A Comparative Evaluation of Indexing Schemes

By Comarapalyam K. Guru Prasad
and John W. Roach

TR 90-8

A Comparative Evaluation of Indexing Schemes

By Comarapalyam K. Guru Prasad
and John W. Roach

TR 90-8

*

Technical Report SRC-90-002

A Comparative Evaluation of Indexing Schemes*

Comarapalyam K. Guru Prasad and John W. Roach

Department of Computer Science
at
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

August 1989

---

*Work supported by the U.S. Navy through the Systems Research Center under Basic Ordering Agreement N60921-83-G-A165 B044.

*Cross referenced as CS-90-8, Department of Computer Science, Virginia Tech.

ABSTRACT

An empirical experiment is reported that compares three indexing techniques used to help answer queries for a medium-sized database. The experiment compares memory and time utilization for B+ trees, superimposed codeword and perfect hashing indexing techniques. Costs for creating and querying the database are compared over a range at different page and buffer sizes for the database. We compare and contrast the advantages and disadvantages of the indexing techniques.

CR Categories and Subject Descriptors: H.2.2 [Database Management]: Physical Design - *Access methods*

Additional Key Words and Phrases: Comparative evaluation, indexing technology

1. Introduction

This paper addresses the problem of determining a good indexing mechanism for a relational database system by empirically comparing various indexing techniques. A large number of experiments were conducted on different indexing schemes over a medium sized relational database and the results compared.

The experiments that we report here were instrumented to compare efficiencies in time and space utilization efficiency. Careful attention has been paid to query design and to measuring efficiencies as various database parameters, such as page size, were varied.

The indexing mechanisms that we implemented and compared are

1. B+ - tree indexing.
2. Perfect (collisionless) hashing.
3. Superimposed codeword indexing.

We choose to test B+ trees because they have become the standard indexing technique; we tested the superimposed codeword technique because it seemed a promising alternative and because it can be extended to more complex data types for use, say, in a Prolog type database; we tested a perfect hashing scheme to determine the best possible performance for the given database and use it as a control in the experiment.

2. The Indexing Problem: A Discussion

Indexing can be thought of as a routing function. Given a key value of an attribute, the indexing mechanism returns the address (logical or physical) of the record containing that key value of the attribute. An index lets us impose order on a file without actually

rearranging the file. This not only keeps us from disturbing already stored records, but also makes operations like record additions, deletions, etc., less expensive. The index structure stores the required information when a record is inserted into the database. During the retrieval process this information is used to search efficiently for that record in the database.Index structure design involves a compromise of speed and space. If more information is stored in the index structure, the retrieval is faster but the space occupied by the index structure is more. If less information is stored in the index structure, the time required to search will increase.

In general the index structure will be involved during the following operations of a database management system.

1. Insertion of records into the database.
2. Retrieval of records from the database with certain constraints.
3. Deletion of records from the database.
4. Updating of records (which is a combination of the above three operations).

In addition to all of the above operations, the indexing mechanism may have its own overhead procedures that come into operation under certain circumstances. For example when the file size grows beyond a particular size, a series of expansions may be triggered within the index structure. These overhead operations can become time consuming and must be considered carefully in the overall selection and design of an indexing scheme.

Indexing schemes with superb retrieval times in terms of disk accesses may consume prohibitive amounts of time during other operations like insertion, expansion etc. A good example in this category is the recent linear hashing with separators proposed by [Lars88].

Applicability of such indexing schemes again depends on the type of database at hand. If the database is not dynamic, i.e., the retrievals far outnumber other operations, then the overall performance might still be satisfactory.

Often the way in which a database is going to be used cannot always be known beforehand. A typical example is a Prolog relational database system that represents facts of the real world (knowledge base). In these cases an indexing scheme that is optimal with respect to all the above database management operations has to be chosen to achieve overall good performance.

There are also other considerations that influence the selection and design of an indexing scheme. They are discussed in detail in the following sections.

Flexibility of the Indexing Structure:

The flexibility of an indexing structure is defined with respect to various factors. They are

Ability to handle different file sizes without discrimination:

Ideally, the performance (retrieval time, insertion time etc.) should not change unreasonably with respect to the file sizes. In practice this could be controlled by the parameters in the design process. For example, the size of the node (internal or leaf) in a B+-tree indexing scheme is a very important parameter which influences the variations of the system's performance with respect to file sizes and storage devices.

Ability to handle duplicate key records in a natural way:

Many indexing schemes assume no duplicate records with respect to an attribute. The duplicate records represent facts of the real world which have common attributes. The way in which the duplicate records are handled by an indexing scheme can often play a crucial role.

Complexity of the Indexing Scheme:

Complexity of the indexing scheme affects the performance in two ways. During implementation stages it will result in

- a comparatively greater number of 'bugs' and higher lead time.
- less scope for improvements.

During the maintenance phase

- extensions and modifications would be relatively more difficult if the system is inherently complex, and
- analysis of performance would be difficult.

Storage Efficiency:

We define storage efficiency of an index structure as follows: storage efficiency of an index structure is the ratio of storage (space) occupied by the index structure to the amount of storage (space) occupied by the actual database (in our case the actual database is the set of all the tuples). This definition gives us a measure of the amount of overhead space used for indexing purposes.

Ideally, this storage efficiency has to be carefully observed during the selection of an indexing scheme, but we will always reap the benefits of quick access, even if the storage efficiency is relatively less. In practice, if the system has relatively large storage, lower storage efficiencies could be tolerated.

Time Considerations:

Time complexity is one of the most important factors in the selection of an indexing scheme. Recently, the applicability and suitability of an indexing scheme has often been measured mainly on this criterion.

Disk access complexity or disk complexity could be defined in the same way as time complexity is defined: O(n) where n is the number of records in the database on which the indexing is applied. O(n) gives us the order of the equation of number of disk accesses made if there are n records.

In more simple terms, the average number of disk accesses that an indexing scheme makes to access a record out of n records gives us the time complexity of the index structure. This obviously should be as low as possible, and this is what most of the indexing schemes in the literature try to achieve. In recent times, this has become the de-facto standard to measure the quality of an indexing scheme as can be seen in [Enbo88].

In practice, during the design and implementation stages of an indexing mechanism very careful attention should be given to disk accesses. There are various compromises seen with respect to disk accesses. If the number of disk accesses during retrieval is reduced, then during insertion or expansion of keys a corresponding increase in the number of disk accesses is observed. A very typical example is the linear hashing with separators scheme [Lars88]. A single disk access retrieval sometimes is accompanied by prolonged disk accesses during an insertion or expansion operation. As mentioned earlier, however, this may not pose a problem for a static database.

Sometimes, in order to reduce the disk access complexity, a very sophisticated algorithm will be tried resulting in a very complex index structure. Again the linear hashing with separators [Lars88] and other dynamic hashing schemes proposed in [Enbo88] are typical examples.

In conclusion, disk access complexity is the most important factor in the overall performance of an indexing scheme and is to be kept at a minimum as long as this does not unduly affect other factors.

3. *B+-tree Indexing*

The B-tree and its variation the B+-tree have become a defacto standard for file organization in large databases requiring dynamic updates [Come79]. File index of users, dedicated database systems and general purpose access methods have all been proposed and implemented using B-tree structures. These provide a general method for storing and retrieving data in large file systems that provides rapid access to the data with minimal overhead costs.

One of the major modifications that needs to be carried out when using a B+-tree indexing for a database system is to enhance its capability to handle duplicate records (duplicate records are defined as records containing the same key value for the same attribute position).

The implementation chosen was conceptually simple. Duplicate keys being inserted are considered as if they were a unique key. The actual position of the duplicate key to be inserted would be at the last position (end) of the duplicate key set. During retrieval, the first key (left most key) in the duplicate key set is looked for. The first instance of the key is retrieved. Splitting and concatenation proceeds in the same way, where duplicate keys are considered as just another key. A separate B+-tree is built for each of the arguments (parameter positions) of a relation.

A paging scheme is used for secondary storage accesses which uses Least Recently Used (LRU) policy for page replacements.

Superimposed Codeword Indexing

In this section we cover the second method of indexing technology used for our relational database system. Superimposed codeword indexing provides a very efficient method of retrieving records from large databases in only a small number of disk accesses [Rama86].

In a superimposed coding scheme [Rama86], each record in the database is associated with a descriptor (bit string). This descriptor is formed by superimposing (bitwise OR-ing) the codewords for the individual attributes in a record. The codewords, in turn, are formed by using the attribute value to generate a series of bit positions to be set to 1. Codewords for attributes are generated using a random number generator passing the attribute as a seed to it. The codewords generated for all the attributes of a fact are then OR ed to form a superimposed codeword.

Instead of a bit-sliced representation, we have byte-sliced representation for record descriptors in our method. This is an improvement over the original superimposed codeword indexing where bit-slicing was proposed to store codewords. Bit-sliced implementation was tried but the results were poor compared to the byte-sliced method. This implies that instead of viewing the descriptor file as N bit-strings, each containing b bits, we view it as b/8 (1 byte = 8 bits assumed) strings (slices) each containing N bytes. The ith slice is a list of the ith bytes of all descriptors. If the k th bit in the i th slice is set, then this indicates that the k th record has the i th bit set in its descriptor. During query matching, the comparison between query codeword and the record descriptors are in terms of bytes instead of in bits. The reason behind this modification is to optimize the number of disk accesses.

These byte-slices (four byte-slices) are stored sequentially in the corresponding byte-slice descriptor page. The pointer (address) to the fact is stored separately in a

"map table". The map table consists of sequence of addresses (page number and offset within the page) and the i th address in this sequence corresponds to the i th descriptor in the descriptor page. When the query codeword matches with a descriptor in the descriptor page at i th position then the i th address is retrieved from the map table and the fact corresponding to the i th descriptor is retrieved for unification.

4. Perfect Hashing:

In perfect hashing scheme, a unique number is generated for each of the database keys and is stored in an atom table as atom numbers. These numbers are used as hash values for hashing. Since atom numbers are unique, the hash table obtained will be collisionless, resulting in perfect hashing.

For each key in the database there will be a unique number associated with it. The index for a relation P is a two dimensional structure as shown in Figure 1. This index structure exists for each of the relations in the database. The rows correspond to the atom numbers of all the keys present in the database. The columns correspond to the argument numbers or positions. In this scheme individual keys of a fact are indexed separately. Depending on the key position (argument number) within the fact the column is selected. Atom number of the key determines the row selected in the index structure. For example, given a fact P a b c, where P is the relation name, a, b and c are keys with atom numbers as shown in Figure 1. To index this fact with b as the primary key the fourth row is selected (since the atom number of b is 4). Key b is in the second argument position of the fact. So the second column is selected. The address of the fact is stored in this position. The duplicate facts (i.e. with different keys in other argument positions) are stored within the same element of the matrix as a linked list.

Similarly the same fact is stored considering a and c (1st and 3rd arguments) as the primary keys as shown in the Figure 1. During searching of a fact an instantiated key (key having a value as opposed to a variable) is selected and the corresponding row is considered. Depending upon the key's argument position the column number is selected and the address of the fact to be searched is obtained. For duplicate facts the linked list is followed.

In the present implementation of this scheme most of the index information is stored in RAM and only the actual facts are stored in the secondary storage. This results in a reduced number of disk accesses during various database operations. The main motivation to use this scheme for comparison is that this would be used as a reference or an ideal indexing scheme, since this uses RAM for storage. Using RAM for indexing is the ideal case, but because of limited availability of RAM in practice, standard indexing schemes use secondary storage.

5. *Experimental Design and Setup*

In this section, design of experiments for comparative evaluation of indexing schemes are discussed and the experimental setup described. The principles for query selection to test indexing schemes are provided. The performance measures to evaluate indexing schemes are described in detail along with the assumptions made for experimentation purposes.

Objective

The aim of the experiments is to determine performance measures of indexing schemes during different operations on the underlying Prolog relational database. The indexing schemes compared are:

1. B+- tree indexing.
2. Superimposed codeword indexing
3. Collisionless (perfect) hashing.

Database

The database used for experimentation purposes comes from the Naval Surface Weapons Center (NSWC), United States Navy, the sponsors of this research effort. The database consists of several relations and each relation has 3000-15000 facts as shown in table 1.

Apart from the indexing module, the relational database system consists of various other modules, all of which work in close coordination. The indexing module uses the services of the page manager for input/output operations and main memory management. The entire database as well as the index structure reside in the secondary storage in the form of pages.

A page manager was developed to handle all functions related to paging, such as "get page", "new page", "write page" etc. This served to separate the low level paging functions from other parts of the program. By making calls to the page manager, the other modules of the program can access secondary storage data objects. This implies that the indexing procedures access the index structure, residing in secondary storage, through the page manager by making appropriate procedure calls.

Index structures are built using these pages and invariably the parameters of the paging scheme affects the performance of the indexing mechanism considerably. For example, in a B+-tree a node is held in a page. If the page size is varied the number of keys held in a node will change, which affects the depth of the tree. Similarly the number of buffer pages held in RAM may mean a difference of disk access or not, for the indexing mechanism. As explained earlier, the number of disk accesses is one

Relation Name	Number of Facts	Number of Args
IsProcess	3046	3
Entity	4029	3
isentity	4029	1
Collectedin	4132	3
utilizes	6350	3
upd	8428	3
Containedin	10906	3
Element	10906	4
iselement	10906	3
partof	10906	4

Table 1. Details of the test database from Naval Surface Warfare Center.

of the important performance characteristics of an indexing scheme. The effects of these parameters on indexing will also be seen in this paper.

The main assumption is that our test database represents in general, a typical database of the real world. The size of our test database is around 100,000 facts including different relations and the statistical characteristics of standard deviation, distributions etc., are assumed to be normal and uniform respectively. The large number of facts present in the database makes us assume this. Moreover, the database used for testing is used by the sponsors in the real world.

6. *Experimental Design*

In this section details of the performance measures used to compare indexing schemes are explained. The design of queries used for experiments along with the principles followed, are discussed. The two stages of experiments conducted are outlined.

Experiments conducted can be broadly classified into two types, depending on the database operations being performed.

1. insertion of facts.
2. searching (querying) of facts.

Tests and performance measures determined during each of the above stages are given below.

Insertion of Facts

Observations are classified into two types

1. primary observations: directly obtained during the experiment.
2. secondary observations: obtained from primary observations.

Before going further, a note of difference between the terms "page access" and "disk access" is in order. The index structure is stored in secondary storage in the form of pages. When the indexing mechanism has to access a page in the secondary storage, it calls the page manager procedure get_page(page number). This is termed as "page access". The page manager holds a number of pages in a buffer pool, residing in primary storage, RAM. The page manager checks this buffer pool to determine whether the requested page is in the pool or not. If the requested page is not in the pool, then a page fault occurs, and a secondary storage access is made to bring the requested page into RAM. This is termed as "disk access".

But if the requested page is in RAM then there will be no disk access. For zero buffer pages the number of disk accesses is equal to number of page accesses. If P is the number of page accesses and D is the number of disk accesses, then at any given time t,

$D = kP$, where k is the constant of proportionality, dependant on the buffer size N.

Performance Measures

Primary Observations:

1. Number of disk accesses made during insertion of all facts of each relation.
2. Number of page accesses made during insertion of all facts of each relation.
3. Total number of pages used during insertion of all facts of each relation.
4. Actual CPU time utilized during insertion of all facts of each relation.
5. Total number of comparisons made by the key with the keys already inserted.

Secondary Observations:

1. Average number of facts inserted per page access.
2. Average number of facts inserted per disk access.
3. Average number of facts inserted per page.
4. Average CPU time per fact inserted.
5. Total index structure size in pages per total facts size in pages.

Variations:

The above observations were made during insertion of facts with the following variations in the experiment.

1. Insertion of facts of different relations.
2. Variation in size of buffer pool (10, 20, 30 pages).
3. Variation in size of a page (1024, 2048, 4096, 8192 bytes).

Searching (Querying) of Facts

Searching of facts is the second stage of experimentation Facts that are searched or retrieved from the database depends on the query specified. A query is a database record (tuple) with zero or more arguments undefined. In general, the set of facts retrieved from the database is entirely dependant on the query specified. A large number of queries is required to test the indexing schemes for performance measures. This points to the need for efficient and effective query design.

Principles of Query Design

Queries to test indexing schemes for comparative evaluation purposes were designed with the following general principles.

- Facts from different regions of the database extracted. Facts from certain regions of the database may favor a particular indexing scheme i.e., distribution of facts for queries is an important issue and the distribution is desired to be uniform. For example, superimposed codeword scheme searches linearly the codewords of the database. Facts that are in the front get retrieved quickly than the facts that are at the end. By retrieving facts from various regions, we would be averaging out the extreme cases.

- Queries with all combinations of instantiation of arguments. For example,

  A query (P a b c) ; where

  P is the relation name.

  a, b and c are the three arguments of relation P.

  Various combinations of instantiations of the above query:

  (P ?x b c) ; where ?x represents an undefined argument.

  (P a ?x c)

  (P a b ?x)

  (P ?x ?y c)

  (P a ?x ?y)

  (P ?x b ?y)

  (P a ?x ?x)

  (P ?x ?x c) ...

This prevents favoring a particular indexing schemes with respect to the number of arguments defined and a particular argument position. For example, in the

superimposed codeword scheme arguments are superimposed to form a codeword and the codewords are stored. During searching the codewords of the query are again superimposed and searched in the index. If more arguments are defined (instantiated) in the query then the false matches would be reduced and the search quicker. But schemes such as B+- tree indexing have single argument indexing i.e., all the arguments of a relation have their own index trees. In this case instantiation of particular arguments may result in favorable performances.

By specifying all combinations of instantiations of queries we are averaging out the variable performances of indexing schemes for individual arguments and the number of arguments instantiated.

- Queries are designed to vary the number of facts that match them. This is defined as the query selectivity and is expressed as a percentage of the size of the database. For example, 1%, 5%, 15% and 25% etc. 5% query selectivity means 5% of the total database records were matched for this query.

  Selectivity of a query is important since it determines how efficient an indexing scheme is to retrieve 'duplicate' or 'similar' records (records which have same key value in one or more argument positions).

- Queries are specified that do not match any fact in the database. Some indexing schemes does not perform well during failure of a query. For example, superimposed codeword scheme has to try all the codewords before failing.

- Queries designed with varying number of arguments. The number of arguments in the query affects the probability of false matches in general, which in turn affects the speed at which the queries are matched.

7. Query Selection

Around two hundred queries were designed from different relations of the database. The queries followed the principles of query design as explained in the previous section. In this section the query selection is justified considering each principle separately.

Queries were designed such that facts from different regions of the database are retrieved. This was done by physically going through the database and randomly selecting facts at approximately equal intervals. In effect facts for queries were selected from different regions in an uniformly distributed pattern. As mentioned earlier the distribution of facts for queries is an important issue.

Queries were also designed with various combinations of instantiations of arguments. In general all possible combinations for a given relation are used for queries. For example, for a relation with two arguments (P a b) the queries (P ?x b) (P a ?x) (P ?x ?x) etc., were used.

Query selectivity was ensured by observing the number of facts retrieved for a given query. Duplicate facts in the database were noted and queries to extract them were designed. In general, we have retrieved facts ranging from 1% to 50% of the facts of relations.

Queries were selected that do not match any fact in the database. Again various combinations of queries were designed such that the number of keys in the query that match the keys of the facts are varied.

Queries with different arguments were ensured since experiments were carried out on twelve relations of the database which had arguments ranging from 1 to 4.

Performance measures

1. Number of disk accesses made during a search for facts.
2. Number of page accesses made during a search for facts.
3. Actual CPU time during a search for facts.
4. Number of false matches per number of true matches.
5. Number of comparisons between the key searched and the keys in the database during search for facts.

Variations:

The above observations were made during searching of facts with the following variations in the experiment.

1. Variation in size of buffer pool (10, 20, 30 pages).
2. Variation in size of a page (1024, 2048, 4096, 8192 bytes).

In this section the experiment design and setup was discussed in detail. Various performance measures used for comparative evaluation of indexing schemes were touched upon along with detailed explanation of query design. The next section will give results of the experiments and the conclusions based on them.

8. Experimental Results

The cycle of experiments is performed as follows: For a given page size (1024, 2048, 4096 or 8192 bytes) and buffer pool size (10, 20 or 30 pages), facts of all the relations of the test database are inserted using a particular indexing scheme. During insertion of facts into the database, various performance measures are noted in a separate file. Then searching for facts (querying) is carried out using around 200 different queries. Again the performance measures are noted in a file. The above cycle is carried out on all the indexing schemes under study.

In the following sections important experimental results are presented in the form of graphs and tables. In general these graphs represent the most important results ob-

tained, but not the entire set of experiments. All time units are equal to 1/60 th of a CPU second allocated to the running process.

Insertion of facts

Around 100,000 facts were inserted into the database and performance measures like disk accesses, page accesses, insertion times are noted. Figures 3 and 4 show plots of disk accesses vs number of facts inserted and Figures 5 and 6 show insertion time vs number of facts inserted for some sample page and buffer sizes. Figures 5 to 7 show plots of disk accesses vs number of facts inserted for each of the indexing schemes with various page and buffer sizes. For example, Figure 5 shows plots for B+- tree indexing scheme the values of disk accesses for various page and buffer sizes.

Analysis

One of the most significant observations that could be made from the Figures is that perfect hashing or collisionless hashing has very good performance but has serious limitations as to the number of facts that can be stored in the database as it uses RAM for index structure. B+- tree has the best overall performance. Superimposed codeword indexing comes far behind B+- tree in performance.

B+- tree indexing performed quite well and has no limitations as to the number of facts that could be inserted. The performance was almost linear and stable for all page sizes and buffer sizes. For page sizes of 1024 and 2048 bytes, and buffer size of 10 pages, performances of all indexing schemes deteriorated. For page size = 8192 bytes and buffer size = 30 pages, the improvement in performance of B+- tree was not considerable when compared with superimposed codeword indexing. It

should be noted that plots of insertion time vs number of facts inserted (refer Figures 5 and 6) follows very closely to the plot of disk accesses vs number of facts inserted (refer Figures 3 and 4) implying, that the disk access is the most significant performance measure of an indexing scheme.

Page and buffer sizes together determine the total RAM allocated to the index structure. In general more the RAM allocated the less the number of disk accesses made which ultimately determine the performance of an indexing scheme. In the Figures 6 through 8, we observe the predicted behavior clearly. The best performances could be seen for page size = 8192 bytes and buffer size = 30.

In general we see large influences of page and buffer sizes on performances.

The performance curve for superimposed codeword indexing is not linear, and follows approximately a second order curve. After insertion of about 75,000 facts it levels off.

Storage Utilization

Figure 9 shows plots of storage pages used vs facts inserted for various indexing schemes. It can be clearly seen again that perfect hashing has the best secondary storage complexity. Unlike the other cases, here superimposed codeword indexing comes second followed by B+- tree. In superimposed codeword indexing single argument indexing is not done as is done in the other indexing schemes. In B+- tree the overflow of nodes (pages) occurs faster than the overflow of codeword pages in the superimposed codeword indexing scheme. So B+- tree's performance is not good compared with the other indexing schemes.

Searching for facts (Querying)

Figures 10 and 11 show bar plots of disk accesses made vs indexing schemes for various experiments with varying page and buffer sizes on several datasets (relations of the test database). The main observation from all the Figures is that perfect hashing again has the best performance. B+- tree comes next followed by superimposed codeword indexing.

Another important observation is that the page size and the buffer sizes does not affect the performances of B+- tree indexing and perfect hashing considerably, whereas the effect on superimposed codeword indexing by page and buffer size is considerable. For example refer the disk accesses made by superimposed codeword indexing for page size = 8192 bytes and buffer size = 30 pages is almost four times less than for page size = 4096 bytes and buffer size = 20 pages. In superimposed codeword indexing the effect of page size and buffer size on disk accesses is of the order of five in certain cases.

9. Conclusions

Tables 3 and 4 give the overall average performance figures of all the indexing schemes during insertion of facts and searching for facts in the database. It can clearly be seen that perfect or collisionless hashing has very good performance but needs huge amount of primary memory (RAM). This is a serious limitation as it poses a bottleneck for large databases. B+- tree has the best overall performance and has no limitations as to the number of facts that can be inserted into the database. Superimposed codeword indexing performs the worst of the three techniques tested. For performance measures like disk accesses, page accesses and insertion time, search time, number of comparisons B+- tree has a very wide lead over superim-

posed codeword indexing. Only in performance measure of storage used (in pages) does superimposed codeword indexing performs better than B+- tree. Considering all the above factors, we conclude that B+- tree is the best indexing scheme of those we tested for a relational database management system.

Acknowledgements

This work was performed under DoD contract number N60921-83-G-A165-B039.

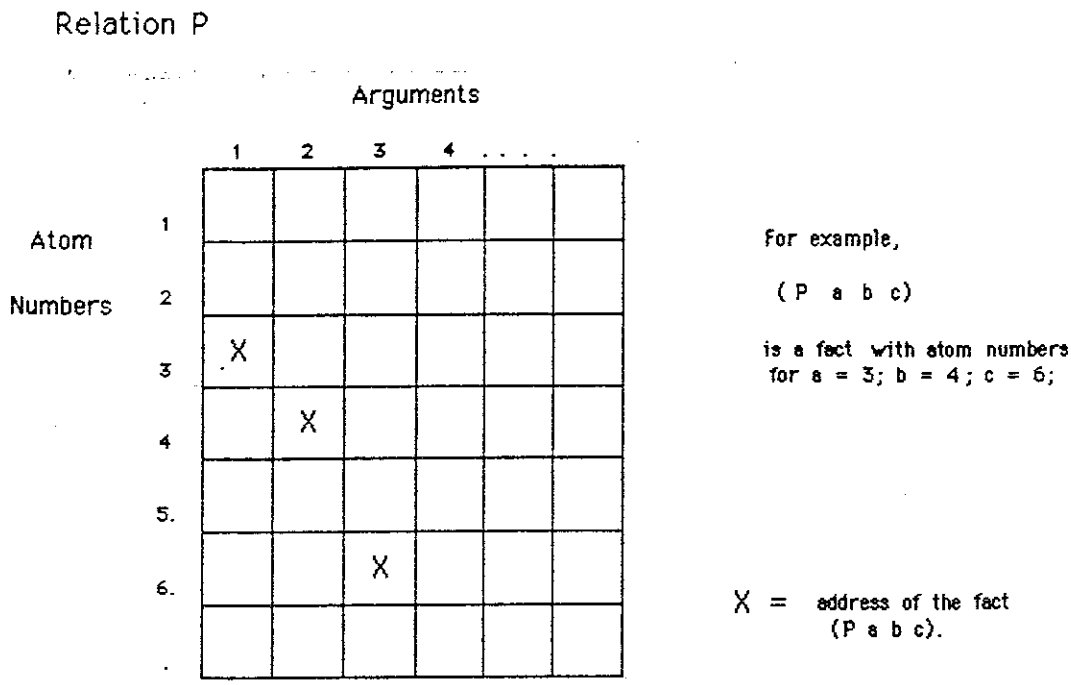

Figure 1. Index Structure of Perfect Hashing.

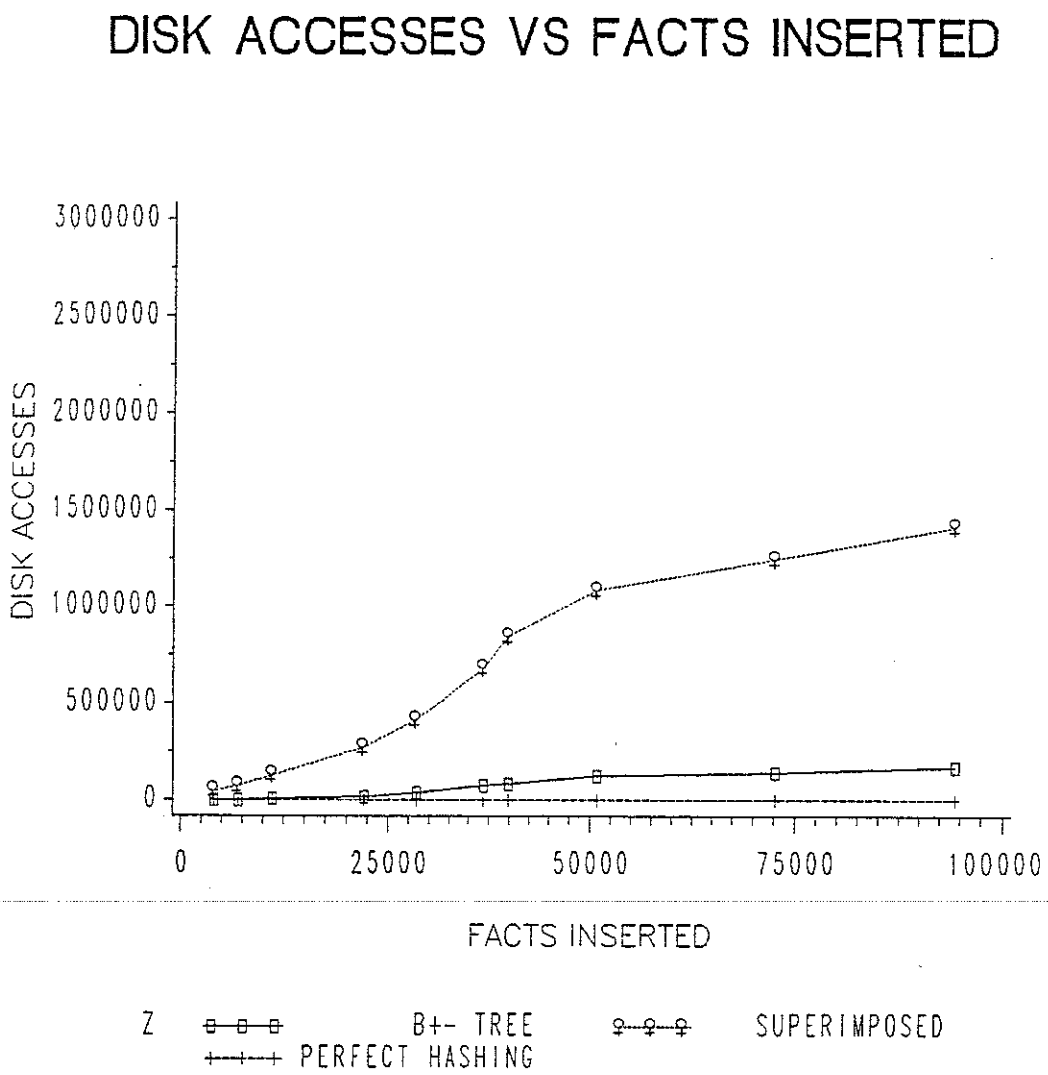

Figure 2. Disk Accesses Vs Facts Inserted

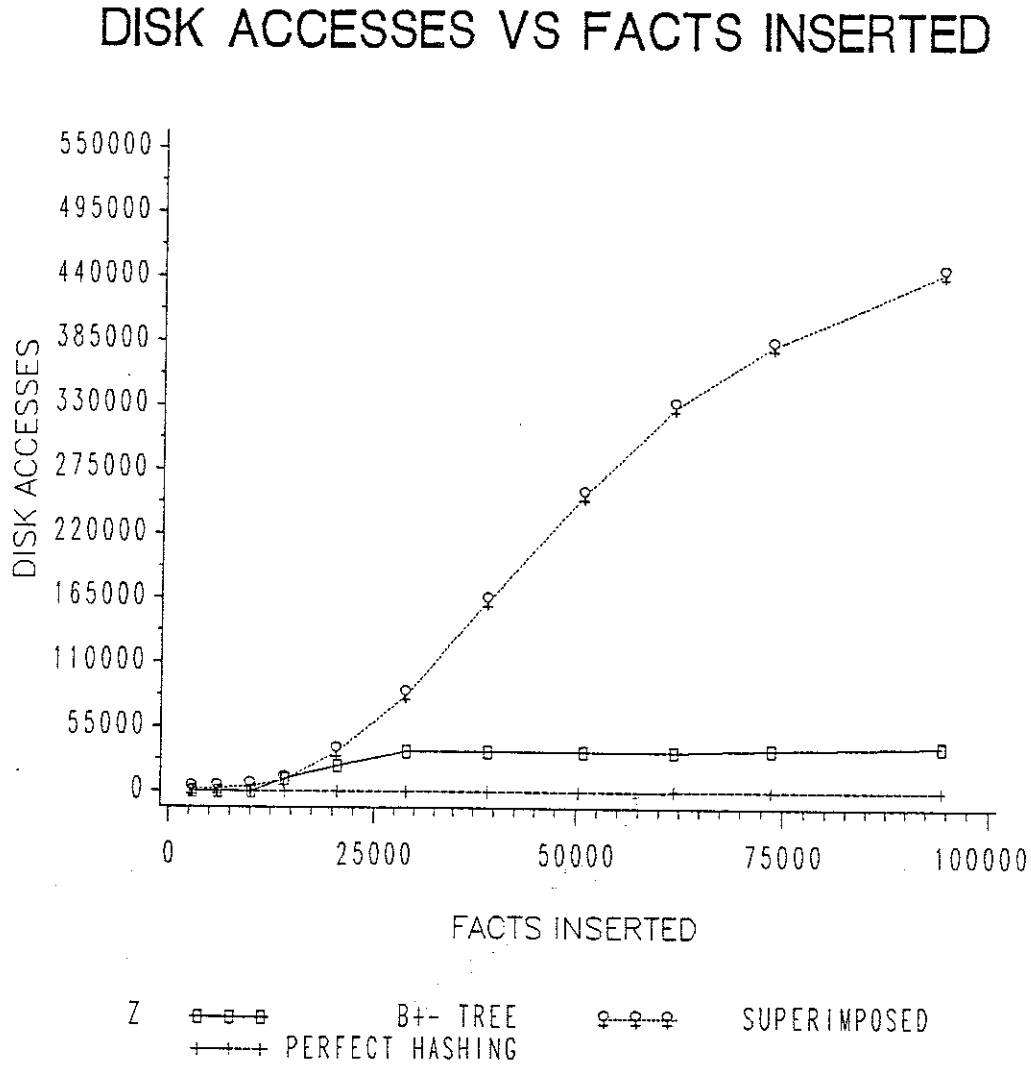

Figure 3. Disk Accesses Vs Facts Inserted

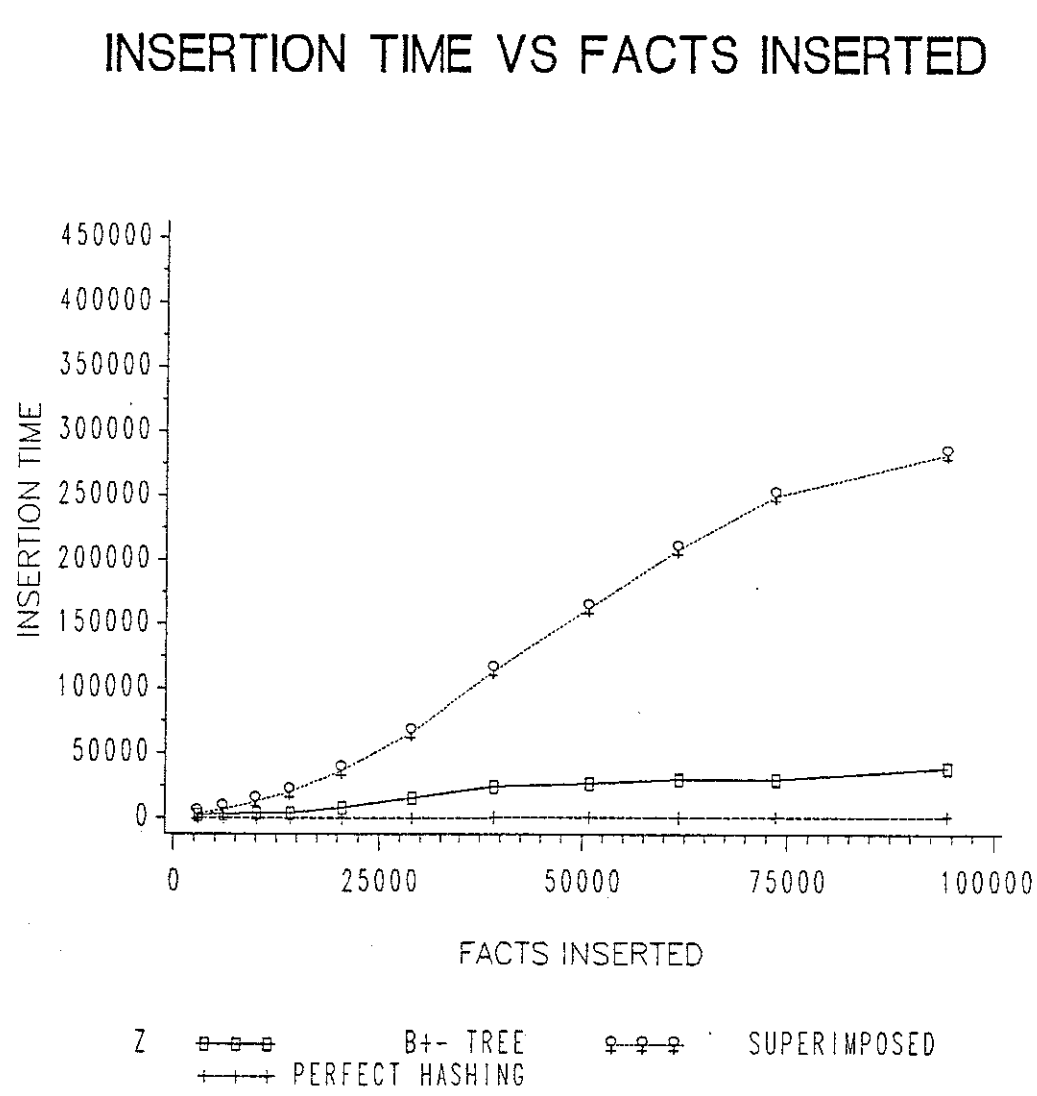

Figure 4. Insertion Time Vs Facts Inserted

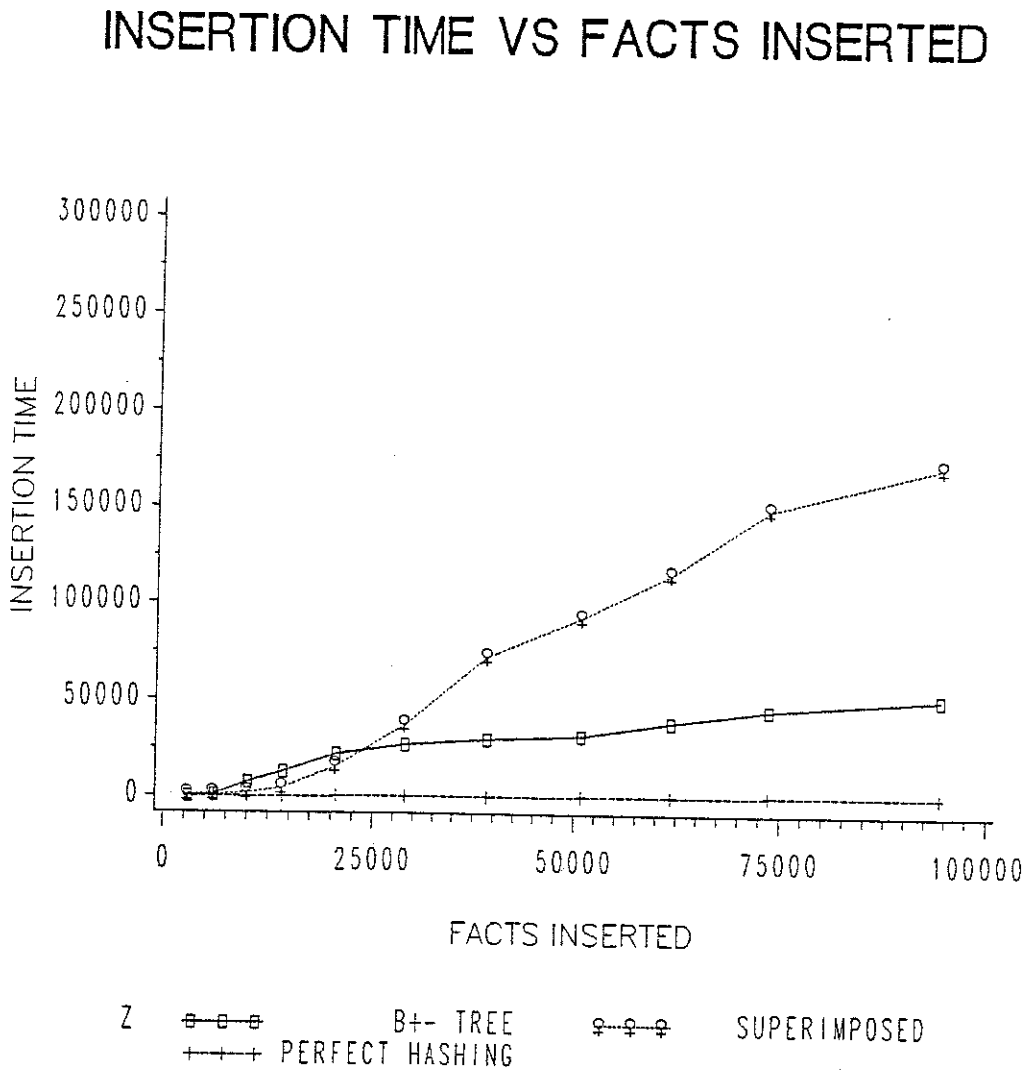

Figure 5. Insertion Time Vs Facts Inserted

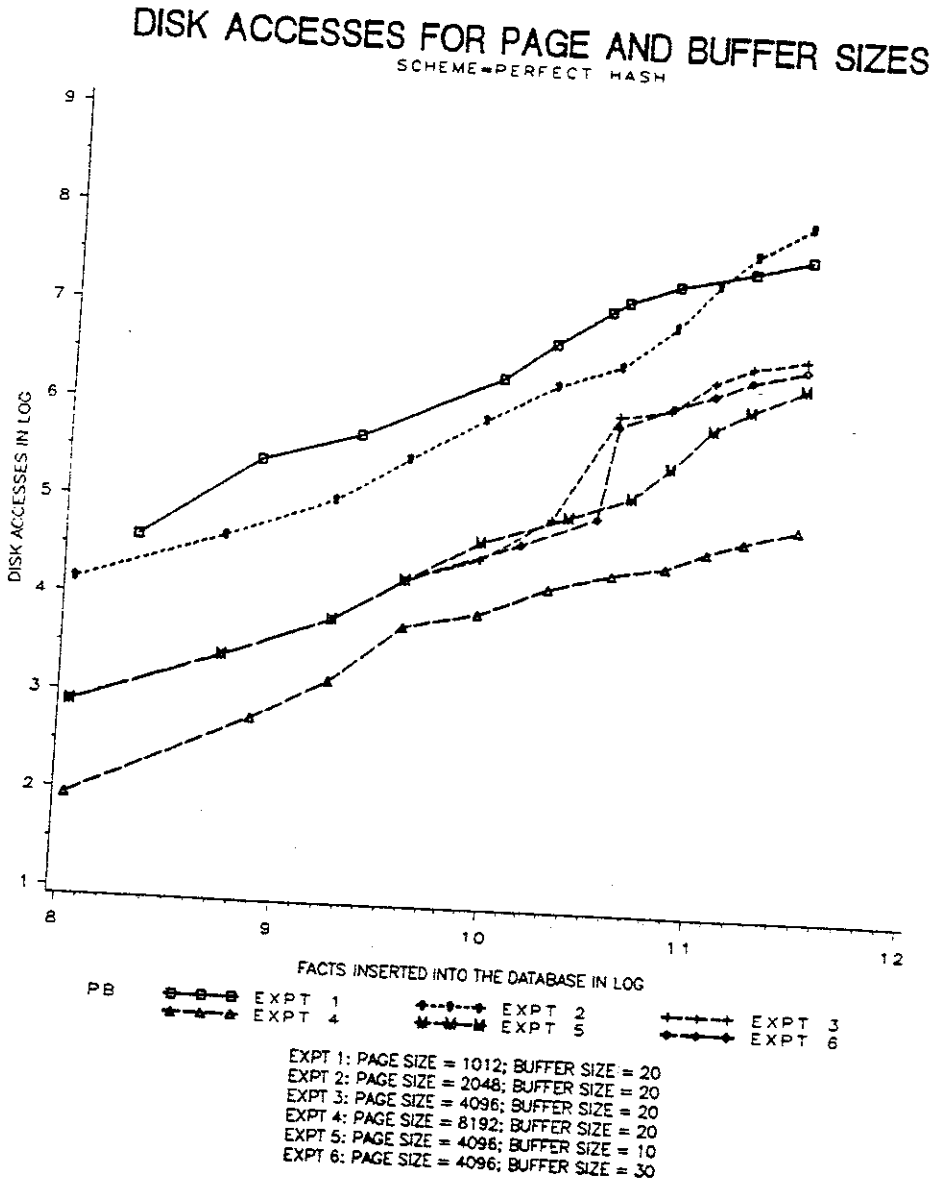

Figure 6. Disk Accesses for Different Page and Buffer Sizes (Perfect Hash).

Abstract

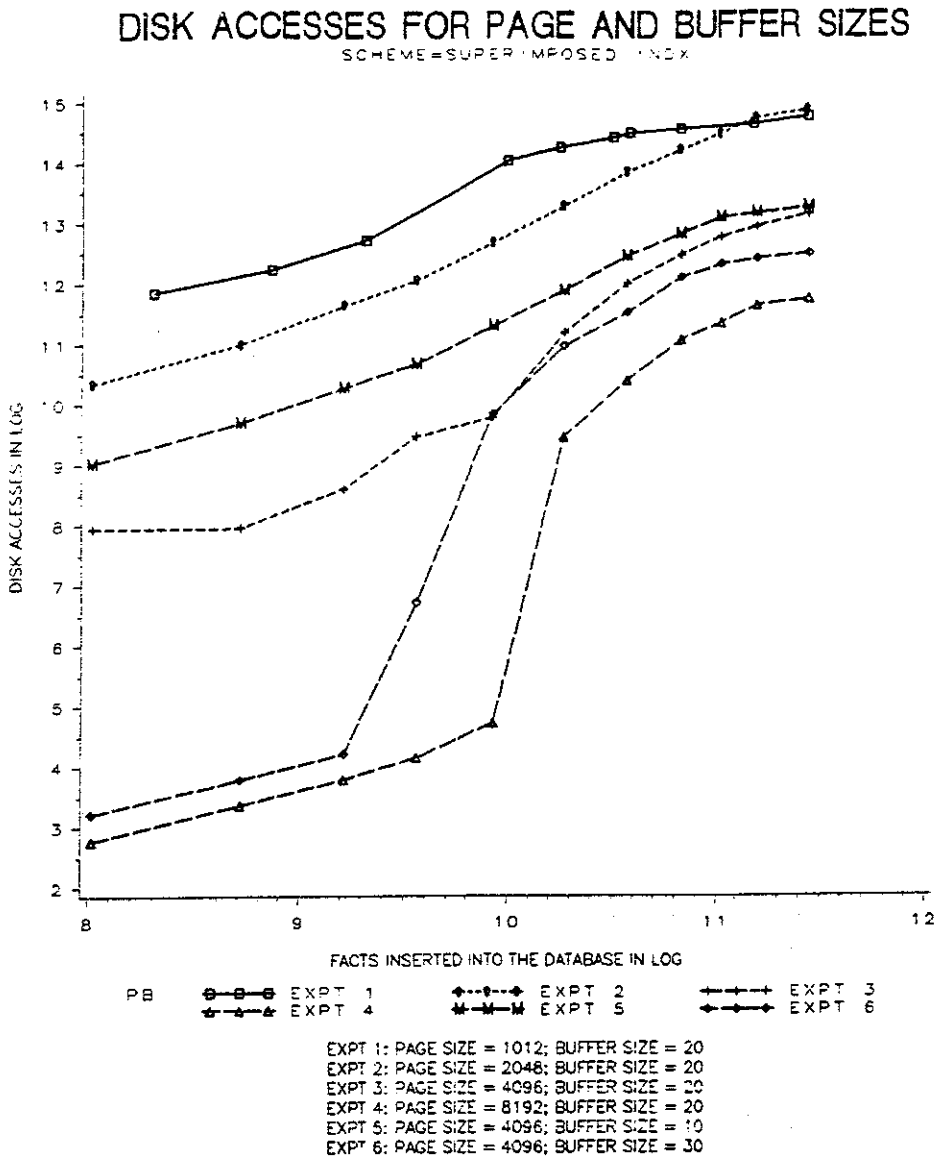

Figure 7. Disk Accesses for Different Page and Buffer Sizes (Superimposed).

Abstract

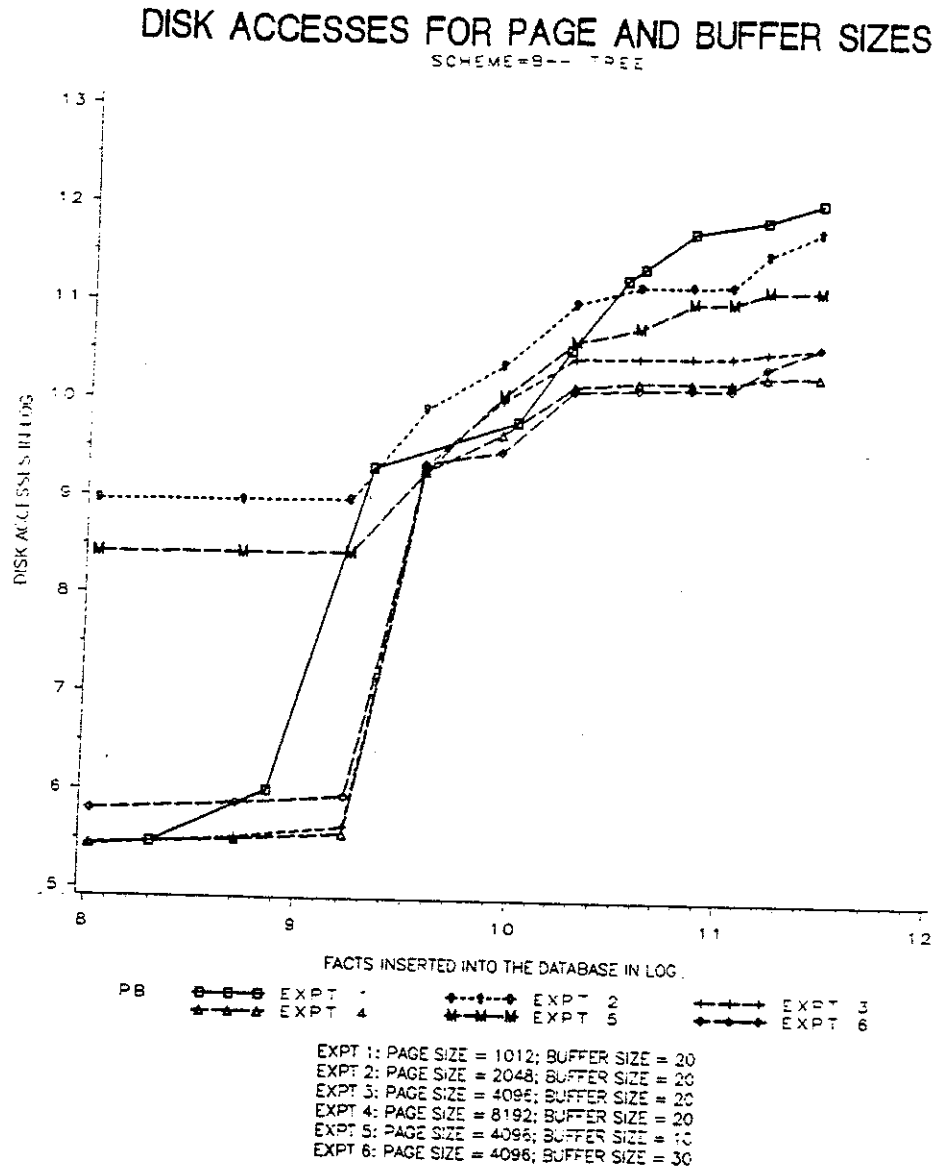

Figure 8. Disk Accesses for Different Page and Buffer Sizes (B+- tree).

Abstract

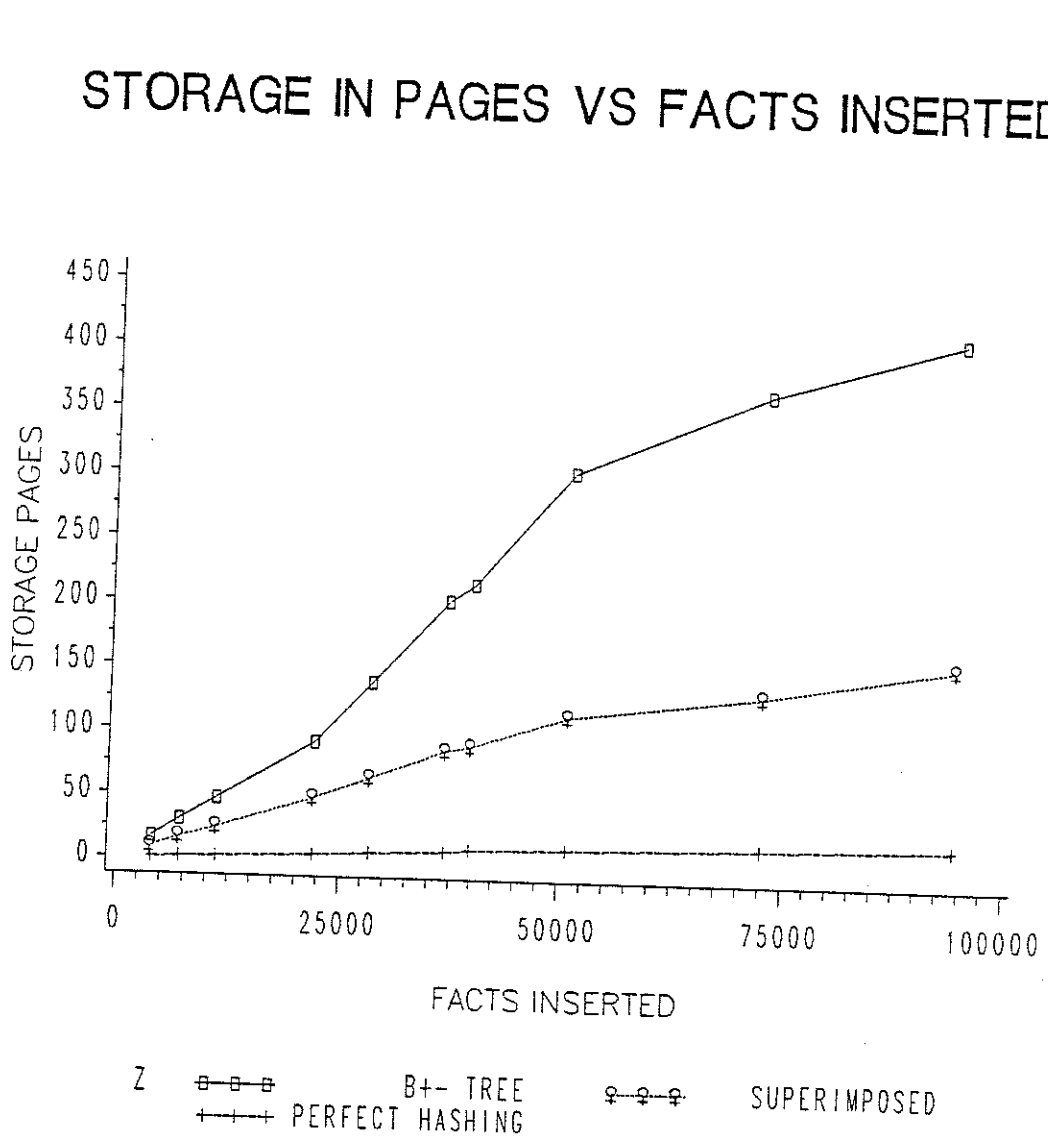

Figure 9. Storage Utilization in Pages Vs Facts Inserted.

DISK ACCESSES FOR A QUERY(1)

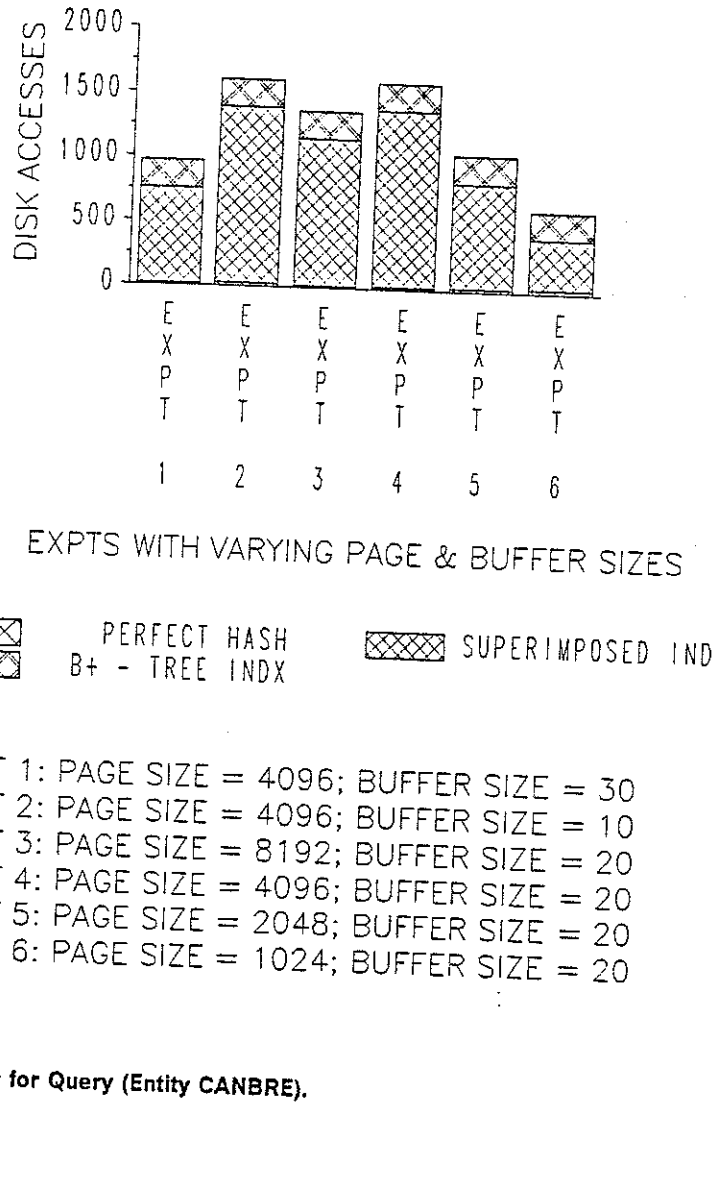

Figure 10. Disk Accesses for Query (Entity CANBRE).

DISK ACCESSES FOR A QUERY(1)

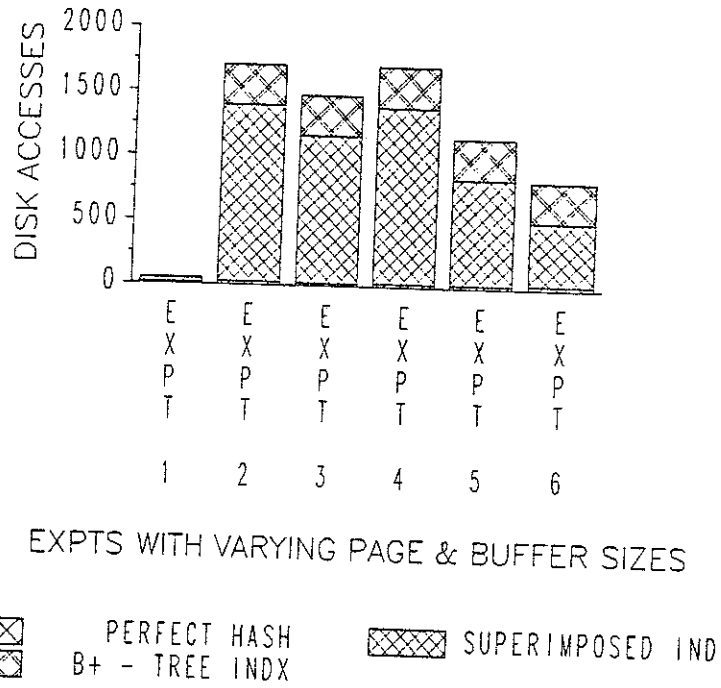

Figure 11. Disk Accesses for Query (upd ?x CALAMSG/BACC-C/D).

Indexing Scheme	Disk Accesses	Page Accesses	Insertion Time	Pages Used
Perf Hash	-	-	-	-
Superimp	32701	89288	30846	15
B+ -Tree	469	64561	5118	89

Table 2. Various Performance Measures during Insertion of 10906 facts for Page Size = 4096 & Buffers = 20.

Indexing Scheme	Disk Accesses	Page Accesses	Insertion Time	Pages Used
Perf Hash	0.0101	2.7065	0.25841	0.006613
Superimp	2.0704	5.1466	2.1828	0.0011
B+ -Tree	0.4473	4.846	0.3144	0.00673

Table 3. Various performance measures per fact inserted into the database for page size = 4096 & buffers = 20.

Indexing Scheme	Disk Accesses	Page Accesses	Search Time	Number of Comp.
Perf Hash	0.001	0.8475	0.072351	0.2350
Superimp	0.7999	1.6886	0.3045	3.241
B+ -Tree	0.5039	2.1865	0.1537	0.6746

Table 4. Various performance measures per fact searched from the database for page size = 4096 & buffers = 20.

Bibliography

1. (Baye72) Bayer, R., and McReight, E., "Organization and maintainence of large ordered indexes." *Acta Informatica* Vol. 1, No. 3 (1972), pp 173-189.

2. (Come79) Comer, D., "The ubiquitous B-tree." *ACM Computing Surveys* , Vol. 11, No. 2 (June 1979), pp 121-137.

3. (Enbo88) Enbody, R.J., and Du, H.C., "Dynamic hashing schemes." *ACM Computing Surveys*, Vol. 20, No. 2, (June 1988), pp 85-113.

4. (Folk87) Folk, M.J., and Zoellick, B., *File structures A conceptual toolkit.* Reading, Mass. Addison Wesley (1987).

5. (Gall84) Gallaire, H., Minker, J., and Nicolas, J.M., "Logic and databases: A deductive approach." *Computing Surveys*, Vol. 16, No. 2 (June 1984), pp 153-185.

6. (Haugh88) Haugh, S., Personal Communication, Department of Computer Science, Virginia Tech, (1988).

7. (Knut73) Knuth, D., *The Art of Computer Programming. Vol III, Searching and Sorting.* Reading, Mass. Addison-Wesley, (1973)

8. (Lars88) Larson, P.A., "Linear hashing with separators - A dynamic hashing scheme achieving one-Access retrieval." *ACM Transactions on Database Systems*, Vol. 13, No. 3, (September 1988), pp 366-388.

9. (Litw80) Litwin, W., "Linear hashing: A new tool for file and table addressing." *Proceedings of the 6th Conference on Very Large Databases (Montreal). Very Large Database Foundation, Saratoga, California*, (1980), pp 212-223.

10. (Rama85) Ramamohanarao, K., and Shephard, J., "A superimposed codeword indexing for very large Prolog databases." *Proceedings of the Third International Conference on Logic Programming, London (1986)*, pp 569-576.

11. (Wagn73) Wagner, R., "Indexing design considerations." *IBM System Journal*, No. 4, (1973), pp 351-367.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

3. DISTRIBUTION/AVAILABILITY OF REPORT
Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
Systems Research Center SRC-90-002

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Systems Research Center

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION
Naval Surface Warfare Center

6c. ADDRESS (City, State, and ZIP Code)
320 Femoyer Hall
Virginia Tech
Blacksburg Virginia 24061-0251

7b. ADDRESS (City, State, and ZIP Code)
Dahlgren, Virginia 22448-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
Naval Surface Warfare Center

8b. OFFICE SYMBOL (If applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)
Dahlgren, Virginia 22448-5000

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Comparative Evaluation of Indexing Schemes

12. PERSONAL AUTHOR(S)
Comarapalyam K. Guru Prasad and John W. Roach

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 2/10/89 TO 2/9/90	August 1989	34

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

An empirical experiment is reported that compares three indexing techniques used to help answer queries for a medium-sized database. The experiment compares memory and time utilization for B+ trees, superimposed codeword and perfect hashing indexing techniques. Costs for creating and querying the database are compared over a range at different page and buffer sizes for the database. We compare and contrast the advantages and disadvantages of the indexing techniques.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE (Include Area Code)

22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE