matched may be defined. The size of the array A_{OUTER} is the number of rows that fits in one block $(R(F_1)/B(F_1))$. Each time a block of file F_1 is read, A_{OUTER} is initialized to indicate none of the rows has been matched. After scanning the entire file F_2, A_{OUTER} is checked for unmatched rows which represent rows deleted from F_1. With this array, the rows inserted into file F_2 can also be detected. However, this requires performing *another pass* of the nested-join algorithm with F_2 as the outer file and F_1 as the inner file (the matching rows are not produced as output). To avoid performing a second pass, a second array A_{INNER} that keeps track of the unmatched rows of F_2 (the inner file) has to be defined. Unlike A_{OUTER}, the size of the array A_{INNER} is $R(F_2)$. Assuming the arrays A_{INNER} and A_{OUTER} fits in memory, the modified nested-loop join algorithms still needs on the average $B(F_1) + B(F_1) * B(F_2)$ *IOs*.

3.2 Merge-Join Algorithm

The merge join algorithm requires that the two input files are sorted on the join attribute (the key K in our problem definition). The two sorted files are scanned and any matching rows that satisfy the join condition are produced as output. The merge join algorithm as described incurs on the average $4 * B(F_1) + 4 * B(F_2)$ *IOs* assuming the sort phase can be done in two passes.

Unlike the nested-loop join algorithm, the merge join algorithm solves the *snapshot differential* problem *without* any additional data structures. However, the output-logic of the merge join algorithm needs to be changed. More specifically, if during the execution of the merge phase of the algorithm, the current rows being scanned are: $< K_i, B_i >$ for F_1 and $< K_j, B_j >$ for F_2, the following output is produced.

1. $< Delete, K_i >$ if $K_i < K_j$
2. $< Insert, K_j, B_j >$ if $K_j < K_i$
3. $< Update, K_i, B_j >$ if $K_j = K_i$ and $B_i \neq B_j$

In the original merge join algorithm, only the third kind of output is produced. With this slight modification of the output logic, merge join also produces the *deletions* and the *insertions*.

In the context of data warehousing, the snapshots arrive periodically. After comparing the snapshot files F_1 and F_2, the latter snapshot needs to be stored. When the third snapshot (F_3) arrives, it will be compared with F_2. In other words, there is a stream of snapshots of the database and the comparison of pairs of snapshots happens continually. The merge join algorithm can further cut down on its cost since it can save the sorted file after each comparison. For example, after comparing F_1 and F_2, the sorted F_2 file is saved. When the new snapshot F_3 arrives, only F_3 needs to be sorted. Thus, the modified merge join algorithm only needs $B(F_1) + 4 * B(F_2)$ *IOs*.

3.3 Hash Join Algorithms

There are numerous hash-based join algorithms with the grace hash join being one of the most popular. The grace hash join has two phases. In the first phase, each file is read and hashed into buckets which are written back to disk. In the second phase, the corresponding buckets of the two files are read into memory and merged together. Assuming there is enough memory, the number of buckets can be chosen such that two buckets can fit in memory in the second phase. The grace hash join algorithm incurs $2 * B(F_1) + 2 * B(F_2)$ *IOs* in the first phase and another $B(F_1) + B(F_2)$ *IOs* in the second phase.

Unlike the merge join algorithm, the hash join algorithm as described does not detect the deleted rows from F_1 and the inserted rows in F_2. For each bucket in memory in the second phase, an array needs to be defined to keep track of the rows that have not been matched (denoted as A_{BUCKET_1} and A_{BUCKET_2}). After matching the two buckets, the two arrays can identify the rows that have been deleted from F_1 and the rows that have been inserted into F_2. After these rows are produced as output, the arrays A_{BUCKET_1} and A_{BUCKET_2} are initialized (to indicate that no rows have been matched) for the next pair of buckets. The total size of the two arrays is approximately $R(F_1) * B(M)/B(F_1)$ where $B(M)$ is the size of the memory in blocks (the A_{INNER} array in nested-loop join can be as large as $R(F_1)$). Therefore, the sum of the sizes of the two arrays is significantly less than the array A_{INNER} used for the nested-loop join algorithm.

The buckets for file F_2 can be saved for the next comparison as well. When the new snapshot F_3 arrives, only F_3 needs to be hashed into buckets. Thus, the cost for the hash join is lowered to $B(F_1) + 3 * B(F_2)$ *IOs*.

Another popular hash-based join algorithm is the hybrid hash join. As in the grace hash join algorithm, each file is read, hashed into buckets and written back to disk. However, during the first phase, a portion of the memory is reserved for k in-memory hash buckets for the first file. These k buckets are never written out. When the second file is hashed, the rows in the second file that match with one of the k in-memory buckets can be merged immediately (without writing it to disk). The equation below determines the maximum value of k given $B(M)$ blocks of memory and $B(F)$ blocks corresponding to the number of blocks of the larger file.

$$N = \lceil \sqrt{B(F)} \rceil$$

$$k = (B(M) - (N + 1))/((\lceil B(F) \rceil / N) - 1)$$

The number of buckets, denoted as N, needed to ensure that the first phase of the algorithm is performed in two passes is first calculated. The numerator of the equation for k represents the number of blocks of memory left after allocating one block per bucket and one input buffer. The denominator represents the memory requirement for each bucket in memory. The *IO* cost of the algorithm is given below.

$$IOcost = (B(F_1) - k * B(F_1)/N) + B(F_2) + 2 * (B(F_2) - k * B(F_2)/N)$$

The first term is the cost incurred in reading the buckets F_1 assuming the k buckets are kept in-memory. The second term is the cost incurred in reading file F_2 to bucketize it. The last term is the cost for writing out the buckets and reading it back in for the merging phase.

Furthermore, assuming there are no duplicate keys, the space in the k in-memory buckets can be reused. For example, given an in-memory bucket that contains the rows with keys that hash to a value l, the bucket will "empty" as rows in the second file with matching keys are read in (the row in the bucket can be deleted based on the assumption of no duplicate keys). The space in the bucket can then be used to service rows of file F_2 with keys that hash to a value m. Assuming we have a hash function that satisifies the condition of uniformity, the rate that rows with a key hash value of l are read is about the same as rate for rows with a key hash value of m. Thus, in the best case, an additional k buckets do not have to be written out. The cost of the hybrid hash join is shown below (note the improved third term).

$$IOcost = (B(F_1) - k * B(F_1)/N) + B(F_2) + 2 * (B(F_2) - 2 * k * B(F_2)/N)$$

3.4 Using SQL

The *snapshot differential* problem can also be solved by writing a relatively simple SQL query. The SQL query is as follows.

select $R1.K$, $R1.B$, $R2.B$
from $R1$, $R2$
where $R1.K = R2.K$ **and** $R1.B \neq R2.B$
$\cup$
select $R1.K$
from $R1$
where not exists (**select** * **from** $R2$ **where** $R2.K = R1.K$)
$\cup$
select $R2.K$, $R2.B$
from $R2$
where not exists (**select** * **from** $R1$ **where** $R2.K = R1.K$)

However, this method is inefficient for several reasons. First of all, the two snapshot files need to be loaded into the database (into relations $R1$ and $R2$) which can be very slow for large files. Secondly, the join operation does not take advantage of the fact that the relation $R2$ will be used again in joining with $R3$ (the relation obtained from the next snapshot). Lastly, this method performs two additional select operations which in the worst case can cost $B(F_1) * B(F_2)$ *IOs*.

3.5 Using the UNIX diff

The UNIX *diff* program finds the *longest common subsequence* (LCS) of lines of the two files. It then takes out the lines that comprise the LCS from both files since these are lines that are identical

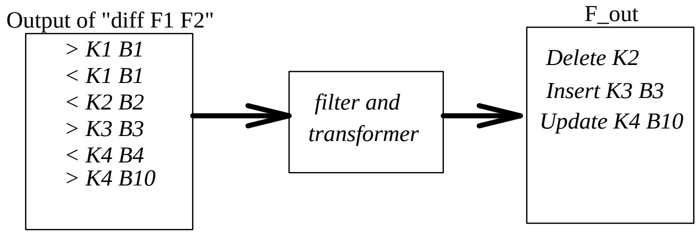

Figure 2: Transforming the UNIX *diff* Output

and occur in the same relative order. The remaining lines are used to produce the list of differences between the two files. More specifically, it produces an output file specifying how the first file can be converted into the second file. For instance, in Figure 1, the output file indicates that in order to transform file F_1 into F_2, the first line must be deleted and the same line must be inserted into the second line and so on. The output file can then be used to produce the update messages to the data warehouse by sending an *insert* message for each line in the output file prefixed by "$>$" and by sending an *delete* message for lines prefixed by "$<$". If the *diff* program is used to solve the *snapshot differential* problem, the final output has to be modified because of two reasons. First of all, the modifications listed in the *diff* output are not minimal. For instance, in Figure 2, the *diff* program suggests deleting $< K4, B4 >$ and inserting $< K4, B10 >$ (the line numbers are omitted). This requires two messages to be sent to the data warehouse. We call these messages *"delete-insert"* pairs. However, a message *"Update, K4, B10"* can convey the same effect in the warehouse using only one message. Moreover, *identical* rows which appear in different positions in F_1 and F_2 also produce two update messages (*"Delete, K1, B1"* and *"Insert, K1, B1"*) when no messages are needed. We call these messages *useless "delete-insert"* pairs.

If the bottleneck is the network link between the information source and the data warehouse, it is useful to streamline the output of the *diff* program by transforming *"delete-insert"* pairs into a single update message if possible and filtering out *useless "delete-insert"* pairs. On the other hand, if the network link is not a bottleneck, then the output of *diff* can be used to produce the necessary delete and insert messages to keep the data warehouse up to date.

If the UNIX *diff* detects a lot of differences between the two files, the output file of the *diff* program might be comparable in size to the original input files. Moreover, if the ordering of the two files are entirely different, a large fraction of these differences may result in *useless "delete-insert"* pairs. This implies that the postprocessing step as discussed above might be costly since it requires matching identical rows (which may cost $B(F_1) * B(F_2)$ *IOs*). If this is the case, it is beneficial to use hashing or sorting to process the output file in an efficient manner. If sorting is used, the *useless "delete-insert"* pairs can then be eliminated by scanning through the sorted output file. Assuming we use the multi-way merge sort algorithm to sort the file and that the algorithm can be done in two passes (there is enough memory), the cost of filtering the *diff* output is $3 * B(F_{OUT})$ (F_{OUT} is the UNIX *diff* output in this case).

Another downside of using the UNIX *diff* program is that it cannot handle very large files. In

Algorithm	Minimal-Set Producing	Incremental	Probabilistic
Nested-Loop Join	YES	YES	NO
Merge Join	YES	YES	NO
Grace Hash Join	YES	YES	NO
Hybrid Hash Join	YES	YES	NO
Using SQL	YES	YES	YES
UNIX *diff* based	NO	NO	NO
PROBMerge Join	YES	YES	YES
PROBHash Join	YES	YES	YES
Window	NO	YES	NO

Figure 3: Table 1 Properties of *Snapshot Differential* Algorithms.

order to compare very large files, the UNIX *bdiff* program is used. The UNIX *bdiff* splits the files into sections and forks a *diff* process to compare the sections. As a result of this segmenting, the *bdiff* produces more file differences than when *diff* is run. Moreover, there is the overhead of forking of a process and interprocess communication (which is done through pipes).

The UNIX *diff* program works best in terms of disk *IOs* when there are no duplicate *rows*. Since there are usually no duplicate rows in databases, the *LCS* algorithm only requires $2 * (B(F_1) + B(F_2)) log(B(F_1) + B(F_2))$ *IOs*. When the output file of the *diff* is large, then the cost for filtering must also be added bringing the total cost to $2 * (B(F_1) + B(F_2)) log(B(F_1) + B(F_2)) + 3 * B(F_{OUT})$ *IOs*.

3.6 Categorizing the Algorithms

The algorithms presented exhibit characteristics that can be used as a basis for categorizing them. One property that a *snapshot differential* algorithm can exhibit is that it produces the minimal set of differences between the two snapshots. The join based algorithms fall under the *minimal-set* producing algorithms. The UNIX *diff* based algorithm produces a correct but a non-minimal set of differences between the two snapshots.

Another property that an algorithm can have is that it always produces the correct answer. All the algorithms discussed in this section have this property. In the next section, we will present some algorithms that do not always produce the correct answer. We call these algorithms *probabilistic snapshot differential* algorithms.

It is interesting to note that the data warehouse can simply be updated by deleting all the data from the old snapshot and inserting the data from the new snapshot. All the algorithms except the UNIX *diff* avoid this method. The UNIX *diff* essentially deletes the old snapshot and inserts the new one in the special case when the two snapshots have opposite ordering of rows and the *LCS*

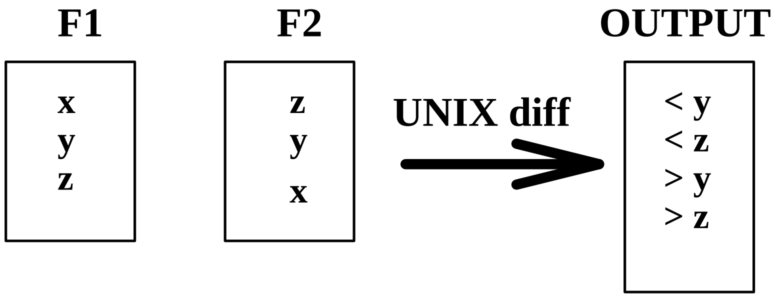

Figure 4: Nonincremental scenario for UNIX *diff*

has a length of one. This is shown in Figure 4. Thus, we call all the algorithms presented, except the UNIX *diff*, *incremental snapshot differential* algorithms.

The table above categorizes all the algorithms presented in this paper. The last three algorithms in the table are optimized algorithms presented presented in the next section.

4 Optimizations and Extensions

In this section, we describe optimizations of the various algorithms presented. The primary optimization that is used in the next subsections is compression. Compression can be performed in varying degrees. For instance, compression may be performed on the rows of a file by compressing the whole row (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 row that are not important in the comparison process. We ignore the details of the compression function and simply refer to it as $compress(x)$ in the next subsections.

There are a number of benefits from processing compressed data. First of all, the compressed intermediate files are smaller. Thus, there will be less *IO* operations to read in the file to be processed. Moreover, if compression is done to a large degree, the entire compressed file may fit in memory. Even if the entire compressed file cannot fit in memory, a join algorithm may still benefit. For example, if the hybrid hash join is used (as described in section 3), compression may allow the algorithm to keep more in-memory hash buckets.

However, compression is not without disadvantages. The compression function, $compress(x)$, may map two different values of x into the same compressed value which will have repurcusions in the optimized algorithms. This is discussed in more detail in the next sections. On the other hand, a loss-less compression function does not have this problem. An example of a loss-less compression function is using Huffman encoding when compressing integers. This guarantees that different integer values are mapped to different compressed values. However, loss-less compression often results in lower compression factors (the ratio of the original size to the compressed size of the data).

In the next subsection we present an optimized algorithm for the general scenario of the *snapshot differential*. In subsequent subsections, we describe optimizations of the algorithms to efficiently handle various realistic scenarios. It was also assumed in the previous section that the keys of the files are unique. We relax this constraint in the last part of this section.

4.1 Using Compression to Optimize the Join Algorithms

We assume that the compressed file of F_1 (denoted as f_1) was produced in the previous snapshot comparison (as in Sections 3.1 to 3.3). The original and compressed rows are shown in Figure 5(b) and Figure 5(c) respectively. That is, the B field is compressed to b and the compressed value is kept in the row of f_1. The compression factor, u, is the ratio of the size of F_1 to f_1 ($B(F_1)/B(f_1)$). We can now show the optimized join algorithms.

In the case of merge join, the algorithm starts out the same way. That is, we assume that the compressed file for the first snapshot is sorted (which is file f_1 instead of F_1). When file F_2 arrives, it is sorted incurring $2 * B(F_2)$ *IOs*. In the merging phase, the file f_1 (incurring $B(F_1)/u$ *IOs*) and the sorted file F_2 ($B(F_2)$ *IOs*) are both read into memory. If the current rows being scanned are: $< K_i, b_i >$ for f_1 and $< K_j, B_j >$ for F_2, the field B_j is first compressed to b_j and the following output is produced.

1. $< Delete, K_i >$ if $K_i < K_j$
2. $< Insert, K_j, B_j >$ if $K_j < K_i$
3. $< Update, K_i, B_j >$ if $K_j = K_i$ and $b_i \neq b_j$

In addition, the row $< K_j, b_j >$ is constructed and is eventually written to disk to file f_2 (the compressed sorted file of F_2). Thus, the total cost incurred is $3 * B(F_2) + B(F_1)/u + B(F_2)/u$ *IOs*. Assuming a compression factor of 10 and that the files are comparable in size, the *IO* cost is $3.2 * B(F)$ (where $B(F)$ is the size of the files). This translates to 20 % fewer *IO* operations. Note however that the improvement in terms of *IO* operations cannot go beyond 25 % (when u is extremely large).

The grace hash join is modified in a similar fashion when compression is used. That is, we assume that the first snapshot has its compressed buckets on disk. When file F_2 arrives, it is bucketized which incurs $2 * B(F_2)$ *IOs*. In the merging phase each pair of buckets is examined in turn and the appropriate output is produced (as in Section 3.3). In addition, the compressed bucketized file of file F_2 is produced when examining the buckets of file F_2 during the merging phase. The total cost incurred is $3 * B(F_2) + B(F_1)/u + B(F_2)/u$ *IOs*. Again, this translates to 20 % fewer *IO* operations using the assumptions made for the merge join.

For the nested loop join, it is easy to see that compression lowers the *IO* cost to $B(F_1)/u + B(F_2) * (B(F_1)/u) + B(F_2)/u$ (the last term is the cost for producing the compressed file for file F_2). Assuming the files are large and comparable in size, a compression factor of 10 can save as

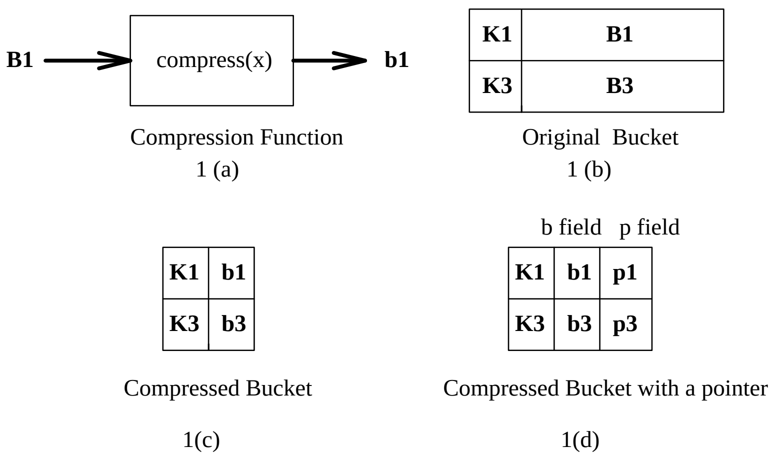

Figure 5: Hash Function for Compression And the Compressed Structure

many as 90 % of the *IO* operations. In the extreme case that the compressed file f_1 fits in memory, the *IO* cost of the algorithm is $B(F_1)/u + B(F_2) + B(F_2)/u$ as well.

In the case of hybrid hash join, the algorithm still benefits even if the entire compressed file f_1 does not fit in memory since compression allows more buckets to be kept in memory. The equations below show why this is the case.

$$N = \lceil \sqrt{B(F)} \rceil$$

$$K = (B(M) - (N+1))/((\lceil B(F)/u \rceil /N) - 1)$$

$$k = (B(M) - (N+1))/((\lceil B(F) \rceil /N) - 1)$$

As in Section 3.3, N denotes the number of buckets for the uncompressed file F_2. The second equation computes the number of buckets that can be kept in memory (denoted as K) for the the hybrid hash algorithm with compression. The third is for the unoptimized algorithm as described in Section 3.3. It is evident from above that $K > k$. As a result, there are more in-memory buckets which leads to a more economical *IO* cost as shown below. Compression also improves the *IO* cost for reading the file F_1 (first term in the equation below) but an additional $B(F_2)/u$ operations needs to be made to produce the compressed file F_2 buckets.

$$IOcost = (B(F_1) - K * B(F_1) * N)/u + B(F_2) + 2 * (B(F_2) - K * B(F_2)/N) + B(F_2)/u$$

Compressing the B field does not come without a price. The compression function $compress(x)$ can compute the same value given two different inputs. If the B field has p bits and the compressed value has p bits, then $2^n/2^p$ *different* B values produce the same compressed B values. Thus the probability that a modification to the B field will not be detected is $((2^p/2^n) - 1)/2^p$. If p is much larger than n, this simplifies to 2^{-n}. This analysis does not only apply when compressing a single field. It applies in the general case when a value with p bits is being compressed to n bits. The optimized algorithms presented above are examples of a *probabilistic algorithms* since it may not detect all the modifications (the PROBHash join and PROBMerge join algorithms in Table 1).

4.2 Growing Files

In some scenarios, the database snapshots monotonically increase in terms of size. This happens when the data is a *history file* or a *log file*. In this scenario, there are no deleted or updated rows in F_1 but there are rows inserted into F_2. We call the snapshot files *growing files* for obvious reasons.

The merge join algorithm and the hash join algorithms can take advantage of this scenario. There is no longer any need to compare the B fields of matching rows since there are no updated rows. In the case of the merge join algorithm, instead of producing a sorted file with all the fields of the original file, it can simply produce a compressed sorted file (denoted as f) with just the key field K and a field containing a pointer to the corresponding record in the original file. During the execution of the merge phase of the algorithm, if the current rows scanned are: $< K_i, p_i >$ for f_1 and $< K_j, p_j >$ for f_2, and $K_j < K_i$, the pointer p_j is used to access F_2 in order to obtain B_j and produce the output $< Insert, K_j, B_j >$. If the key and the pointer occupy $1/u$ of the total space occupied by a row, the size of the compressed sorted file, f_1, is $B(F_1)/u$ blocks. The optimized algorithm incurs $B(F_2) + B(F_2)/u$ *IOs* to read the second snapshot for sorting and write the sorted file f_2 to disk. The two compressed files are then read into memory for comparison which incurs $B(F_1)/u + B(F_2)/u$ *IOs*. Assuming I insertions are detected, and that each insertion requires a random *IO*, an additional I *IOs* are needed. The total cost of the optimized algorithm is $B(F_1)/u + B(F_2) + 2 * B(F_2)/u + I$ *IOs*. The savings in *IO* operations is dependent on the number of insertions I and on the compression factor u. If $B(F_2)$ is much larger than I (and assuming $u = 10$ and $B(F_2) = B(F_1)$), then the optimized algorithm performs 67.5 % less *IO* operations. A similar savings can be made for the hash join algorithms as well.

4.3 Stagnant File

In some scenarios, the database snapshot may reach a point where only few changes are made to the data (most transactions are read only). When this happens, we call the snapshot files *stagnant files* since there are only few changes between the two snapshots. Unlike the *growing files* scenario, an optimized algorithm for this case needs more information than just the keys during the matching phase. However, since there are only a few updates, it may be wasteful to keep the whole row especially if the B field is large.

In the case of the grace hash join algorithm, space can be conserved in the "bucketized" file by compressing the B field. In addition, a pointer (p field) to the corresponding record in the original file is needed (Figure 5(d)). During the matching phase, if an updated row or an inserted row is detected, the row needs to be brought into memory using the pointer field. Again, we denote the ratio of the original size of the row and the compressed size of the row as u.

This algorithm incurs $B(F_2) + B(F_2)/u$ *IOs* to read the second snapshot and to write the compressed buckets to disk. The "bucketized" files have to be read into memory for comparison ($B(F_1)/u + B(F_2)/u$ *IOs*). Assuming we detect I insertions and U updates, only $B(F_1)/u + B(F_2) + 2 * B(F_2)/u + I + U$ *IOs* are needed to compare the two snapshots F_1 and F_2. This can result in as much as 75 % savings (as in Section 4.2).

Note that deletions do not require any extra *IO* since only the keys of the deleted rows need to be reported which are already in the bucket structure. A similar savings in *IO* can be achieved in the merge join and the hybrid hash algorithms. For all these algorithms, the savings gained from compressing the rows diminishes as I and U increase (the file becomes less stagnant). However, compressing the rows may still be feasible even if the file is not stagnant, as long as the compression factor u is large. The disadvantage in using this scheme is that the algorithm is *probabilistic*. It does not detect all the *updates* since it used the compressed values of the B field for comparison.

The optimized algorithm just described assumes that there are still insertions and deletions made to the database. The merge join and hash join can be optimized even further if we assume that there are few updates *and* there are no insertions or deletions. In this scenario, there is no need to keep the keys in the compressed file since the keys are only used to detect insertions and deletions between snapshots. The compressed row will just have the b field and the p field.

We can also take advantage of the fact that accessing a random row requires accessing one block from the disk. Instead of having each compressed row $< K, b, p >$ represent one original row $< K, B >$, a compressed row can represent a *whole block*. The function $compress(x)$ in Figure 5(a) is now modified to add all the B fields of the rows in one block. Again there is the possibility that two different blocks may "produce" the same value. However, since we are now considering a whole block, the range of values of the hash function may be larger which makes collisions less probable. This idea can be extended by arranging the compressed value to represent, not one block, but k blocks. The main advantage of letting b represent k blocks is smaller intermediate files. We derive an expression for the size of a compressed file F for this generalized form of file compression. The number of rows that result is $B(F)/k$ since one compressed row represents k blocks. Since the number of blocks per compressed row is given by $B(F)/(u * R(F))$, the number of blocks for the compressed file F is $B(F)^2/(u * k * R(F))$. The disadvantage of making b represent $k > 1$ blocks, is that k blocks have to be read in when the b values do not match. Thus, the total *IO* cost is $B(F_1)^2/(u * k * R(F_1)) + B(F_2) + 2 * B(F_2)^2/(u * k * R(F_2)) + U * k$ (note that $B(F)/u > B(F)^2/u * R(F)$ since $R(F) > B(F)$). However, even with this scheme, the maximum savings in *IO* is still 75 %.

4.4 Shrinking File

The *shrinking file* scenario is similar to the *growing file* scenario. Instead of having only insertions between the two snapshots F_1 and F_2, there are only *deletions* in this case. A similar optimized algorithm can be adopted as in the *growing file* scenario. The main difference is that only the keys are kept in the structure used in the matching phase. The pointer is no longer needed because in the event that a delete is detected, only the keys of the deleted row need to be reported to the data warehouse. This information is contained in the intermediate structure already.

With this optimization $B(F_1)/u + B(F_2) + 2 * B(F_2)/u$ *IOs* are needed to compare the two snapshots F_1 and F_2 for both the merge and hash join algorithms. The analysis is similar to the previous two scenarios.

4.5 Similar Dumps

The location of the rows in the snapshots is not significant in the *snapshot differential* problem. However, more efficient algorithms may be possible if the snapshots maintain "enough" similarity in terms of the location of the rows. To make the concept of similarity between two files more precise, we define the *distance* of two matching rows (rows wth matching key fields). If a row is in the pth block of file F_1 and the matching row is in the qth block of file F_2, the matching rows have a *distance* of $|p - q|$ blocks. For rows that do not have a matching row in the other snapshot, the *distance* is not defined. Note that the relative order of rows within files is irrelevant in this definition. We can now describe the *divergence* of two files in terms of a divergence function (denoted as $D(r, d)$) which depends on the *distance* of matching rows. If two files have 20 % of the matching rows with a distance of more than 10 blocks, the two files are deemed to have a $D(0.20, 10)$ *divergence*. Two files have high *divergence* if the parameter r is close to 1 and the parameter d is high. In a *similar dumps* scenario, the two files have low *divergence*.

We now describe an algorithm which takes advantage of files with low *divergence* and then measure its effectiveness using the divergence function $D(r, d)$. The algorithm , which we call the *window* algorithm, is as follows.

1. Initialize the variable *count* to 0.
2. Given M blocks of memory available, read $M/2$ blocks of file F_1 and $M/2$ blocks of file F_2.
3. Detect all the matching rows that are in memory and produce the *updates* when a modification is detected.
4. The rows that have been matched surrender the space that they occupy. The rest of the rows (the unmatched ones) are reorganized to eliminate fragmentation, freeing as many blocks of memory as possible. In addition, these unmatched rows are tagged with *count*. The *count* is incremented by one after all the matching rows have been found.
5. Repeat Step 2 until there are no more blocks to be read. However, the amount of memory available M is now decreased since the unmatched rows occupy some of the blocks. If there is not enough memory to read in new blocks, we eliminate enough of the unmatched rows, starting with the rows with the smallest tags. We produce the *deletions* for the rows eliminated from the portion of memory for file F_1 and *insertions* for file F_2.

The number of matching rows in Step 3 depends on the *divergence* of the file. If the *divergence* of the two files is $D(0.0, d)$, this means that all the matching rows are within d blocks of each other.

Thus, in the case where d is equal to $M/2$ (half the memory size), all of the memory can be freed up since after matching the rows, we are guaranteed that the unmatched rows are either inserted or deleted rows. A *divergence* of $D(0.0, M/2)$ is reasonable if the memory is large. For example, if the system has 256 MB of memory with 1 KB blocks, then a *divergence* of $D(0.0, M/2)$ means that matching rows need to be $131,072$ blocks of each other. Thus in cases where there is large memory or/and low *divergence*, the *IO* cost can be just $B(F_1) + B(F_2)$. This translates to 50 % savings in terms of *IO* operations *without* compression (it is not *probabilistic*). If the *divergence* of the two snapshot files are larger than $D(0.0, M/2)$, the *window* algorithm may produce *useless delete-insert* pairs (Section 3.5) which makes the algorithm *non-minimal-set producing.*

This algorithm can benefit from compression as well. Instead of reading in the file F_1, a compressed file f_1 can be read in instead. If the compression factor is u, then we can allocate $M/(2*u)$ blocks to file F_1 and $M*(2*u-1)/(2*u)$ blocks to file F_2. If the compression factor is 10, for instance, the blocks allocated to file F_2 is $0.95*M$. The *divergence* of the two files required to produce a cost of $B(f_1) + B(F_2) + B(f_2)$ *IOs* is $D(0.0, 0.95*M)$. Thus, not only does the *IO* cost decrease, but also the required *divergence*.

A separate study was made by [GL95] to analyze the performance of the *window* algorithm for files with higher *divergence*. In the study, they show that the performance of the *window* algorithm is comparable to the hybrid hash join and merge join algorithms even with files of relatively high *divergence*.

A more restrictive scenario than the *similar dump* is the *ordered dump* scenario, wherein not only do the files have low *divergence*, but the files also maintain the same relative ordering of rows. The *window* algorithm handles this case since it is more restrictive than the *similar dump* scenario. However, it is the UNIX *diff* that benefits from this scenario. Since the relative ordering of rows is maintained, the algorithm does not produce the *useless delete-insert* pairs that was discussed in section 3.5. This means that the algorithm does not require expensive post-processing, which lowers the *IO* cost to $2*(B(F_1) + B(F_2))log(B(F_1) + B(F_2))$.

4.6 Handling Duplicate Keys

Until now, it has been assumed that the key K is unique. The figure below (Figure 6) shows how the hash join algorithm can be extended to handle duplicate keys. A similar matching can be used for the merge join and hybrid hash join algorithms as well. The figure also illustrates the matching that produces the minimal number of messages to the data warehouse. The matching phase can be decomposed into three stages.

1. Match identical rows. These rows are enclosed in an oval in Figure 6.

2. Match two different rows with the same key value. The row in F_1 is an updated row (to become the matched row in F_2). These rows are enclosed in a rectangle in Figure 6.

3. After the first two stages, there will either be excess rows in F_1 or excess rows in F_2. The excess rows in file F_1 are deleted rows. On the other hand, the excess rows in file F_2 are

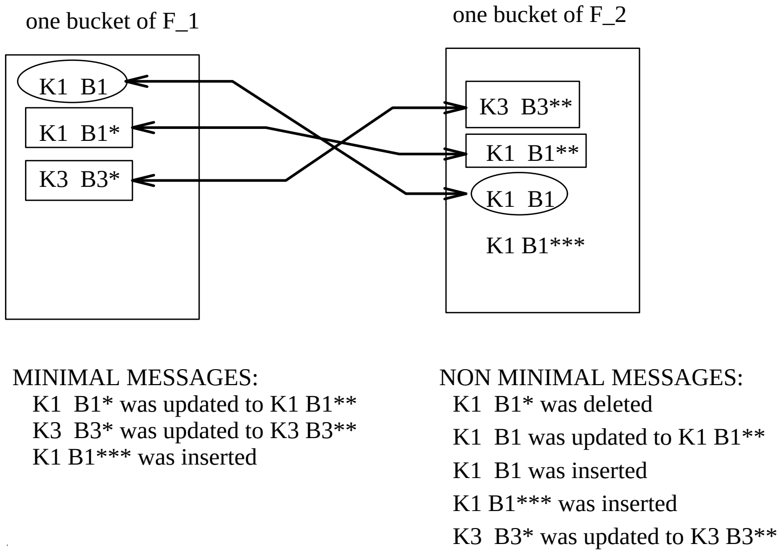

Figure 6: Handling Duplicate Keys

inserted rows. In Figure 6, $< K1, B1^{***} >$ is an inserted row.

The rows matched in the first stage do not require any messages to be sent to the data warehouse. The rows matched in the second stage require an *update* message to be sent to the data warehouse. The unmatched rows generate either an *insert* or *delete* message. It is easy to generate other sets of messages that are not minimal. One such set is shown in Figure 6 as well. The cost incurred to handle duplicate keys is additional CPU time. The number of *IO* operations is not affected since all the matchings are done between a pair of buckets. The merge join algorithm can be extended in a similar fashion.

4.7 Putting It All Together

We have enumerated several specialized scenarios and the modified algorithms that can take advantage of these scenarios. A single database may produce snapshots that fall under different scenarios. As a result, a single algorithm may not be suitable for handling all the snapshots. Optimally, we want to choose the most appropriate algorithm for each pair of snapshot files to be compared. We can approximate this by choosing algorithms based on the number of deletions, insertions and updates between pairs of snapshots that have been compared. It is also useful to capture the divergence of the two files. Lastly, we also need to keep track of how large the rows are to estimate the compression factor u. With these statistics, an appropriate algorithm can be chosen for a pair of snapshots. The choice will be based on the performance of the algorithm in a given scenario which is the subject of the next section.

5 Performance Evaluation

5.1 Performance Metrics

Since a number of the solutions suggested are based on join algorithms, the join algorithms performance metrics (number of disk accesses or the number of pages transferred) can be employed in comparing the various algorithms. Haas et. al. recently suggested execution time (based on the number of seeks, number of pages transferred and the rotational latency) as a more accurate metric in comparing *ad hoc* joins. This metric might prove useful in the context of data warehousing because it is the execution time that limits the frequency of the snapshots of the database. The lesser the execution time of the algorithm is, the more often snapshots of the database might be performed and consequently, the more accurate the information in the data warehouse will be. However, in practice, the snapshots of the database will arrive in a weekly basis. Thus, it is reasonable to adopt a simpler metric, such as the *number of IO* operations incurred.

Another performance metric which is applicable in data warehousing is the *number of messages* that need to be communicated over the network to update the data warehouse. However, this performance metric may only be useful for the UNIX *diff* based algorithm since almost all the algorithms produce a minimal set of messages.

Another performance metric is the *probability of error* of the algorithm. Some of the algorithms will not identify all the modifications that have been made on the last snapshot. If this is not tolerable in the data warehouse application, then we need to rule out these algorithms. If the application can tolerate such errors, then all the algorithms are feasible and the probability of error as a performance metric is applicable.

Lastly, the *size of the intermediate files* can also be used as a metric. For instance, the hash join and merge join algorithms use intermediate files that are about the same size as the input files. On the other hand, the nested-loop join does not use any intermediate files.

5.2 Comparison of Algorithms

We first compare the algorithms that can handle any scenario (algorithms in Section 3 and 4.1). We use the *number of IO* operations and the *size of the intermediate files* as performance metrics. We do not use the *number of messages* as a performance metric since only the UNIX *diff* based algorithm does not produce the minimal set of messages. Moreover, we assume that the output file of the UNIX *diff* is sorted as a postprocessing step (as discussed in section 3.1) to eliminate these extra messages. We assume that the size of the input buffer is the same for all the algorithms (i.e. which means that the number of *IO* operations is proportional to the number of pages transferred).

As shown in the table above, the compressed version of the join algorithms achieve the best performance in terms of *IO* cost. The compressed version of the hybrid hash join (Section 4.1) is the most economical in terms of *IO* cost. The advantage of the hybrid hash join over the grace hash join and merge join algorithms diminishes as the files get larger since less in-memory buckets

Algorithm	*IO* operations	Blocks for Intermediate Files
UNIX *diff*	$2 * (B(F_1) + B(F_2)) log(B(F_1) + B(F_2)) + 3 * B(F_{OUT})$	$B(F_1) + B(F_2)$
NL	$B(F_1) + B(F_1) * B(F_2)$	0
M	$B(F_1) + 4 * B(F_2)$	$B(F_1) + B(F_2)$
GH	$B(F_1) + 3 * B(F_2)$	$B(F_1) + B(F_2)$
HH	$(GH) - k * B(F_1)/N - 2k * B(F_2)/N)$	$(GH) - k * B(F_1)/N - k * B(F_2)/N$
Using SQL	$B(F_1) * B(F_2)$	$B(F_1) + B(F_2)$
CP NL	$B(F_1)/u + B(F_1) * B(F_2)/u + B(F_2)/u$	0
CP M	$3 * B(F_1) + B(F_1)/u + B(F_2)/u$	$B(F_1)/u + B(F_2)/u$
CP GH	$3 * B(F_2) + B(F_1)/u + B(F_2)/u$	$B(F_1)/u + B(F_2)/u$
CP HH	(HH)	(HH)

HH - Hybrid Hash Join, GH - Grace Hash Join, M - Merge Join, NL - Nested Loop Join
CP - compressed, (HH) - Hybrid Hash Entry , (GH) - Grace Hash Entry

Figure 7: Table 2 Comparison of General *Snapshot Differential* Algorithms.

can be kept. The nested loop join algorithm has the worst performance in terms of *IO*. However, it is the most conservative in terms of space requirement. The compressed version of the merge join and the hash join algorithms require less space for intermediate files when compared to the unoptimized versions. The disadvantage of the algorithms that involve compression is that some modifications may not be detected. However, this may be tolerable to some extent in the data warehouse.

There were several specialized scenarios introduced in the previous section. We now tabulate the algorithms that works best for each scenario in terms of a specific performance metric. Since some of the algorithms in section 4 fall under the *probabilistic snapshot differential* algorithms, we include the *probability of error* as a performance measure. We also denote I as the number of insertions, U as the number of updates, and D as the number of deletions. Some of the algorithms also rely on compressing the original row (as discussed in Section 4). We denote u as the ratio of the uncompressed row with the compressed row. For conciseness, we assume that the two snapshots are comparable in size.

The hash join and the merge join algorithms perform the best in terms of *IO* cost for all the scenarios, except the *similar dump* scenario. However, these algorithms requires space for intermediate files. In addition, these algorithms may not detect some modifications for the *stagnant* and *similar dumps* scenarios. The *window* algorithm works best in terms of *IO* for the *similar dump* scenario. Moreover, it does not require any intermediate files. Howeverm, with insufficient memory, the *window* algorithm becomes a *non-minimal-set producing* algorithm.

Scenario	*IO* operations	Blocks for Intermediate Files	Probability of Error
Growing Files	HH, GH, M $(B(F) + 3 * B(F)/u + I)$	NL	M, HH, GH, NL,*diff*
Stagnant Files	HH, GH, M $(B(F) + 3 * B(F)/u + D + U)$	NL	NL, *diff*
Shrinking File	HH, GH, M $(B(F) + 3 * B(F)/u)$	NL	M, HH, GH, NL, *diff*
Similar Dump	W $(2 * B(F)/u + B(F))$	W, NL	NL, *diff*

HH - Hybrid Hash Join, GH - Grace Hash Join, M - Merge Join, NL - Nested Loop Join , W - Window

Figure 8: Table 3 Comparison of *Snapshot Differential* Algorithms for Special Scenarios.

6 Applicability to Outerjoins

The *snapshot differential* algorithm can be considered to be an outer join (section 2.2). Thus, the algorithms that were discussed in section 3 can be modified to calculate the outer join of two relations. For instance we can use the merge join algorithm in section 3.3 to calculate the outer join of two relations R_1 and R_2, both with schema (K, B) and with K as the key. More specifically, if during the execution of the merge phase of the algorithm, the current tuples being scanned are: $< K_i, B_i >$ for relation R_1 and $< K_j, B_j >$ for relation R_2, the following output is produced for an outer join.

1. $< K_i, NULL >$ if $K_i < K_j$
2. $< K_j, NULL >$ if $K_j < K_i$
3. *Join Output Format* if *join condition* is satisfied. The *Join Output Format* is specified in the *select* clause in the case of SQL.

A similar modification to the output-logic can be made to the other approaches (hash join, nested loop join, UNIX *diff* based) so that the algorithms can be of use in calculating an outer join.

7 Conclusion

In this paper, we have defined the *snapshot differential* problem which arises when comparing database snapshots. This problem is encountered in data warehousing when there is a desire to incorporate legacy system data or periodic database dumps as information sources. We have shown that the *snapshot differential* problem is in essence an outer join (and is similar to text comparison). We have presented several solutions to this problem and compared them using our proposed metrics. We used the *number of IO* operations, the *number of messages* communicated, the *probability of error* and the *probability of error* as performance metrics. Based on these metrics, we conclude that the merge join based and hash join based (both grace and hybrid) algorithms are all promising. We then used compression to optimize these algorithms and observed that as much as 25 % improvement in *IO* cost can be obtained with a mere compression factor of 10 for the merge and hash join based algorithms. Several realistic scenarios were also identified (*growing file*, *stagnant file*, *shrinking file* and *ordered dump* scenarios). The join algorithms were optimized even further for these scenarios. With a compression factor of 10, the *IO* cost can be reduced by as much as 75 %. We also proposed a *window* algorithm to solve the *snapshot differential* problem for *ordered dump* scenario. This algorithm can attain a 75 % improvement in *IO* cost without requiring intermediate files. Morever, even without compression, the *window* algorithm still achieves a 50 % improvement in *IO* cost. The disadvantage of using compression is that the algorithms become *probabilistic* in the general scenario.

In future work, we will further investigate the effects of the input parameters (the number of modifications, the types of modifications and the measure of how out of order a file is) on the merge join , hash join and *LCS* based algorithms. Once a concrete understanding of the relationships of the parameters and the performance of the algorithms is established, good decisions as to which algorithm to use can be made on-line in a data warehousing environment. The *window* algorithm is promising but a clear understanding of the effects of the *divergence* of the snapshots has yet to be established. We intend to investigate this topic as well.