# Adaptive Hashing

# with

# Signatures

*Technical Report*

Eric A. Schweitz
Department of Electrical and Computer Engineering

Alan L. Tharp
Computer Science Department
email: alan_tharp@ncsu.edu
(919) 515-7435

North Carolina State University
Raleigh, North Carolina 27695-8206 USA

## *Abstract*

Adaptive hashing with signatures combines the adaptive hashing file structure together with superimposed signatures and several new algorithms to produce a new order-preserving data structure. This new technique has excellent direct retrieval performance, localized index organizations, and improved file index balance. In keeping with the principle advantage of the original adaptive hashing technique, algorithms to improve both primary and secondary memory storage utilization are also discussed. Furthermore, the new data structure has a high degree of flexibility, allowing it to be tailored for the optimum performance versus storage utilization ratio for a given application.

## *Section 1: Introduction*

Several papers in the literature combine the idea of record signatures with dynamic file hashing methods to improve the performance of these data structures. In [1], Larson describes a technique using signatures and separators with linear hashing which guarantees retrieval of a record in a single probe. The use of signatures and separators is described in [2] and [3] and is referred to as signature hashing in [4]. A similar idea using signature hashing with dynamic hashing is presented in both [5] and [6]. Work in the general area of using signatures to improve retrieval performance continues, as in [7] which discusses an optimum distribution of signatures for signature hashing methods in general.

Hsiao and Tharp's original adaptive hashing data structure and algorithms are presented in [8]. Adaptive hashing was introduced as an alternative data structure to order-preserving linear hashing (OPLH) [9] and the B+-tree [10]. A comparison of the advantages and disadvantages of adaptive hashing, bounded disorder, and B+-trees can be found in [11]. All of these data structures share the property that both sequential and direct access of data records can be performed in an efficient manner. Adaptive hashing's main advantage over the B+-tree is that it requires less primary memory for storing its index.  Other advantages include generally faster insertion time, less frequent reorganizations, and easier implementation.

This paper presents the new adaptive hashing with signatures (AHWS) method which will address some of the deficiencies of and will improve upon the original adaptive hashing technique. The main objective of this new approach is to improve the average number of retrieval probes over that of the adaptive hashing algorithm, primarily through the use of superimposed signatures.  In addition, AHWS will present improvements in the areas of file reorganization, file balancing, and overall storage utilization compared to that of adaptive hashing.

First, adaptive hashing and the superimposed signatures will be reviewed.  Adaptive hashing with signatures will be described in the third section.   A comparison of the experimental results of the two methods will be presented in the fourth section.  Finally, conclusions and areas for further research drawn from the results of this study will be given.

## *Section 2:  Review of Adaptive Hashing and Signatures*

Adaptive Hashing:  Adaptive hashing uses a two level dynamic hash table to access records in the file.  The hash table is stored in primary memory and consists of a number of buckets.  Records are hashed to the hash table by the function $H_d(key) = \frac{key}{2^{m-d}}$ , where *m* is the number of bits in the key and *d* is the number of reorganizations of the hash table that have been performed.  Because this hashing function starts hashing keys from their most significant bits down to bits of less significance, the keys are hashed to their respective buckets in ascending order.  For example, if key X hashes to bucket 12 and key Y hashes to bucket 13, it necessarily follows that X is less than Y. Therefore, one can process the keys in the file in sequential order by starting at the first page and following the page links until NULL is reached. (When NULL is reached, there are no more pages in the file.)  Care must be taken to maintain the sequential order of the records on a given page, and this is done during the insertion process.

Each bucket contains both a bucket pointer and a data page pointer. The bucket pointer (BP) points to the appropriate bucket which in turn holds the first data page pointer (DP) which points to a page containing records that hashed to the initial bucket location. For example, a record X may hash to the bucket 12. Bucket 12 has a bucket pointer to bucket 7. Therefore, the data page pointer in bucket 7 must point to the first page in the file that may contain records that hashed to location 12. Hence, if one knows record X is in the file, it must reside on the page pointed at by the data page pointer in bucket 7 or on some subsequent page. Figure 1 shows the adaptive hashing data structure. The hash table resides in internal (primary) memory; whereas, the records and page links are stored as pages of the file in external (secondary) memory.

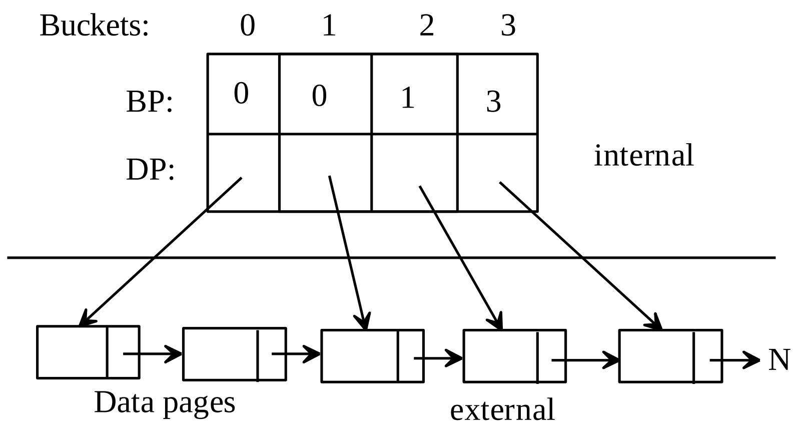

Figure 1: Adaptive Hashing Data Structure

As can be seen from Figure 1, the bucket pointers provide a level of indirection between the bucket location a key hashes to and the data page pointers. This level of indirection allows adaptive hashing to adapt to the distribution of the input data by utilizing more data page pointers for a particular bucket location. By using more data page pointers for the hash table locations with the most records and fewer data page pointers for those hash table locations with the least records, adaptive hashing is able to perform better than OPLH.

Another advantage over OPLH is that adaptive hashing has stable, predictable, and controllable performance because of the manner in which hash table reorganizations are triggered. Rather than splitting when the overall file utilization level exceeds a certain point or when a bucket becomes full, adaptive hashing reorganizes when the ratio of data pages to data page pointers exceeds a predetermined limit. This limit is called the ANDPP value. (ANDPP stands for a͟verage n͟umber of data pages pointed to by each d͟ata p͟age p͟ointer.)

Despite the strengths of adaptive hashing, the method does have some shortcomings. The weakness that is the focus of this paper is the fact that very large chains of pages can develop in an adaptive hashing file prior to a reorganization. These chains must be accessed sequentially until the next reorganization is performed. As can be seen in Figure 2, retrieving a record from the fifth page would take at least five probes. In general, an adaptive hashing hash table has $2^n$ entries, where $n$ is the number of reorganizations performed. The file will contain (ANDPP * $2^n$) + 1 pages when the ($n$ + 1)-th reorganization takes place. At this point, the file will not undergo another reorganization until another ANDPP * $2^n$ pages are added to the file. The weakness of the adaptive hashing method lies in the fact that all ANDPP * $2^n$ new pages may fall on the same chain. Thus all but one data page pointer could point to exactly one data page, and the remaining data page pointer may point to a chain of length ANDPP * $2^n$ pages, approximately. Such a file would be very unbalanced, with respect to the lengths of the page chains, since up to half the pages in the file would be accessible via a single bucket and no less than half the pages would be evenly distributed over the other $2^n$-1 (for $n \geq 1$) buckets. Interestingly, a file may become out of balance by merely inserting the records into the adaptive hashing file in sorted order. (Figure 2 would result from inserting the keys in descending order.) In files suffering from this imbalance, performance for the average retrieval will be extremely poor.

Another weakness in adaptive hashing is the reorganization procedure itself. When the ANDPP value is exceeded the bucket values are essentially discarded and a new file index is rebuilt from scratch. To rebuild this index, the reorganization procedure must follow the linked list of data

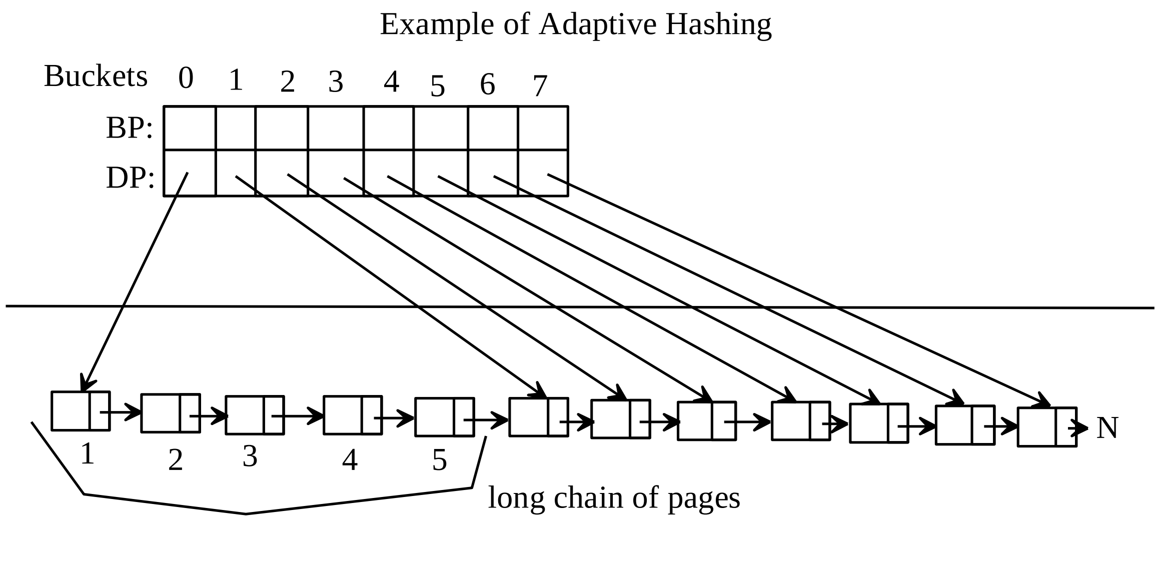

Figure 2: Long Chain of Pages

pages in secondary memory and access almost all the pages of the file. For large files, a large amount of time can be spent retrieving data pages during a reorganization operation.

Superimposed Signatures: A record signature is a bit string that represents the actual key value [4]. This bit string has a pre-determined length and a pre-determined number of set bits (ie. bits with a value of 1). The pattern of set and clear bits for a signature is psuedo-randomly generated for a particular key value. Thus, while the exact signature is somewhat arbitrary, it is also reproducible given the same key value. Therefore, a record signature represents a primary key value compactly. A page signature is formed by superimposing, by logical OR'ing, the record signatures of all the records on that page.

If one knows the page signatures of all the pages in a file, one can easily determine if a particular record might or might not reside on a particular page by comparing the sought record's signature to the signature of each page. If the page signature contains 1-bits in every location that the record signature does, then that page may (or may not) hold that record. One must check the actual contents of the page to be certain that the correct page has been isolated since this match may in fact be a false drop. A false drop could result if the records on a page set the same bits as are in the desired record's signature. Thus, even though the sought record is not present on the page, the bits in the page signature falsely indicate that the record is present. An access must therefore be made to determine if the information gained by comparing the signatures led to the correct page or not. Conversely, if the page signature does not have the same 1-bits as does the record signature, one knows for sure that the page can not contain the desired record and that page need not be accessed.

Thus, signatures provide a relatively easy way to reduce the search space when looking to retrieve a particular record from a file. By eliminating unnecessary accesses to the file through the comparison of a record's signature to the page signatures, AHWS improves the retrieval performance of adaptive hashing.

## ***Section 3: Adaptive Hashing With Signatures***

The AHWS data structure is shown in Figure 3. This data structure consists of four logical levels. The first three of these levels are stored in primary (internal) memory and will be referred to as the file index. At the first level of the file index are the bucket pointers (BP) into which keys are initially hashed using the same hash function as in adaptive hashing ($H_d$ (*key*) = *key* / $2^{m-d}$). The values stored in the bucket pointers are index values into the chain pointer table (CPT). An entry in the chain pointer table, called a chain pointer, points to a chain structure, or more simply a “chain”. A chain consists of an index value, called the chain’s origin, and a table of page descriptors of pre-determined size. A page descriptor is a pair of values ($\mathcal{L}$, $\mathcal{S}$) where $\mathcal{L}$ is a pointer to the page in the file and $\mathcal{S}$ is the page’s signature. A page consists of a pre-defined number of records and a pointer to the next logical page in the file.

The AHWS data structure has several new features which improve the overall performance over that of adaptive hashing. The addition of page signatures and chain origins improve retrieval performance markedly. Retrievals benefit from the expandable chain pointer table as well. Furthermore, AHWS's new approach to file reorganization improves the methods performance during insertions.

Signatures: Signatures are the primary means by which AHWS improves retrieval performance. There are many aspects that affect the rate of false drops when using signatures and thus affect the retrieval performance of AHWS. These aspects include the signature length, the number of set bits per record signature, the number of records per data page, the average number of records per page (page utilization), the length of a chain, and the average number of pages per chain (chain utilization).

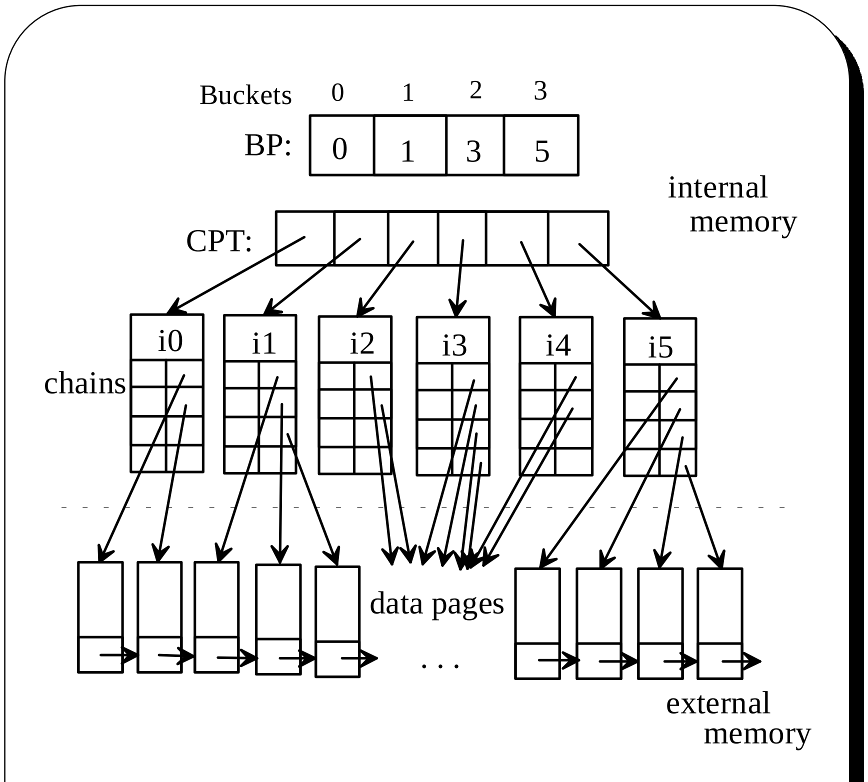

Figure 3: AHWS General Data Structure

If we assume that the probability of setting a given bit in a record signature is evenly distributed then there is a $\frac{1}{w}$ chance that a particular bit is set, where $w$ is the width of the signature. Thus the chance that a particular bit is not set is $\left\lceil 1-\frac{1}{w} \right\rceil$. If $r$ is the number of bits set in a record's signature, then the probability that a bit is not set for a record's signature is $\left[1-\left(\frac{1}{w}\right)\right]^{r}$. If $u$ is the number of record's on a particular page, then $\left[1-\left(\frac{1}{w}\right)\right]^{ru}$ is the probability that a bit is not set for a particular page signature. We let this probability be called $P_{pg,unset}$. The probability that a bit is set for a particular page signature is therefore $1-P_{pg,unset}$. The probability that all $r$ bits of a signature are set is therefore $(1 - P_{pg,unset})^r$ . Thus, the probability that a page signature has the same bits set as the sought key's signature is given as,

$$\left(1 - \left[1 - \left(\tfrac{1}{w}\right)\right]^{ru}\right)^{r} \quad \text{[Equation 1].}$$

To determine the probability of a false drop, we assume that the sought key is in the file. The key is on the last page in the last chain searched, by definition. Thus, the probability that the page being compared is not the one with the desired key is $1 - \frac{1}{N}$ , where $N$ is the total number of pages searched. The probability of a false drop is therefore,

$$\left(1 - \tfrac{1}{N}\right)\left(1 - \left[1 - \left(\tfrac{1}{w}\right)\right]^{ru}\right)^{r} \quad \text{[Equation 2].}$$

Furthermore, $N$ is a function of the number of chains searched and the lengths of those chains. When chain origins are used, $N$ can be approximated as product of the length of a chain and the average chain utilization. Similarly, $u$ can be approximated as the product of the page size and the average page utilization. Equation 1 can be used for worst case analysis and equation 2 can be used for an optimistic case inspection. Together they can be used to tune the various parameters of AHWS for the desired performance level.

Chain origins: A chain origin is simply a copy, in whole or in part, of the first key on the first page of the chain. The benefit of chain origins is twofold. First, the origin can be used during reorganization so that no, or few, accesses to secondary memory need be made. (See Localized Re-organization.) Secondly, the origin can be used to further improve retrieval performance. (See Re-trievals, for more information.)

There are four types of origins which can be employed in AHWS. The first type is that of the “full” origin. A full origin is a complete copy of the first key on the first page of the chain. For example, if one was hashing 32-bit integers the full origin would consist of a 32-bit integer. The benefits of using full origins include very fast retrievals and reorganizations. In fact, no accesses to external memory are necessary to reorganize the hash table of a file with full origins.

The “prefix” origin is very similar to the full origin. The prefix origin consists of the entire hashed portion of the primary key. For example if the keys are strings and the hash function, $H_d(key) = key / 2^{32-d}$, is used to hash the first 4 bytes of the string, then the prefix origin would consist of the first four bytes of the string. As with the full origin, no accesses to secondary memory are necessary to perform a reorganization and retrieval performance is improved. The disadvantage for both full and prefix origins is that they require additional space in primary memory.

The third type is that of the “partial” origin. A partial origin consists of only a piece of the “hashable” key. The advantage of using a partial origin is that it provides better retrieval performance than does no origin. The primary disadvantages are that partial origin algorithms are more complicated than either full or prefix origins, and that their usage may require some accesses to secondary storage during a reorganization operation. Partial origins will be discussed in detail along with other methods to reduce the size of the file index below.

Lastly, the degenerative case is that of “null” origins, or no origin at all. A null origin has the principal advantage of requiring no storage overhead in primary memory. However with null origins, both file reorganizations and retrieval performance are degraded compared to methods which use some origin.

Localized Reorganization: AHWS’s method of localized reorganization corrects both the problem of costly reorganization operations as well as the unbalanced file problem. Instead of using an ANDPP value to determine when the file needs to be reorganized, the file reorganizes itself at the time and place it is necessary to do so as records are inserted in the file. The goal of a reorganization is to distribute the number of pages accessible through a given chain pointer evenly. In the case of adaptive hashing, the pages are distributed evenly to each of the data page pointers during each global reorganization. After such a reorganization, the file index is again allowed to become unbalanced.

Adaptive hashing keeps the number of data page pointers fixed with respect to the size of the hash table. By removing this artificial constraint, AHWS can create new chain pointers when and where they are needed. This is accomplished by specifying an upper limit (*chain_size*) to the number of pages in a chain. A chain is said to be full when it has the maximum number of pages allocated to it. When the insertion of a new record requires that a page be split on a chain which is full, the chain is divided into two chains. The original chain and the new chain are each allocated one half of the *chain_size* + 1 pages. (The original chain consisted of *chain_size* pages and the page that was split created one additional page.) A new chain pointer is then added immediately after the original chain's location in the CPT. Chain pointers which succeeded the original chain in the table are simply moved over one position to make room for the new chain pointer. Moving these successor chain pointers means that the hash table's BP values must be incremented if they are pointing to one of the shifted chain pointers. BPs which point to chain pointers preceding the split chain do not need to be changed since this part of the table has not been altered.

When the BP value points to the CPT entry of the chain being split, some additional work needs to be done since it can not be determined offhand whether such BP values must be incremented or not. For example, the chain to be split may contain keys that have all hashed to a particular bucket in the hash table, as shown in Figure 4. Thus when the chain is split, the BP value must not be incremented since those records in the first new chain would be irretrievable.

However, the chain being split may in fact point to pages which belong in several of the primary hash table's logical buckets, as depicted in Figure 5. When such a chain is split, the keys that hash to a particular bucket may reside either on the first new chain or the second new chain or even both new chains concurrently. It is important to remember that the keys are ordered such that if key *K* is known to hash to bucket *B* then any keys which follow *K* in the file must hash to bucket *B* or some subsequent bucket. So that all keys will continue to be retrievable, it must be discovered which keys in the old chain have moved to the second new chain and which keys have stayed in the first new chain. This is done by hashing the first key on the first page of the second new chain. For

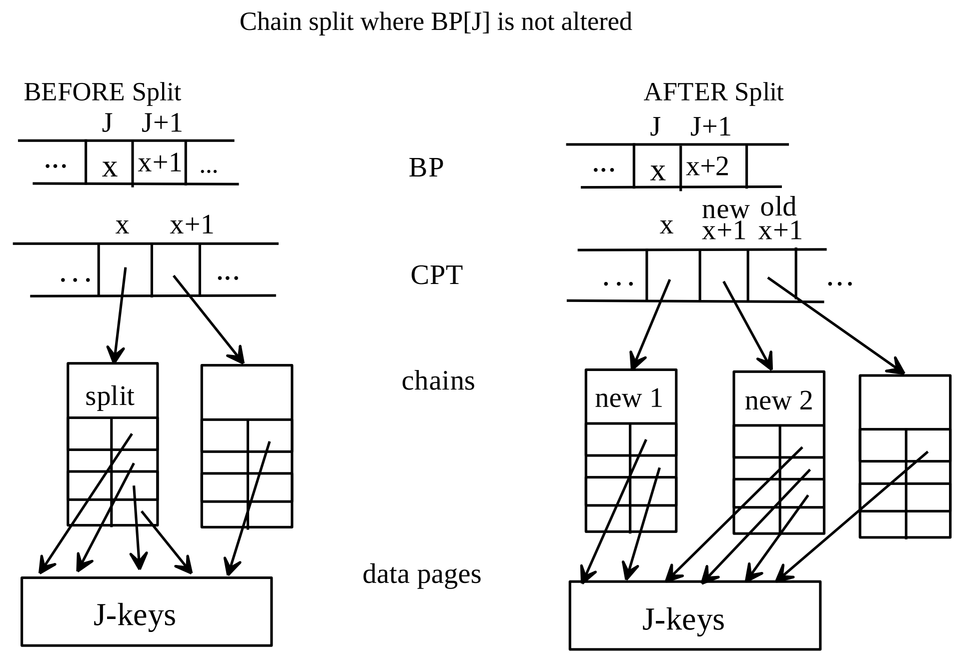

Figure 4: Chain with keys belonging to 1 bucket being split.

those buckets, pointing at the original chain which was split, with a location greater than this computed hash value, the bucket pointers must be incremented to point to the new chain since all the keys (if they exist in the file) hashing to such a bucket must reside on the newly created chain. For logical buckets pointing to the divided chain with a location less than or equal to the computed hash value, no change to the bucket pointer need be made since keys hashing to these hash table locations reside on the first new chain or the beginning of the second new chain. The buckets greater than the hash value do indeed need to be incremented for if these values remained constant the retrieval algorithm, which uses both the bucket pointer of the bucket the desired key hashed to and the next logical bucket to determine the chains to be searched, would not be able to retrieve all the keys from the file.

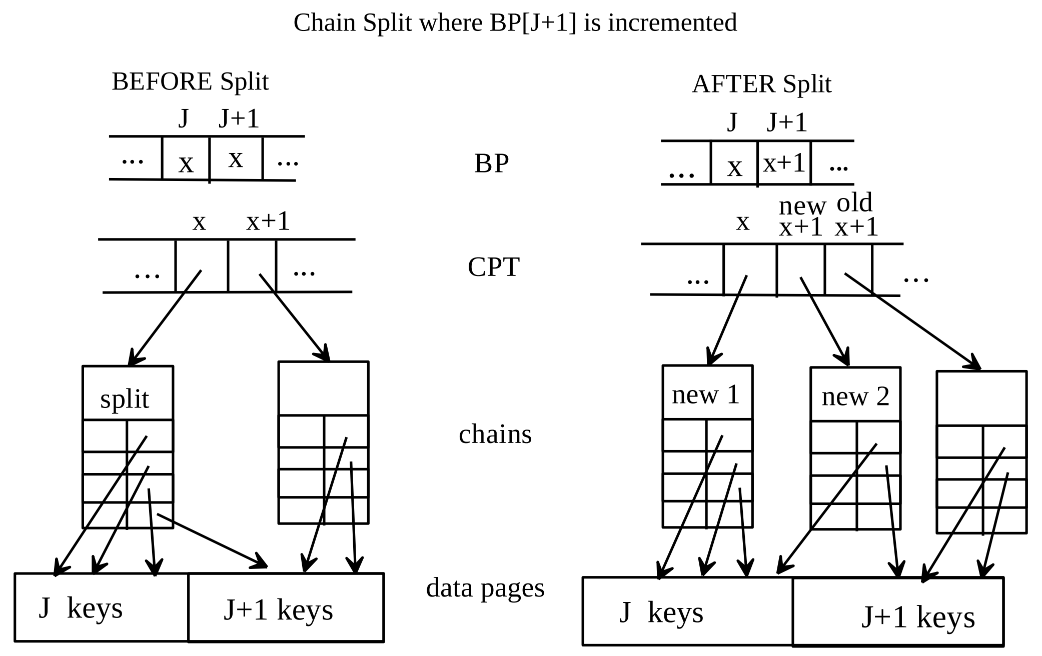

Figure 5: Chain with keys belonging to 2 buckets being split.

The last aspect of file index reorganization to be discussed is the manipulation of the hash function and the bucket pointer table. The hash function level is incremented when the number of chain pointers exceeds a specified threshold value. The threshold is defined to be a multiple of the hash table size. For example, for a threshold value of two, if when the hash function level is incremented then the CPT size must be greater than 2 * the hash table size. When this happens the hash table is expanded to twice its former size and the BP indices must be recomputed. To recompute the new BP values the chain origins (either full or prefix), already in memory, are hashed using the new hash function and the BP table is refilled with these new values. Thus no disk accesses are necessary to recompute the values of the BP array. Remember that the chain origins are maintained during the normal insertion process. (Psuedo-code for the insertion/reorganization process can be found in Appendix B.)

Retrievals: Chain origins are also useful for retrievals from the file. Full origins can be used to determine the exact chain which must be searched to find a record. Prefix and partial origins can be used to determine a subset of possible chains that the record may be on. If the key's value is greater than chain $i$'s full origin and less than chain $i + 1$'s origin then the record must lie on a page in the $i$-th chain. To determine the exact page, the key's signature is compared to the page signatures for the page's on the $i$-th chain. A binary search algorithm, similar to the one proposed in [8], using chain origins rather than data read from pages in the file can be employed as well.

The reason that prefix and partial origins can not isolate the exact page is that they do not contain enough information. If the sought key's prefix matches the prefix origin then the key

may either reside on that chain or the previous chain. Therefore both chains must be searched for a matching signature. Since more than one chain can potentially be searched, this may have a negative effect on performance since the chance for false drops increases with the more pages that are searched. (See Signatures, Eqn. 2.)

Since the improvements in performance discussed earlier are achieved by using more primary memory, methods for minimizing this new overhead were considered. It should be pointed out that the sizes of many of the aspects of the data structure are fully under the control of the programmer. This allows the programmer to tune the method to produce the best file index size to performance ratio for the given application. Some algorithms to reduce the overall size of the file index at the expense of some retrieval performance include partial origins, smaller page signatures, and greater chain and storage utilization. These algorithms are discussed below.

Partial Origins: The first method for reducing the size of the index is to use a partial origin value. Partial origins are formed by a new hashing function, $po_i\ (key) = (key \bmod 2^{m-d})\ \text{div}\ 2^{m-d-ow}$, where $m$ is the bit-width of the key, $d$ is the current hash level, and $ow$ is the partial origin bit-width. The experiment used partial origins of one byte, or a bit-width of 8.

Recall that the primary hashing function ($H_d$) hashed keys based on the quotient of the operation $key$ div $2^{m-d}$. Thus, this information has already been extracted from the key when it was initially hashed.  The new hash function, $po_i$, returns the first *ow* bits of the remainder after the initial hashing function has been applied.  Thus partial origins created in such a manner are simply the truncated remainder of the initial hashing operation.  (See Figure 6.)

Storing this truncated remainder is useful for both reorganization and retrieval, as mentioned before.  The value $po_i$ (*key*) can be used to determine a subset of chains which the record may be stored on.  For example, if $po_i$ (*key*) returns the value 5 for a given key then only the chains with partial origins of 5, as well as the chain immediately before the first chain with a partial origin of 5, need to be searched for the record.  (The chain just before the first chain with a partial origin of 5 needs to be searched since keys with $po_i$ (*key*) = 5 may exist at the end of the chain.)

Partial origins are also useful during reorganization since they can be used to determine new bucket pointer values without having to access the file.  When the hashing function level is incremented, keys hashing to a bucket *V* with the previous hash function are subsequently hashed to the buckets 2*V* and 2*V*+1, for all $0 \leq V \leq$ (old hash table size - 1).  Since the chain pointer table is not being modified, the values of the original hash table's bucket pointers can be copied to the even numbered bucket pointers in the new table without modification.  (ie. old-BP[*V*] $\rightarrow$ new-BP[2*V*].)  This follows from the fact that if *key* div $2^x = V$ then *key* div $2^{x-1}$ is at least 2*V*.  The partial origin can then be used to determine the BP values for the odd numbered buckets (new-BP[2*V*+1]).  Since the partial origin is actually the truncated remainder of the original hash function, any chain which has a remainder $\geq 0.5$ should be placed in the odd bucket.  In other terms, BP[2*V*+1] should point to the chain immediately before the first chain with a partial chain origin with its most-siginificant-bit set to 1 between the chains pointed at by BP[2*V*] and BP[2*V*+2].

Unfortunately, there are a few special cases that must be addressed.  For instance, if BP[2*V*] equals BP[2*V*+2] then there are no chains between these two buckets and any key hashing to BP[2*V*+1] must also lie on this chain.  One can not always rely on the presence of a partial origin for a chain between BP[2*V*] and BP[2*V*+2] that meets the condition of having its most-significant-bit set to 1. In this case, a file access is necessary to retrieve the first page of the chain pointed to by BP[2*V*+2]. The first key on this page is then hashed. If the key hashes to bucket BP[2*V*+1] or greater then bucket BP[2*V*+1] is set to point to the chain immediately before the one pointed to by BP[2*V*+2]. Otherwise, BP[2*V*+1] is set equal to BP[2*V*+2].

Because the hash function level changes during such reorganization operations, the partial origin hashing function is also altered. It might first appear that a file access is necessary to recalculate the new chain origins. However, it is possible to delay such an access. If the partial origin hashing function is thought of as a sliding window mask along the key, then it becomes apparent that when the hash function level is increased the window is shifted over one. Thus the most significant bit in the window is shifted out and a new unknown bit is shifted in the least significant position. This shift can be easily replicated by shifting the partial origin one place to the left. The obvious issue is whether a zero or a one bit should be shifted into the new origin value. To help answer this question, it should be made clear that the retrieval operation assumes that the actual value of the first key on the first page in a chain must be at least as big as the origin value, for if the origin value where in fact less than the actual value the entire chain would be skipped rather than be searched for the desired key. Thus if the origin value is less than the actual value, keys will be irretrievable. By always shifting in a one bit into the partial origin value during a reorganization, it is guaranteed that the origin value will always be greater than or equal to the first key's actual partial origin value.

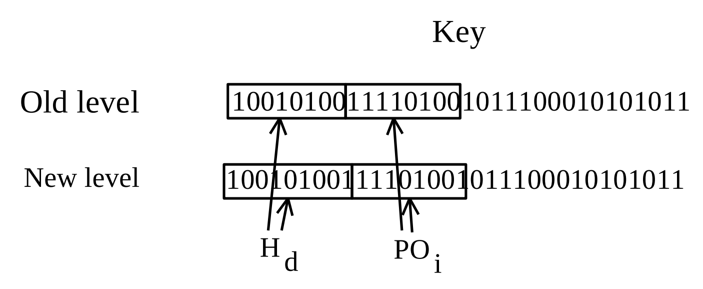

Figure 6: Partial Origins

Clearly, one can not shift in one bits indefinitely; otherwise, the origin ceases to have utility in narrowing the search space since all the origins would become identical. To prevent this indefinite "guessing" at the partial origin's new value, the origin should be tested to see if it consists of all one bits. Should the origin be all ones, the origin is assumed to be in need of calibration. This calibration of the partial origin consists of a accessing the file to retrieve the first page on the chain. The first key on the page is then used to recalculate the origin. Fortunately, this calibration operation may be avoided indefinitely since each time the first page in a chain is accessed the origin is updated.

Smaller Signatures: The second method proposed for reducing the size of the file index in primary memory was to use smaller signatures. Since each page descriptor in a chain contains a page signature, it would appear that reducing the size of the signatures would save memory. For example, a reduction from 32-bit to 16-bit signatures would save two bytes for every page descriptor in the file index. For a file with many pages, one would expect a great deal of savings in storage. The negative side of reducing the size of the signature is that the rate of false drops increases as the signature width is decreased. (See Signatures, Eqn. 1.)

Ripple Insertion: When attempting to add a record to a full page, the original adaptive hashing insertion algorithm simply creates a new page and stores the high key on this new page. This algorithm then relies on chance to add records to the new page. If no more keys hash to the new page, no new records will be stored on the page and the page will be underutilized. An even worse ramification of this insertion algorithm is when repeated attempts are made to insert different keys on the same full page. With each new attempt a new page is created to hold the overflow record. When this situation occurs, overall file storage utilization can become extremely low. (In the worst case, a file could contain a single full page and many pages with only a single record stored in each.)

The ripple insertion algorithm addresses this problem, by delaying the creation of new pages. When an attempt is made to insert a key on a full page, the largest key overflows the page and is thus removed from that page. If altered, the page is then written to secondary storage. An attempt is then made to insert the overflowed key onto the next page in the chain. If there are no more pages in the chain, but the chain is not full, a new page is appended to the end of the chain. However, if the chain is full, then the chain must be split and a local reorganization ensues. Thus the insertion of keys causes a ripple effect as each full page overflows its largest key into the next page in the chain. This ripple effect is tightly bounded by the length of the chain; therefore, only the pages on a single chain are influenced by the ripple insertion process.

## *Section 4: Test Results*

This investigation compared the performance of the original adaptive hashing technique with that of several variations of AHWS. AHWS Version 0 (AHWS V0) was an initial attempt at determining the affect that using superimposed signatures with the adaptive hashing file structure. In AHWS V0, the DP was modified to point to a linked list of 32-bit signatures in internal memory. The same global reorganization and insertion methods as adaptive hashing were used. AHWS Version 1 (AHWS V1) implements the structure of Figure 3. That is, Version 1 uses 32-bit signatures, localized reorganization, and full chain origins. AHWS Version 2 (AHWS V2) implements the various file index reducing methods as well as those methods used by AHWS V1

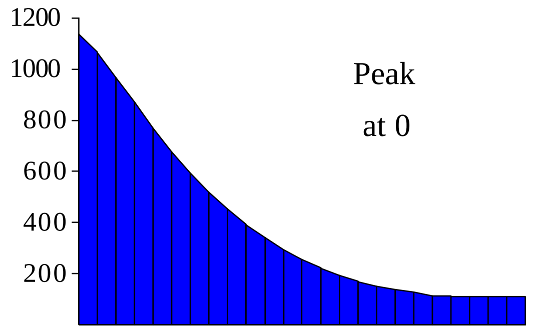

Figure 7: Distribution 1

for performance enhancement. Thus AHWS V2 includes 16-bit signatures, localized reorganization, partial chain origins, and ripple insertion.

Each of these methods were tested using four data distributions. These distributions were peak at zero, even distribution, peak at 0 and Maxint, and peak at Maxint / 2, and are shown in Figures 7 through 10. Each distribution consisted of five sets of 10,000 keys each. The keys themselves were random integers ranging from 0 to $2^{32}$-1 (31 bits of significance). The tests were performed on an IBM RS/6000 machine running the AIX 3.2 operating system. Figure 11 and 12 show distributions 5 and 6, respectively. Distribution 5 consisted of one file of 100,000 random 31-bit keys. Distribution 6 consisted of one file of 10,000 sorted (ascending) 31-bit keys. As can be seen in the graph, these keys were tightly clustered in a single region of the overall range of possible values. These two distributions were used to show both the effects of using adaptive hashing and AHWS with a very large data set and to determine the performance of the methods under the worst case situation for the original adaptive hashing insertion technique.

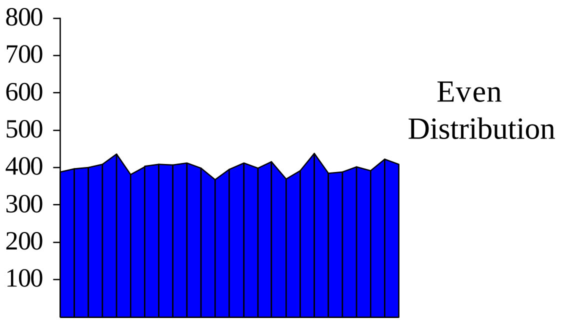

Figure 8: Distribution 2

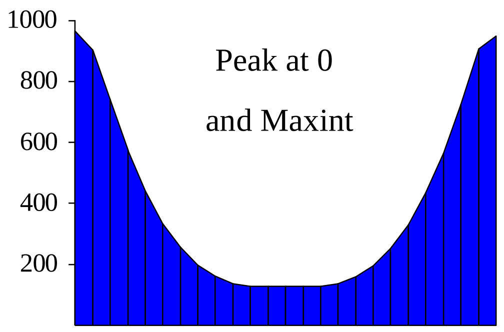

Figure 9: Distribution 3

Using Equation 1 (see Signatures), the minimum probability of a page having matching one bits with a given signature, given a page size of 4 and a signature width of 32 bits, was found to be when the number of bits set per record was 5. The best case for a signature of width 16 bits and a page size of four was calculated to be 3 bits.

For the first set of tests, the ANDPP value of the adaptive hashing and AHWS Version 0 programs was set to the minimum value of 2 (AH.2 and AHWS V0.2, respectively). AHWS Version 1 and 2 were each run with a chain size of 5 and a hash level threshold of 2. The methods were compared on the following criteria: average number of probes per retrieval, amount of internal memory used, external memory utilization, and number of accesses per complete file build. With a chain size of five, AHWS V1 and AHWS V2 have a much higher effective ANDPP value than that used for the AH.2 and AHWS V0.2 programs. Taking the number of pages in the AHWS file and dividing by the hash table size, one can determine the average number of data pages per hash bucket. The experimental results yield an effective ANDPP value, using the calculation above, of approximately 9. So that a fair comparisons could be made, additional experiments were performed for both the adaptive hashing program (AH.8) and AHWS Version 0 (AHWS V0.8) with the ANDPP value set to 8. (Eight being the closest power of two to nine.) The results of these trials are shown in Tables 1 through 4.

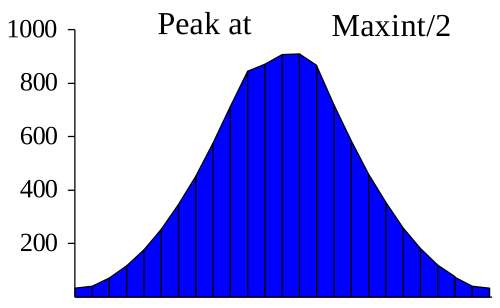

Figure 10: Distribution 4

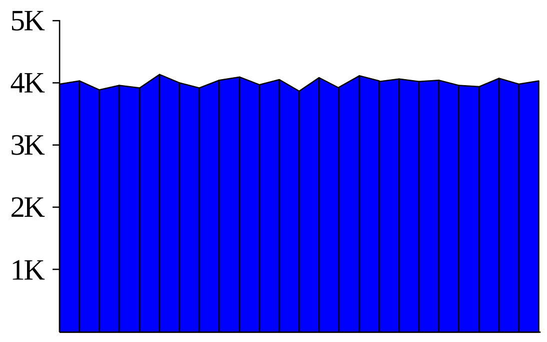

Figure 11: Distribution 5

Table 1 shows that AHWS outperforms the original adaptive hashing method in every case with respect to the average number of probes per retrieval. Retrieval performance was increased from 5.95 to 1.01 through 1.26 average probes per retrieval (79 - 83%), for all cases with an effective ANDPP value of 8. Even when comparing the various versions of AHWS with AH.2, which had an average probes per retrieval of 3.00, AHWS performed 58 - 66% better. (All data values can be found in Appendix A.)

Using Equation 1 and the data gathered from the experiment, the theoretical rates for false drops were calculated to determine how well the actual results followed the predicted results. The theoretical results are shown in Table 5. Comparing these tables indicates that the actual performance of the signature methods was not quite as good as that predicted theoretically. The actual results differed from those that were predicted by < 1% to 10%. Thus it would appear that although the experimental results did not follow those from the analysis, Equation 1 and 2 are reasonably good for predicting of the relative performance of the method given various parameter values.

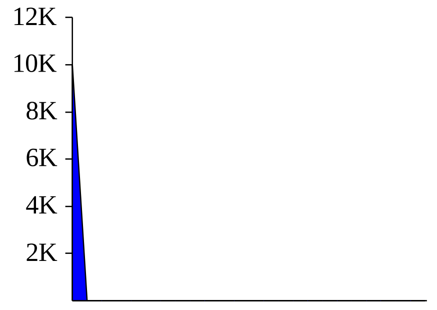

Figure 12: Distribution 6

This dramatic increase in performance may at first seem to be at the expense of much more additional primary memory. However, looking at Table 2 offers some indication that this is not necessarily true. While AH.8 uses by far and away the least amount of primary memory, it has very poor performance, comparatively. The method that generated the largest sized file index was AHWS V0.2 which did not use either localized reorganization or ripple insertion. Interestingly, AHWS V2 uses less primary memory than AH.2 by over 2,000 bytes.

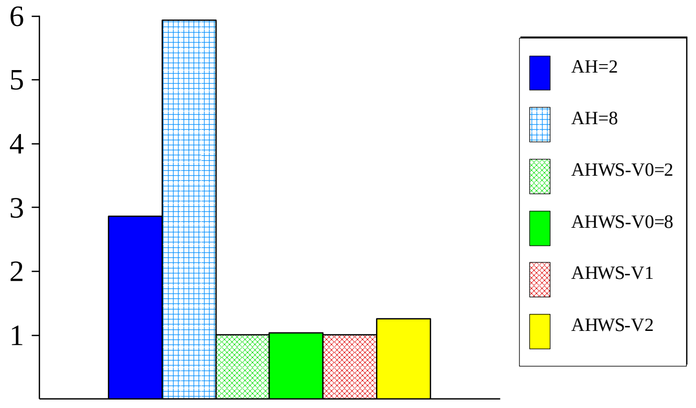

Table 1: Avg. Retrieval Probes

Furthermore, AHWS V2, with ripple insertion, had 36 - 38% better storage utilization than did the other methods. (See Table 3.) This poses the question of whether the savings in the file index size were the result of the reduction in the size of the signatures or the effect of the ripple insertion algorithm.

Therefore, another experiment was performed using the AHWS V2 program with 32-bit signatures with 5-bits per record (AHWS V2.32). This experiment indicated that the file index increased 19% from that of AHWS V2 with 16-bit signatures, and a 11% increase over AH.2. However, the performance of using 32-bit signatures increased 17% and 65% over AHWS V2 and AH.2, respectively. It therefore appears that using the ripple insertion algorithm has a greater overall effect in reducing the file index size than did reducing the signature size.

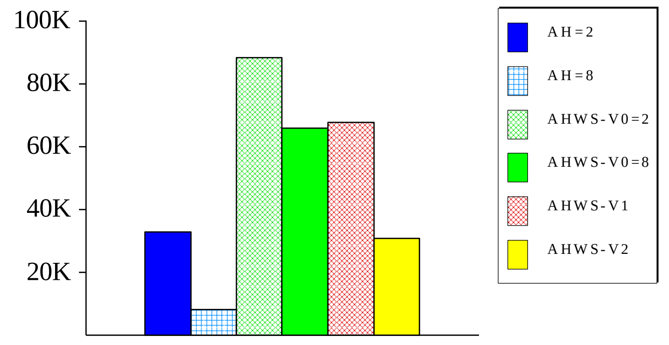

Table 2: File Index Size (bytes)

The final criteria for comparison of the methods was that of the number of accesses used to construct the data structure. (See Table 4.) The number of accesses varied dramatically from method to method. Since AHWS V1 used full chain origins, it was able to outperform all the other methods by 937 to 52,813 average accesses (2 - 122%). AHWS V2 had largest number of accesses which was to be expected since it was the only case which employed the ripple insertion technique.

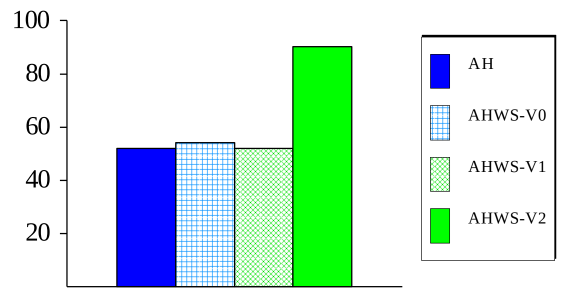

Table 3: Page Utilization (%)

This raises the issue of how much of a benefit is derived from using chain origins for reorganizations during construction. To answer this question, two additional experiments were performed. This time, both the AHWS V1 and AHWS V2 programs were modified to use null origins (AHWS V1.N and AHWS V2.N, respectively). The AHWS V1.N showed an increase in the average number of retrieval probes from 1.01 to 1.04 (2.5%) and an average increase in accesses to build the file of 2,044 probes (5%). However, the size of the file index was reduced 5,316 bytes (8%) on average. Similarly, the results of comparing AHWS V2.N to AHWS V2 show an increase in the average number of retrieval probes from 1.26 to 1.83 (45%), an average increase of 440 (<1%) accesses to build the file, and a decrease in the average size of the file index of 715 bytes (2%).

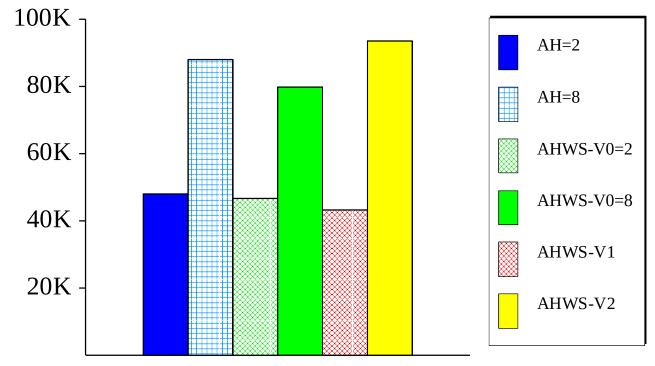

Table 4: Accesses to Build

To determine the effects of the chain size on the performance of AHWS, another experiment was performed with AHWS V2 with 32-bit signatures and a chain size of 10 (AHWS V2.32.10). This experiment showed an increase of 4.5% in page utilization, an increase of 4% in the average number of retrieval probes, an increase of 55% in the number of accesses to build the file, and a reduction in file index size of 10%. Overall chain utilization decreased 5% as well.

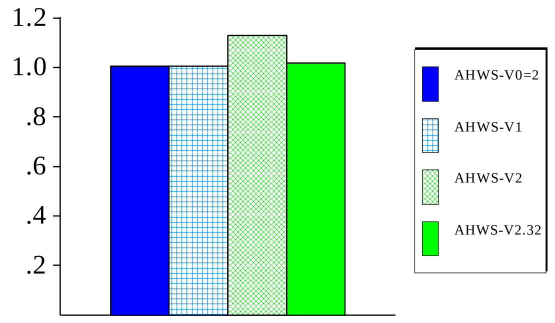

Table 5: Theoretical Avg. Probes

The last experiments performed were running adaptive hashing, AHWS V1, and AHWS V2 with the data files of distributions 5 and 6. The 100,000 key data set, distribution 5, showed few surprises as most of the results remained very predictable. The only significant change was for in the average number of retrieval probes for the adaptive hashing algorithm, which rose 18%. The rest of the results indicate fairly constant average probes, page utilization, and chain utilization, and an order in magnitude increase in the number of accesses for build and file index sizes, as was expected.

The results for distribution 6 were not nearly as callable. AHWS V1 performed extraordinarily well in this case, with nearly all figures of comparison improving as compared to its average performance on the other 4 distributions. The only areas were AHWS V1 did not perform better were in the number of read accesses made during file construction and overall chain utilization. Adaptive hashing was expected to perform poorly for distribution 6, and the results supported these expectations. The average retrieval probes increased by 819% to 54.71, and the number of accesses needed to build the file soared to well over 3 million accesses.

## *Section 5: Conclusion*

The results of these experiments have shown that the use of superimposed signatures, chain origins, localized reorganization, and ripple insertion can each dramatically improve various characteristics of adaptive hashing. Superimposed signatures improve retrieval performance by 79 - 83% for files with an effective ANDPP value of approximately eight. Using this new method of page signatures to reduce the search space, retrieval performance approaches single probe retrieval with an error of less than 5% for signatures with a width of 32-bits.

These experimental results do not appear to strongly suggest that the use of chain origins will lead to significant performance improvement in all cases. The data does show that chain origins will provide improvements in both retrieval performance and the number of accesses when the file is being constructed; however, both of these gains tend to be dominated by other characteristics of AHWS. When using signatures sized for a fairly small false drop rate, the use of chain origins for retrievals provides only minimal gains in performance. Yet when smaller signatures are being used and the false drop rate is relatively high, the chain origins affect retrieval performance rather markedly. Thus the utility of chain origins on retrieval performance appears to be dominated by the effects of signatures on retrievals.

The use of chain origins for reorganization reduced the number of total file accesses during its construction by 5%, and read accesses by 7%. However, this gain is rather insignificant when compared to either the decrease in accesses resulting from localized reorganization or the increase in file accesses caused by the ripple insertion algorithm. Perhaps the most interesting aspect of using chain origins was that retrieval performance remained essentially constant for any distribution; contrastingly, the retrieval performance of AHWS without any origins fluctuated for the various distributions.

The use of localized reorganization had a positive effect on both average number of probes per retrieval and average number of accesses per build. Although the increase in retrieval performance was less than 5%, the improvement in accesses per build when using the localized reorganization method was 76 - 85% depending on if chain origins were used or not. This decrease had a very noticeable effect on decreasing the overall run-time of the test programs.

The use of the ripple insertion algorithm resulted in a file which had a much higher page utilization and thus a much reduced file size in secondary storage. Since fewer pages were needed to store the information in the file, the corresponding file index structure was also very compact. This compacting of the file and its index resulted in more false drops, as was expected, but the retrieval performance was still 79% better than that of the original adaptive hashing algorithm.

Although ripple insertion could not compete with adaptive hashing with an ANDPP value of 8 based on file index size alone, it compares extremely well in general.  Using smaller signatures, partial chain origins, and the ripple insertion algorithm, AHWS produced both better retrieval performance and a smaller index than could adaptive hashing when tuned for best performance with a minimum ANDPP value of two.

It would seem appropriate to compare this new method, AHWS, with that of the currently popular B+-tree data structure.  In the optimum case when the entire B+-tree index can be stored in primary memory, the B+-tree guarantees retrievals in a single probe.  These experiments show that AHWS V2.32 is nearly as efficient for retrievals, with a false drop rate of only 4%.  Another criteria for comparison could be the page utilizations and the relative file index sizes.  In [12], B+-trees are said to typically have a page utilization of 69%.  B+-trees using algorithms to increase storage utilization achieve as high as 80%.  On the other hand, AHWS can easily achieve 90% storage utilization with ripple insertion for a 21% overall improvement.

The primary advantage of adaptive hashing as compared to other order-preserving data structures is that adaptive hashing does not require as much primary memory to store its file index [11].  Although, the B+-tree is rather complex to analyze, it is possible to determine a range of sizes for a typical B+-tree file index.  Working from the 69% storage utilization figure, a B+-tree of capacity order one uses at least 69,100 bytes and as many as 138,060 bytes for its index.  Again, comparing these to AHWS V2.32, AHWS uses 48 - 74% less primary memory for its file index than does this B+-tree.  Thus AHWS, like adaptive hashing before it, uses less primary memory to store its index than does a comparable B+-tree.  In other studies, such as [13], it has been shown that the B+-tree file index is not optimal in its use of primary memory.  This work suggests that AHWS is closer to the optimum than is the B+-tree.

Unfortunately, it is not possible to guarantee single probe retrievals with AHWS. While, the B+-tree is capable of retrievals in a single probe, this only happens in the best case where the entire file index can be stored in primary memory. Therefore, with very large data sets, it is more often the case that the B+-tree index must be at least partial stored in secondary memory, thus increasing the average number of probes needed for the retrieval of a key. However, research in the area of an order-preserving form of signature hashing, or another method of reducing the search space to a unique page, may provide a means to achieve the optimum goal of a single probe per retrieval for the AHWS method. Other areas where additional research may improve AHWS would include employing optimal signature extraction techniques [14] to further reduce the overall storage overhead. Algorithms which efficiently implement dynamic chain sizes could also improve storage utilization and possibly retrieval efficiency. The major disadvantage of AHWS is the time it takes to construct the file. Although improved greatly via localized reorganization, this improvement is lost by the ripple insertion technique. An improved algorithm which would minimize the number of accesses during construction, yet maintain the high page utilization, might prove beneficial. Finally, experiments in using a binary search type algorithm as in [8] with AHWS, rather than the linear search used in this work, should provide some increase in overall retrieval performance, since a binary search-like algorithm would have a complexity of $\mathbf{O}(\log_2 n)$ and the linear search has complexity $\mathbf{O}(n)$. This algorithm change would, in all likelihood, benefit the retrieval performance of structures using prefix or partial origins directly, since these structures do not certify bounded retrieval accesses.

This paper has presented a data structure which improves the retrieval performance of adaptive hashing and is competitive with the popular B+-tree with respect to retrievals. AHWS enjoys an advantage over B+-trees in that it can use both internal and external storage more efficiently. Furthermore, the new AHWS method appears easier to implement than those of the B+-tree and its derivatives.  The major disadvantage to the new method is that it requires more file accesses during the construction of the file than did its predecessor.  However, since most applications retrieve data much more frequently than they insert it, this extra work is not imposed on the routine usage of the AHWS data structure.

***References:***

[1] Larson, P. Linear hashing with separators - a dynamic hashing scheme using one probe. *ACM Trans. Database Syst.* **13**(3), 366-388, (Sep. 1988).

[2] Gonnet, G. and Larson, P. External hashing with limited internal storage, *Proc. ACM Symp. Principles of Database Syst.*, ACM, New York, 256-261, 1982.

[3] Larson, P. and Kajla, A. File organization - implementation of a method guaranteeing retrieval in one access. *CACM*, **27**(7), 670-677, (1984).

[4] Tharp, A. *File Organization and Processing*. John Wiley and Sons, New York, NY, 1988.

[5] Chung, Y. and Ramakrishna, M. Dynamic signature hashing. *Proc. 13th Int. Computer Softw. & Applications Conf.*, Orlando, FL, 257-262, (Sep. 1989).

[6] Cesarini, F. and Soda, G. A dynamic hash method with signature. *ACM Trans. Database Syst.* **16**(2), 309-337, (June 1991).

[7] Ramakrishna, M. and Ramos, E. Optimal distribution of signatures in signature hashing. *IEEE Trans. on Knowledge and Data Engr.*, **4**(1), 83-88, (Feb. 1992).

[8] Hsiao, Y. and Tharp, A. Adaptive Hashing. *Information Systems*. **13**(1), 111-127, (1988).

[9] Orenstein, J. A dynamic hash file for random and sequential accessing. *Proc. 9th Int. Conf. on Very Large Data Bases*, Florence, 132-141, (1983).

[10] Knuth, D. *Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[11] Tharp, A. and Boswell, W. B+ trees, bounded disorder and adaptive hashing, *Information Systems*, **16**(1), 65-71, (1991).

[12] Chu, J.-H. and Knott, G. An analyis of B-trees and their variants, *Information Systems*, **14**(5), 359-370, (1989).

[13] Garg, A. and Gotlieb, C. Order Preserving Key Transformations, *ACM Trans. on Database Syst.*, **11**(2), 213-234, (June 1986).

[14] Faloutsos, C. and Christodoulakis, S. Optimal signature extraction and information loss. *ACM Trans. on Database Syst.*, **12**(3), 395-428, (Sep. 1987).

**AH.2**

Adaptive Hashing with ANDPP = 2. 10,000 records. Page size 4.

| Distribution | Avg Retrieval Probes | Index Size (bytes) | Page Utilization | Write Accesses for Build | Read Accesses for Build |
|---|---|---|---|---|---|
| 1 | 3.09 | 32,768 | 52.1% | 14,799 | 37,237 |
| 2 | 2.79 | 32,768 | 52.0% | 14,805 | 31,893 |
| 3 | 3.04 | 32,768 | 51.7% | 14,831 | 36,346 |
| 4 | 3.08 | 32,768 | 52.0% | 14,803 | 36,519 |
| Avg | 3.00 | 32,768 | 52.0% | 14,810 | 35,499 |

**AH.8**

Adaptive Hashing with ANDPP = 8. 10,000 records. Page size 4.

| Distribution | Avg Retrieval Probes | Index Size (bytes) | Page Utilization | Write Accesses for Build | Read Accesses for Build |
|---|---|---|---|---|---|
| 1 | 5.96 | 8,192 | 52.1% | 14,799 | 75,862 |
| 2 | 5.91 | 8,192 | 52.0% | 14,805 | 71,490 |
| 3 | 5.99 | 8,192 | 51.7% | 14,831 | 73,898 |
| 4 | 5.94 | 8,192 | 52.0% | 14,803 | 71,373 |
| Avg (1-4) | 5.95 | 8,192 | 52.0% | 14,810 | 73,156 |
| 5 | 7.05 | 65,536 | 52.2% | 147,937 | 705,555 |
| 6 | 54.71 | 4,096 | 100.0% | 12,499 | 3,364,634 |

**AHWS V0**

AHWS Version 0.  ANDPP = 2. Signature: width 32, bits/rec 5. 10,000 records. Page size 4.

| Distribution | Avg Retrieval Probes | Index Size (bytes) | Page Utilization | Write Accesses for Build | Read Accesses for Build |
|---|---|---|---|---|---|
| 1 | 1.01 | 88,774 | 53.6% | 14,666 | 29,619 |
| 2 | 1.01 | 88,069 | 54.2% | 14,607 | 30,543 |
| 3 | 1.01 | 88,707 | 53.6% | 14,661 | 28,840 |
| 4 | 1.01 | 88,479 | 53.9% | 14,642 | 29,038 |
| Avg | 1.01 | 88,507 | 53.8% | 14,644 | 29,510 |

**AHWS V0.8**

AHWS Version 0. ANDPP = 8.

| Distribution | Avg Retrieval Probes | Index Size (bytes) | Page Utilization | Write Accesses for Build | Read Accesses for Build |
|---|---|---|---|---|---|
| 1 | 1.05 | 65,631 | 52.2% | 14,789 | 67,199 |
| 2 | 1.03 | 65,655 | 54.2% | 14,788 | 63,606 |
| 3 | 1.04 | 66,063 | 51.8% | 14,791 | 65,626 |
| 4 | 1.05 | 65,718 | 52.1% | 14,808 | 63,733 |
| Avg | 1.04 | 65,767 | 52.6% | 14,794 | 65,041 |

**AHWS V1**

AHWS Version 1. Signature: width 32, bits/rec 5. 10,000 records. Chain size 5. Page size 4.

| Distribution | Avg Retrieval Probes | Index Size (bytes) | Page Utilization | Write Accesses for Build | Read Accesses for Build | Chain Utilization |
|---|---|---|---|---|---|---|
| 1 | 1.01 | 67,802 | 52.3% | 14,782 | 28,440 | 72.1% |
| 2 | 1.01 | 67,658 | 52.5% | 14,765 | 28,467 | 72.0% |
| 3 | 1.01 | 68,032 | 52.1% | 14,796 | 28,422 | 72.0% |
| 4 | 1.01 | 68,042 | 52.2% | 14,785 | 28,393 | 71.9% |
| Avg. (1-4) | 1.01 | 67,884 | 52.3% | 14,782 | 28,431 | 72.0% |
| 5 | 1.01 | 665,072 | 52.5% | 147,657 | 285,130 | 72.4% |
| 6 | 1.00 | 42,032 | 100.0% | 12,499 | 39,972 | 60.0% |

**AHWS V1.N**

AHWS Version 1. Null origins.

| Distribution | Avg Retrieval Probes | Index Size | Page Utilization | Chain Utilization | Write Accesses | Read Accesses |
|---|---|---|---|---|---|---|
| 1 | 1.05 | 62,493 | 52.3% | 72.1% | 14,782 | 30,485 |
| 2 | 1.03 | 62,361 | 52.5% | 72.0% | 14,765 | 30,512 |
| 3 | 1.04 | 62,704 | 52.1% | 72.0% | 14,796 | 30,465 |
| 4 | 1.04 | 62,714 | 52.2% | 71.9% | 14,785 | 30,437 |
| Avg | 1.04 | 62,568 | 52.3% | 72.0% | 14,782 | 30,475 |

**AHWS V2**

AHWS Version 2. Signature: width 16, bits/rec 3. 10,000 records. Chain size 5. Page size 4.

| Distribution | Avg Retrieval Probes | Index Size (bytes) | Page Utilization | Write Accesses for Build | Read Accesses for Build | Chain Utilization |
|---|---|---|---|---|---|---|
| 1 | 1.26 | 30,368 | 90.4% | 31,015 | 66,603 | 78.1% |
| 2 | 1.26 | 31,016 | 90.1% | 29,604 | 62,362 | 76.7% |
| 3 | 1.26 | 30,496 | 90.2% | 30,554 | 65,661 | 77.9% |
| 4 | 1.26 | 30,400 | 90.5% | 30,827 | 67,477 | 77.9% |
| Avg. (1-4) | 1.26 | 30,750 | 90.3% | 30,500 | 65,526 | 77.7% |
| 5 | 1.26 | 303,744 | 90.0% | 297,197 | 627,883 | 96.7% |

**AHWS V2.N**

Null origins. AHWS Version 2. Signature: width 16, bits/rec 3. 10,000 records. Chain size 5.

Page size 4.

| Distribution | Average Retrieval Probes | File Index Size (bytes) | Page Utilization | Write Accesses for Build | Read Accesses for Build | Chain Utilization |
|---|---|---|---|---|---|---|
| 1 | 1.90 | 29,652 | 90.4% | 31,015 | 67,218 | 78.1% |
| 2 | 1.66 | 30,291 | 90.0% | 29,604 | 62,929 | 76.7% |
| 3 | 1.86 | 29,785 | 90.2% | 30,554 | 66,256 | 77.9% |
| 4 | 1.88 | 29,691 | 90.5% | 30,827 | 67,477 | 77.9% |
| Avg | 1.83 | 29,855 | 90.3% | 30,500 | 65,970 | 77.7% |

**AHWS V2.32**

AHWS Version 2. Signature: width 32, bits/rec 5. 10,000 records. Chain size 5. Page Size 4.

| Distribution | Avg Retrieval Probes | Index Size (Bytes) | Page Utilization | Write Accesses for Build | Read Accesses for Build | Chain Utilization |
|---|---|---|---|---|---|---|
| 1 | 1.04 | 36,032 | 90.4% | 31,015 | 66,603 | 78.1% |
| 2 | 1.03 | 36,810 | 90.0% | 29,604 | 62,362 | 76.7% |
| 3 | 1.04 | 36,186 | 90.2% | 30,554 | 65,661 | 78.0% |
| 4 | 1.04 | 36,070 | 90.5% | 30,827 | 66,798 | 77.9% |
| Avg | 1.04 | 36,275 | 90.3% | 30,500 | 65,356 | 77.7% |

**AHWS V2.32.10**

AHWS Version 2. Signature: width 32, bits/rec 5.
10,000 records. Chain size 10. Page size 4.

| Distribution | Avg Retrieval Probes | Index Size | Page Utilization | Chain Utilization | Read Accesses for Build | Write Accesses for Build |
|---|---|---|---|---|---|---|
| 1 | 1.08 | 32,387 | 94.9% | 74.0% | 101,376 | 49,591 |
| 2 | 1.08 | 33,250 | 94.7% | 72.1% | 96,708 | 47,447 |
| 3 | 1.08 | 32,774 | 94.9% | 73.0% | 100,255 | 48,784 |
| 4 | 1.08 | 32,792 | 94.8% | 73.0% | 101,632 | 49,021 |
| Avg | 1.08 | 32,801 | 94.8% | 73.0% | 99,993 | 48,711 |

Theoretical results:

Average probes per retrieval

| | |
|---|---|
| AHWS V0.2 | 1.002 |
| AHWS V1 | 1.002 |
| AHWS V2 | 1.126 |
| AHWS V2.32 | 1.016 |

These figures were calculated using Equation 1, and adjusting for average page utilizations which varied from 52% to 90%. ie. *u* = (Page Size) * (Page Utilization) = 4 * (Page Utilization).

## *Appendix B: Algorithm Pseudo-code*

The following algorithms are for full/prefix origins:

```
insert (key : key_type) is
   do
      if hash (key) = hash_table_size-1 then
         top = chain_table_size-1;
      else
         top = hash_table[hash(key)+1];
      endif;
      for i = hash(key) to top do
         if key > chain_table[i+1].origin then
            iterate_loop;
         endif;
         for j = 0 to last_page_on_chain do
            page = fetch_page (chain_table[i].page_num[j]);
            if (key <= max_key(page)) OR (last_page(i,j,top)) then
               put_in_page (page,i,j,key);
               exit;
            endif;
         endfor;
      endfor;
   end -- insertion

put_in_page (pg:page_type, chn:int, pg_offset:int, key:key_type) is
   do
      x = pg.data[PAGE_SIZE];
      if x = EMPTY then    -- no split needed
         pg.data[PAGE_SIZE] = key;
         sort_keys (pg);
         if pg_offset = 0 do update_origin;
         compute_new_signature (chn, pg_offset);
         write_page (pg, chain_table[chn].page_num[pg_offset]);
      else
         mx = max (key, x);
         mn = min (key, x);
         pg.data[PAGE_SIZE] = mn;
         sort_keys (pg);
         write_page (pg, chain_table[chn].page_num[pg_offset]);
         create_new_page (mx, chn, pg_offset);
      endif;
   end

create_new_page (key :key_type, chn :int, offset :int) is
   do
      new_pg_descrip = add_page_to_file (key);
      if NOT chain_full(chn) then
         insert_on_chain (new_pg_descrip, chn, offset+1);
            -- shifts subsequent pages right by one
      else  -- split this chain
         new_chain = create_new_chain (chn);
         insert_in_table (new_chain, chain_table, chn+1);
            -- shifts subsequent pages right by one
         put_half_pages (chn, chn+1);
            -- move 0.5 pages from chn to chn+1
         if offset > PAGE_SIZE / 2 then
            chn = chn+1;
            offset = offset - (PAGE_SIZE / 2);
         endif;
         insert_on_chain (new_pg_descrip, chn, offset+1);
         if chain_table_size > (Threshold * hash_table_size) then
            reorganize;
         endif;
      endif;
   end

reorganize is
   do
      hash_func_level = hash_func_level + 1;
      hash_table_size = 2 * hash_table_size;
      expand (hash_table, hash_table_size);
      for i = 1 to hash_table_size-1 do
         hash_table[i] = EMPTY;
      endfor;
      for i = 1 to chain_table_size-1 do
         this_chain = chain_table[i];
         this_origin = this_chain.origin;
         j = hash(this_origin);
         while hash_table[j] = EMPTY do
            hash_table[j] = i-1;
            j = j - 1;
         endwhile;
      endfor;
      j = hash_table_size - 1;
      while hash_table[j] = EMPTY do
         hash_table[i] = chain_table_size-1;
         j = j - 1;
      endwhile;
   end -- reorganization for full origins

retrieve (key : key_type) : record_type is
   do
      bucket = hash (key);
      if bucket = hash_table_size-1 then
         -- bucket is last in hash table
         top = chain_table_size-1;
         -- top is last chain in chain table
      else
         top = hash_table[bucket+1];
         -- top is first chain for next hash table bucket
      endif;
      for i = hash_table[bucket] to top
         if (key >= chain_table[i+1].origin) then
            -- for full origins
            -- prefix (key) > origin  for prefix origins
            -- partial (key) > origin for partial origins
            iterate_loop;
         endif;
         j = 0;
         while j < Chain_Size do
            if signature_match (key, chain_table[i].page_sig[j])
               page = fetch_page (chain_table[i].page_num[j]);
               if key_on_page()
                  return page.data[j];
               endif;
            endif;
            j = j + 1;
         endwhile;
      endfor;
   end -- retrieve
```

The following are algorithms for partial indices:

```
reorganize is
   do
      hash_func_level = hash_func_level + 1;
      old_size = hash_table_size;
      hash_table_size = 2 * hash_table_size;
      expand (hash_table, hash_table_size);
      for i = old_size-1 downto 1 do
         hash_table[2*i] = hash_table[i];
      endfor;
      for i = 1 to chain_table_size-1 step +2 do
         start = hash_table[i-1];
         if i = hash_table_size-1 then
            stop = chain_table_size-1;
         else
            stop = hash_table[i+1];
         endif;
         if start = stop then
            hash_table[i] = start;
         else
            entered = false;
            found = false;
            for j = start+1 to top-1 do
               entered = true;
               if chain_table[j].origin > 7Fh then
                  found = true;
                  break; -- retains j's value
               endif;
            endfor;
         endif;
         if not found or not entered then
            page = fetch_page (chain_table[top].page_num[0]);
            if not entered then
               if hash (page.data[0]) >= i then
                  hash_table[i] = bot;
               else
                  hash_table[i] = top;
               endif;
            else
               if hash (page.data[0]) >= i) then
                  hash_table[i] = top-1;
               else
                  hash_table[i] = top;
               endif;
            endif;
         else
            hash_table[i] = j-1;
         endif;
      endfor;
      adjust (prefix_mask, prefix_shift_count);
      for i = 0 to chain_table_size-1 do
         chain_table[i].origin = 2 * chain_table[i].origin + 1;
         if chain_table[i].origin = FFh then
            page = fetch_page (chain_table[i].page_num[0]);
            chain_table[i].origin = origin (page.data[0]);
         endif;
      endfor;
   end -- reorganization for partial origins

insert (key : key_type) is
   do
      if hash (key) = hash_table_size-1 then
         top = chain_table_size-1;
      else
         top = hash_table[hash(key)+1];
      endif;
      for i = hash(key) to top do
         page = fetch_page (chain_table[i+1].page_num[0]);
         chain_table[i+1] = calc_prefix (page.data[0]);
         if key >= page.data[0]
```

```
            iterate_loop;
         endif;
         for j = 0 to last_page_on_chain do
            page = fetch_page (chain_table[i].page_num[j]);
            if (key <= max_key(page)) OR (last_page(i,j,top)) then
               ripple_insert (page,i,j,key,top);
               exit;  -- leave insert
            endif;
      endfor;
   end -- insert

ripple_insert (pg:page_type, chn:int, pg_offset:int, key:key_type,
               top:int) is
   do
      x = pg.data[PAGE_SIZE];
      while x != EMPTY do
         if last_page (top,chn,pg_offset) then
            mx = max (key, x);
            mn = min (key, x);
            pg.data[PAGE_SIZE] = mn;
            sort_keys (pg);
            compute_new_signature (chn, pg_offset);
            write_page (pg, chain_table[chn].page_num[pg_offset]);
            create_new_page (mx, chn, pg_offset);
            return;
         else
            if key < x then
               pg.data[PAGE_SIZE] = key;
               sort_keys (pg);
               if pg_offset = 0 then
                  update_origin (pg, chn);
               endif;
               compute_new_signature (chn, pg_offset);
               write_page (pg, chain_table[chn].page_num[pg_offset]);
               key = x;
            endif;
            pg_offset = pg_offset + 1;
            if pg_offset >= CHAIN_SIZE then
               chn = chn + 1;
               pg_offset = 1;
            elseif chain_table[chn].page_num[pg_offset] = NULL then
               pg_offset = pg_offset - 1;
               top = chn; -- force a split on this
                          -- chain since there is room
            endif;
            pg = chain_table[chn].page_num[pg_offset];
            x = pg.data[PAGE_SIZE];
         endif;
      endwhile;
      -- ok to insert on this page
      pg.data[PAGE_SIZE] = key;
      sortkeys (pg);
      compute_new_signature (chn, pg_offset);
      if pg_offset = 0 then
         update_origin (pg, chn);
      endif;
      write_page (pg, chain_table[chn].page_num[pg_offset])
   end -- ripple_insert
```