Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A UNIFIED INDEX ACCESS METHOD FOR PARALLEL DATABASE SYSTEMS

R.-C. Hu
J. W. Carlyle

June 1993
CSD-930017

A Unified Index Access Method for Parallel Database Systems

Ron-Chung Hu and *Jack W. Carlyle*

Computer Science Department
University of California
Los Angeles, CA 90024-1596

ABSTRACT

Index access methods have been well studied for conventional single-processor systems. In designing a parallel database system with multiple nodes, one immediately faces a problem: location of the index rows for non-partitioning attributes. Currently, there exist two index access methods: (1) the *local index mechanism*, in which an index record is saved on the same node as its data record, and (2) the *distributive index mechanism*, in which an index record is distributed and stored into a node based on the index attribute value. Both mechanisms are vulnerable to the effects of skewed distribution, common and inherent to real-world databases. Furthermore, neither mechanism is sufficiently robust in a dynamic environment where database contents are changing rapidly. We propose the *unified index access method*, which incorporates both local and distributive index mechanisms concurrently, and adapts to the well-performing one at run time. We perform simulation experiments to validate the effectiveness of our new index mechanism. In addition, we describe how to create and maintain the unified index access method efficiently in parallel database systems.

1. Introduction

Parallel computers composed of multiple independent processors promise to be the technological "dream machines" of the 1990's. During the past decade, database systems employing parallel architecture have increased in popularity because of their good cost/performance, high scalability, and easy availability. Commercially successful systems of this type include Teradata's DBC/1012 [Tera88] and Tandem's Non-Stop SQL [Tand88]; available research prototypes are the Gamma machine at University of Wisconsin [Dewi90] and the Bubba system at MCC [Bora90]. All these systems horizontally partition a data relation across the processors based on the range of key values [Dewi90], hashing mechanisms [Dewi90] [Tera88], or access frequencies. Based on this mechanism, such database operations as select, project, and join, can be performed in parallel and system performance can be significantly enhanced.

The index mechanism, a fast access method to locate data records quickly, is an effective alternative to the full relation scan when the number of qualified records is not large. Index mechanisms have been extensively studied in the literature, and B-tree has become the *de facto* standard index structure for file organization [Come79]. When designing the index mechanism for a parallel database system with multiple nodes, a database engineer immediately faces a difficult problem: where to put the index records? Currently, two methods exist: (1) *local index mechanism* in which an index record is saved in the same node as its data record resides, and (2) *distributive index mechanism* in which an index record is distributed and stored into a node which usually differs from the one on which its data record resides, based on the index attribute value [Tand88] [Tera88]. When a query references an index, the local index mechanism employs all nodes in execution, while the distributive index mechanism involves only those nodes containing relevant data records. In practice, either of these methods is useful for certain indexes, and the choice is dependent on such factors as number of qualified records for a query, number of nodes in a machine configuration, startup and termination costs of a task, etc.

With the intrinsically skewed distributions found in real-world databases, the number of qualified records varies greatly from value to value, which can render either of the existing methods useful for only a small subset of values for a given index. Furthermore, the number of qualified records may change significantly in a dynamic environment. In this case, an initially well-performing local index method may become a troublesome performance problem at a later time, and vice versa. In this paper, we propose a new index access method, termed *unified index mechanism*, for parallel database systems. Unlike other index methods, the unified index method takes skewed distribution into account. In particular, our new mechanism combines both local and distributive index mechanisms, and adapts to the well-performing

one in a dynamic environment.

The remainder of this paper is organized as follows. In section 2, we outline the parallel system architecture under consideration and briefly describe the skewed distribution found in real-world databases. In section 3, we describe the two conventional index methods: local index mechanism, and distributive index mechanism, as well as their execution steps in a parallel database system. In section 4, we present our new index method, *unified index mechanism*, and show how it performs adaptively at run time. In section 5, we describe our simulation experiments using CSIM and give experimental results in verifying the effectiveness of the new index mechanism. In section 6, we discuss how to create and maintain the unified index mechanism in a dynamic database environment. Lastly, section 7 comments on the user-friendliness of our new approach and draws final conclusions.

2. System Architecture and Skewed Distribution

To facilitate discussion, we follow the parallel database system architecture first outlined in [Nech84], and later used in the Bubba system [Bora90] and the DBC/1012 [Tera88]. As shown in Figure 1, there are two kinds of processors: Interface Processor (IP) and Access Processor (AP). The IPs handle the interaction with users. They receive queries from users and determine the query access plans, which are broadcast to all the involved APs. The AP is designed to have a shared-nothing structure [Ston86], i.e., each AP has its own cpu, main memory, and disk. The tuples of each relation are distributed across all the APs, based upon the values directly or the hash values of the partitioning attribute in the tuples. Each AP is in charge of the access to the tuples appearing on its disk. The AP services the select, insert, delete, and update database operations against the tables saved on disks. All processors, both IPs and APs, communicate and coordinate with each other by messages passed along the interconnection network. Since the network physically connects a large number of nodes, it must be very fast to handle the high volume of message traffic.

It is well known that the uniformity assumption is not realistic for describing the distribution of attribute values in a relation [Chri83]. Data bases often contain information describing populations of the real world, and many tuples will have the same attribute value. For instance, a relation storing all UCLA students can lead to many tuples with the same value, California, in the state attribute since more than 70% of UCLA's students are from California. This phenomenon is common and evident in many databases. It has been found that the data distributions of many real-world databases follow "Zipf's law", in which the n-th most commonly appearing value occurs with a frequency inversely proportional to n [Zipf49].

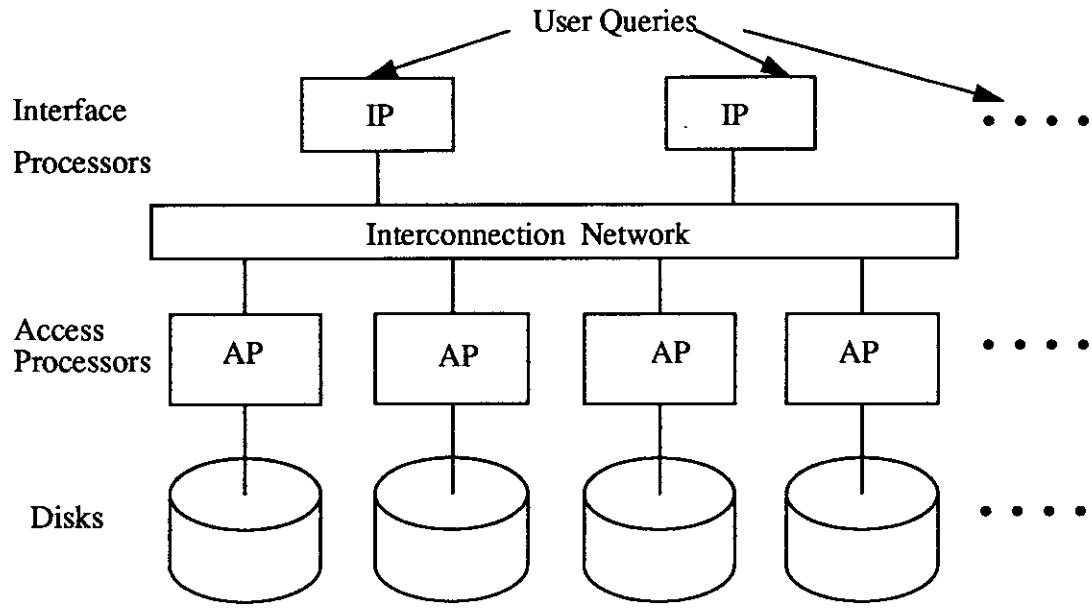

Figure 1. Parallel Database System Architecture

3. Conventional Index Methods

We describe the existing index access methods on parallel systems and their corresponding execution steps, with performance costs specified.

3.1. Local Index Mechanism

When the conventional index method is generalized to a shared-nothing parallel database system, there is a strong tendency to put the index records into the same node as their data records. This approach, termed *local index mechanism*, characterizes the index designs in several systems, including Gamma machine [Dewi90] and DBC/1012 [Tera88]. The rationale for this approach is that each node of a shared-nothing system is essentially a "computer" possessing all necessary components; hence building the index locally is a natural and straightforward generalization of conventional index methods.

For the horizontal partitioning attribute of a relation, it makes good sense to build its index locally since the system has a mechanism to determine the node in which a tuple is located, based on the attribute value. For an index built on non-partitioning attributes, retrieval operations through the local index mechanism suffer a performance impact. This is because the system has no way to tell which nodes contain the qualified index records since the index rows meeting the predicate condition are scattered over

the nodes. Hence, each user query which references a non-partitioning index must be dispatched to all nodes.

To illustrate the performance effects, suppose a relation Student, containing information for each student in a university, has the columns StudentId, Name, State (from which the student comes), etc. As shown in Figure 2, the relation is saved on a 10-node system using "StudentId" (Sid) as the partitioning attribute. An index record normally contains an index attribute value followed by a number of tuple identifiers (TupleID) of those tuples having the same value for the index attribute. The TupleID (Tid) consists of two parts: the node number and a unique number within the node. These two numbers together uniquely define the TupleID for each record within the whole parallel system. In our example, an index on the attribute "State" is built locally. When a query to find all students coming from Texas

SELECT * FROM Student WHERE State = 'TX';

is executed, it is sent to every AP to locate the index rows with value 'TX' first, then the TupleIDs of the qualified records are used to fetch the Student records from the data table.

We use an analytical model to study the performance of various index mechanisms. Suppose there exists an index with D distinct values in relation R residing on an N-node parallel system. For a condition k, there are $\{R(k)\}$ qualified tuples satisfying the condition in the whole system. In order to compare the performance difference for different index mechanisms, we detail the execution steps required for each mechanism and associate the cpu pathlength with the corresponding execution step. The cpu pathlength parameters we have defined are I_p for processing a tuple (including locating a column and comparing its value), I_c for sending or receiving a message, I_d for initiating a disk IO access, I_{move} for moving a record around in main memory, I_{start} for starting a task in an AP, and I_{term} for terminating a task in an AP. In our analysis, although we take into account the disk IOs in accessing data blocks, we will not attach the disk times to the execution steps because the required disk IOs for all the methods are identical in fetching the data blocks containing qualified records.

We now outline the execution steps for the local index mechanism after an IP receives a user query and is ready to dispatch its instructions to the APs:

1. The IP broadcasts a message to every AP.

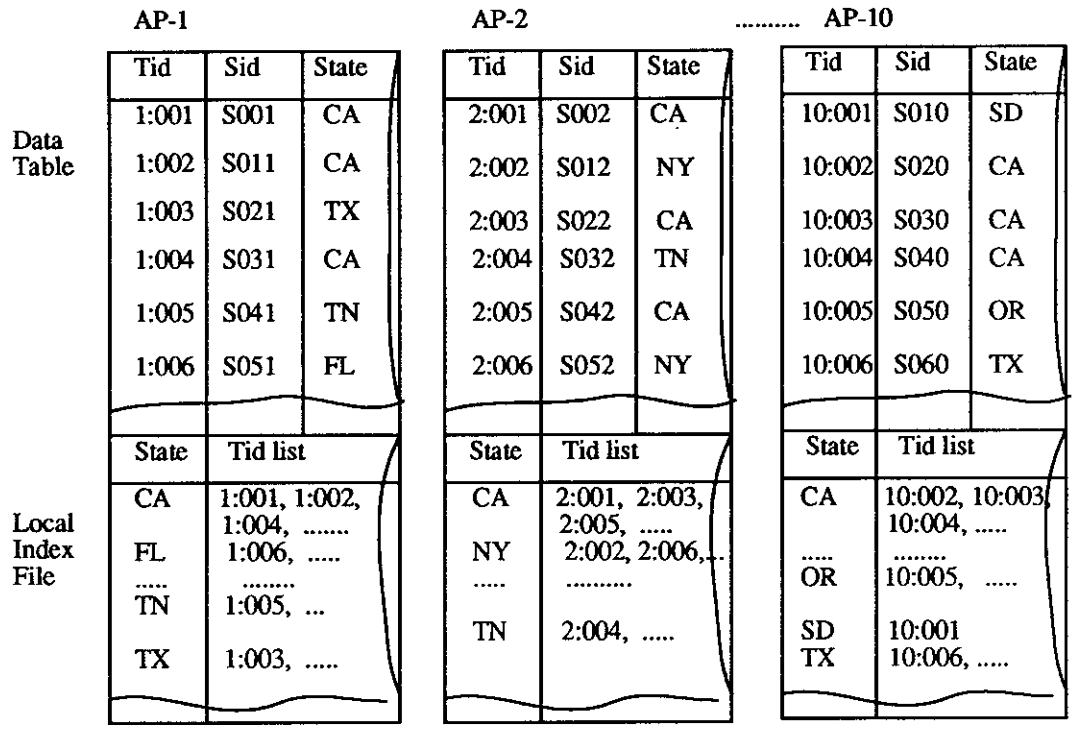

Figure 2. Local Index Built on a Non-Partitioning Attribute

2. Each AP receives a message sent from the IP. The total cost is $N*I_c$.

3. Each AP starts a task to examine its local index to find the qualified TupleIDs. Assuming the index structure is a B-tree with m entries in an index block, then the height of the B-tree is approximately $\log_m(D_i)$ for D_i distinct values in node i, and we need about $\log_2(m)$ comparisons to do binary search within an index page. Thus, the total cost for this step is: $N*I_{start} + N*I_p*\log_2(m)*\log_m(D_i)$.

4. Each AP retrieves the qualified records through the TupleIDs found in the above step. The cost to process a single record is: I_d for initiating a disk IO, $\frac{1}{2}\log_2(\{R_B\})*I_p$ for an average cost in using binary search for a tuple within a memory page buffer holding $\{R_B\}$ tuples, and I_{move} for moving the qualified tuple to an output buffer. Hence, the total cost to process $\{R(K)\}$ tuples is $\{R(k)\}*\left[I_d + \frac{1}{2}\log_2(\{R_B\})*I_p + I_{move}\right]$.

5. All the APs terminate their tasks and synchronize at the end. The total cost is: $N*I_{term} + N*I_c$.

Summing the costs in the above steps, we obtain the total cpu pathlength consumed when the local index is employed:

$$I_{local} = 2N*I_c + N*(I_{start} + I_{term}) + N*I_p*\log_2(m)*\log_m(D_i)$$
$$+ \{R(k)\}*\left[I_d + \frac{1}{2}\log_2(\{R_B\})*I_p + I_{move}\right]$$

Figure 3 draws the control flow for the execution steps using the local index mechanism. The numbers in parentheses represent the execution steps, as described above, during processing.

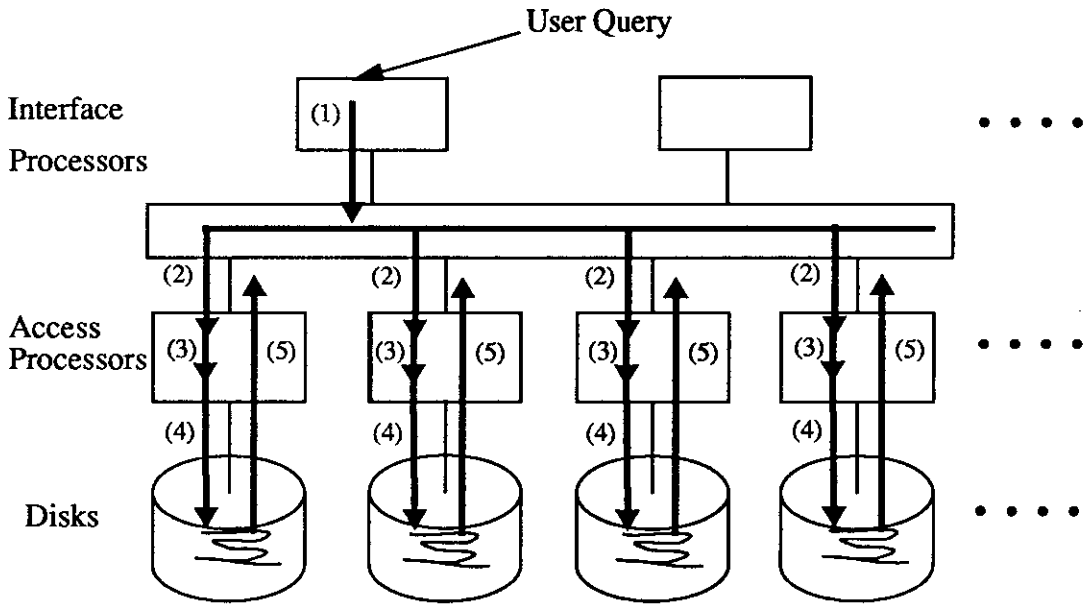

Figure 3. Execution Steps Using Local Index

3.2. Distributive Index Mechanism

As we demonstrate above, no matter how few data records qualify for the predicate condition, all nodes must participate in execution of the query when a local index is used. This situation causes a significant waste of system resources when a node participates in execution, but later retrieves no tuples from its data portion. This waste occurs because starting a task, terminating a task, and searching the index files together consume a non-trivial amount of resources [Cope88] [Ghan92]. The waste can be relatively

large for short-running OLTP (On-Line Transaction Processing) transactions, thereby leading to poor system performance, when the number of qualified records is fewer than the number of nodes. For instance, in the previous Student relation spread over a 10-node parallel system, assume there is only one student from South Dakota. When the query

SELECT * FROM Student WHERE State = 'SD';

is executed through a local secondary index on "State" attribute, all nodes will be involved in the processing. Upon completion, however, only one node will return rows, while all the other nodes perform fruitless work and retrieve nothing.

As can be seen, the local index mechanism is not effective in the above case. An efficient way to limit the number of nodes involved is to partition the index rows horizontally based on the index values. Like the way a relation is declustered, we can apply either a range partition or hash partition to distribute the TupleIDs in the index rows, based on index values [Cope88] [Tera88]. That is, all the TupleIDs of the tuples with same index value are bundled together and stored in a single node, i.e., there is only one index row for each distinct index value in the entire system. We refer to this index method as the *distributive index mechanism.* When the index is not the partitioning attribute of a relation, this mechanism normally causes TupleIDs in index rows to be saved into nodes different from the ones on which their data records reside. To clarify the difference between local and distributive index methods, we again use the Student relation saved in a 10-node system, and use "StudentId" as the partitioning attribute as an example. Figure 4 shows a distributive index built on the "State" attribute.

With the distributive index, a query referencing a non-partitioning attribute is first sent to a single node. This is because the IP can compute, based on the known partitioning strategy, and find the exact AP containing the index row having a specified value. This AP locates the TupleIDs of all qualified records, then transmits only to those APs containing the qualified records. The APs which do not contain qualified records are not involved in the execution and are free for other work. As an example, the query to find all students from South Dakota is first dispatched from an IP to AP-1 to locate the TupleIDs of all the students from state 'SD'. Since there is only one student, the only TupleID identified in the index file will direct the query to AP-10 to obtain the qualified record. Nodes AP-2 through AP-9, thus, do not participate in the query execution.

For the distributive index mechanism to be effective, the key is not to have every node involved in

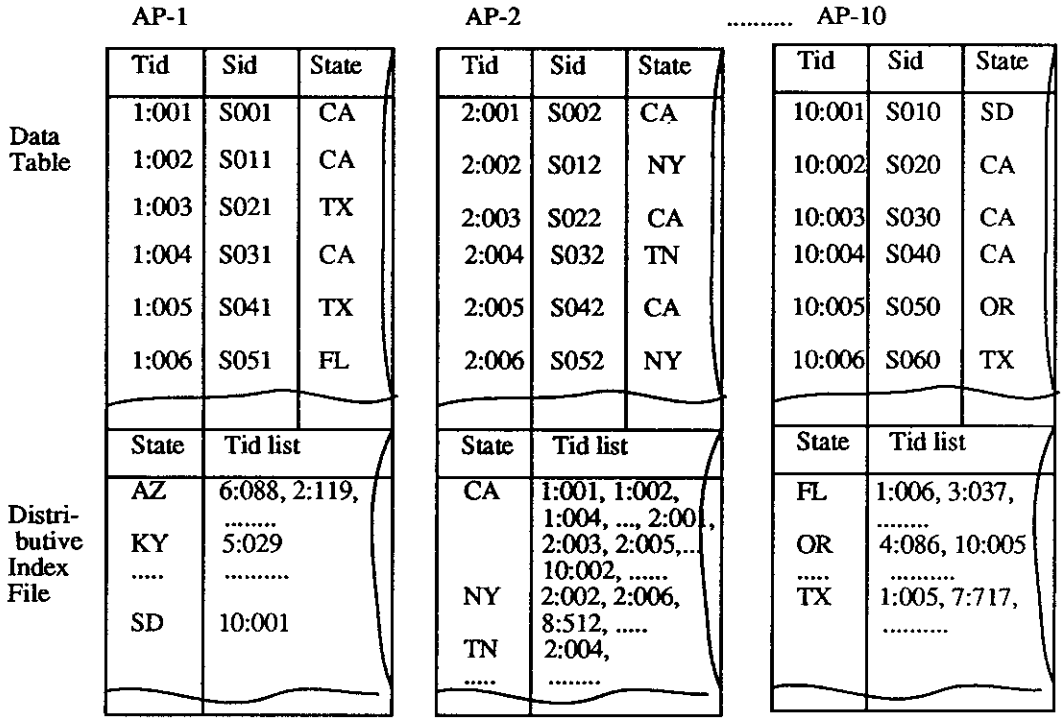

Figure 4. Distributive Index Built on a Non-Partitioning Attribute

processing. To make this happen, the number of nodes $N(k)$ holding $\{R(k)\}$ qualified records meeting a condition k must be less than the number of nodes N in the configuration. In [Hu93], we find the relationship between these three parameters as: $N(k) = N^*\left[1.0 - \left[1.0 - \frac{1}{N}\right]^{\{R(k)\}}\right]$. In the following, we detail the execution steps required for the distributive index mechanism:

1. The IP sends a message to the index-designated AP.
2. The index-designated AP receives the message and starts a task to locate the TupleIDs of the qualified records. Suppose each AP has approximately D/N index tuples, the total cost is $I_c + I_{start} + I_p * \log_2(m) * \log_m(D/N)$.
3. The index-designated AP locates and moves the TupleIDs to the output message queue, then sends them to the $N(k)$ nodes holding qualified records. Afterwards, this index-designated AP terminates its task and synchronizes. The cost is $(I_p + I_{move}) * \{R(k)\} + \left[N(k) + 1\right] * I_c + I_{term}$.

4. The APs containing qualified records receive the message and move the TupleIDs out of the input message queue. Each then starts a task within itself to be ready to fetch data records. The total cost is $N(k)*I_c + N(k)*I_{start} + I_{move}*\{R(k)\}$.

5. The APs retrieve the qualified records through the identified TupleIDs. The total cost to process $\{R(K)\}$ tuples is the same as for step 4 of the local index mechanism, i.e., $\{R(k)\}*\left[I_d + \frac{1}{2}\log_2(\{R_B\})*I_p + I_{move}\right]$.

6. All the involved APs terminate their tasks and synchronize at the end. The total cost is: $N(k)*I_{term} + N(k)*I_c$.

Summing the costs for the above steps, we obtain the total cpu service demand when the distributive index is employed:

$$I_{distri.} = \left[3N(k)+2\right]*I_c + \left[N(k)+1\right]*(I_{start}+I_{term}) + (I_p+2I_{move})*\{R(k)\}$$
$$+ I_p*\log_2(m)*\log_m(\frac{D}{N}) + \{R(k)\}*\left[I_d + \frac{1}{2}\log_2(\{R_B\})*I_p + I_{move}\right]$$

Figure 5 draws the control flow of the execution steps using the distributive index for a given condition. The numbers in parentheses represent the execution steps, as described above, during processing.

4. Unified Index Approach

We explain the rationale for proposing the unified index mechanism, then give the execution steps with performance costs specified.

4.1. Rationale

So far, we have shown that both local index and distributive index can yield good performance under certain situations. The local index mechanism is an effective alternative to the full file scan by performing index access in parallel on all nodes. Although the use of a local index involves all APs in execution, it limits the number of data blocks and tuples to be processed by the APs. However, when the number of qualified records is not large and not all nodes contain qualified records, the distributive index mechanism should be employed such that only those APs containing qualified rows are engaged in execution, hence

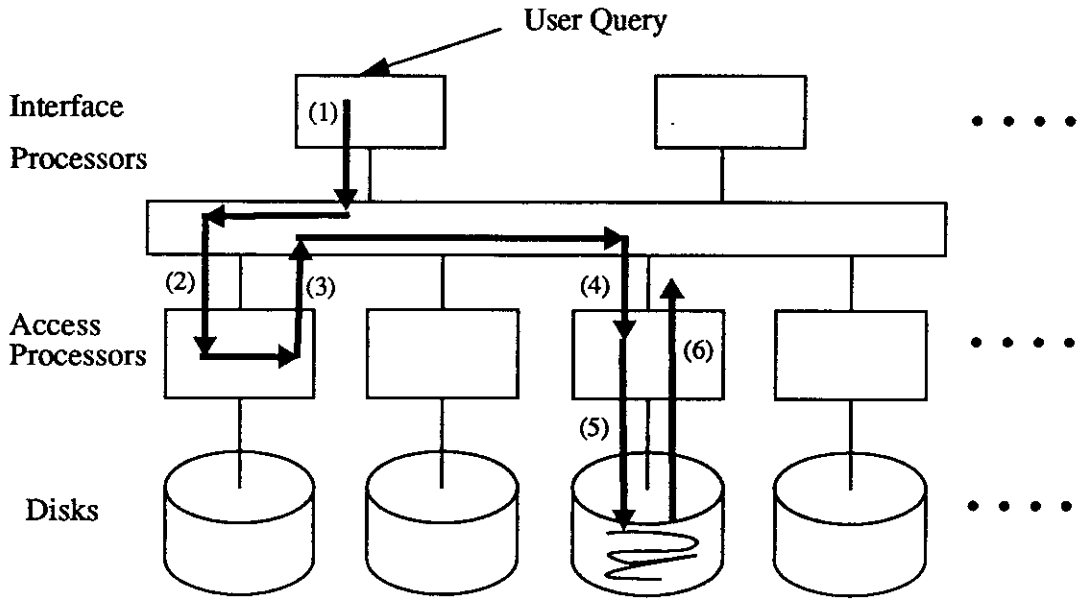

Figure 5. Execution Steps Using Distributive Index

leading to no waste of system resources from those APs which do not hold qualified records.

As described earlier, data skew effects are intrinsic and inevitable in real-world databases. A ubiquitous fact is that some values for an attribute occur more frequently than the other values. The distributive index can perform well for those values having few record occurrences, while the local index can do well for those values having more record occurrences. Neither index mechanism, however, can perform well for all the values in a single index. Moreover, database contents change quickly in a dynamic environment. After many inserts, an index value which initially has few records so that a distributive index operates well may come to have too many records. If we still have a distributive index, it not only will cause index-TupleID placement skew (i.e., many TupleIDs in the index row), but also will cause extra communication overhead in sending/receiving messages. Therefore, we believe that a fixed mechanism on a shared-nothing parallel system is not sufficiently robust for real-world circumstances. We suggest a robust index mechanism which incorporates both local and distributive mechanisms concurrently in a single index. We refer to this as the *unified index mechanism.*

The key to the unified index mechanism is to adapt to the well-performing index, either local or distributive, in a dynamic database with skewed distributions expected. That is, a unified index assumes a hybrid

form, with both local and distributive methods concurrently existing in a single index. In a unified index file, there exist two subfiles: namely a distributive index subfile and a local index subfile. In the *distributive index subfile*, there exists exactly one row for each distinct index value. Each index row contains the index value followed either by TupleIDs or a special value like "@@", but not both. If it contains TupleIDs, the regular distributive index is utilized for this index value; if the special value "@@" is found, the local index is employed for this index value. In the *local index subfile*, all values with many record occurrences have their index rows built locally. Those values which utilize the distributive index do not appear in the local index subfile. To clarify this point, we again use the Student relation, saved in a 10-node system using "StudentId" as the partitioning attribute, as an example. Figure 6 shows this unified index built on the "State" attribute.

4.2. Execution Steps

As each user query referencing a unified index comes in, the IP directs execution to the index-designated AP by assuming a distributive method first. If the distributive method is really employed for the specified index value, then the system proceeds identically as a distributive index. If the local index method is actually employed, then the designated AP will find a special value such as "@@" to inform the system that a local index is utilized for this value. Next the designated AP broadcasts an instruction to all APs to cause the system to switch to proceed in the local index approach from this point. Similar to the analysis for other index mechanisms discussed previously, we now detail the execution steps required for a unified index mechanism:

1. The IP transmits a message to the index-designated AP.
2. The index-designated AP receives the message and starts a task to locate the index row in the distributive index subfile. If the special value "@@" is found in the index row, then go to step 7. Otherwise, proceed continuously to step 3. The total cost is $I_c + I_{start} + I_p * \log_2(m) * \log_m(D/N)$.

3-6. Steps 3 through 6 correspond to Steps 3 through 6 respectively as described for the distributive index mechanism.

7. The index-designated AP broadcasts a message to every AP, then terminates itself. The cost is $I_c + I_{term}$.

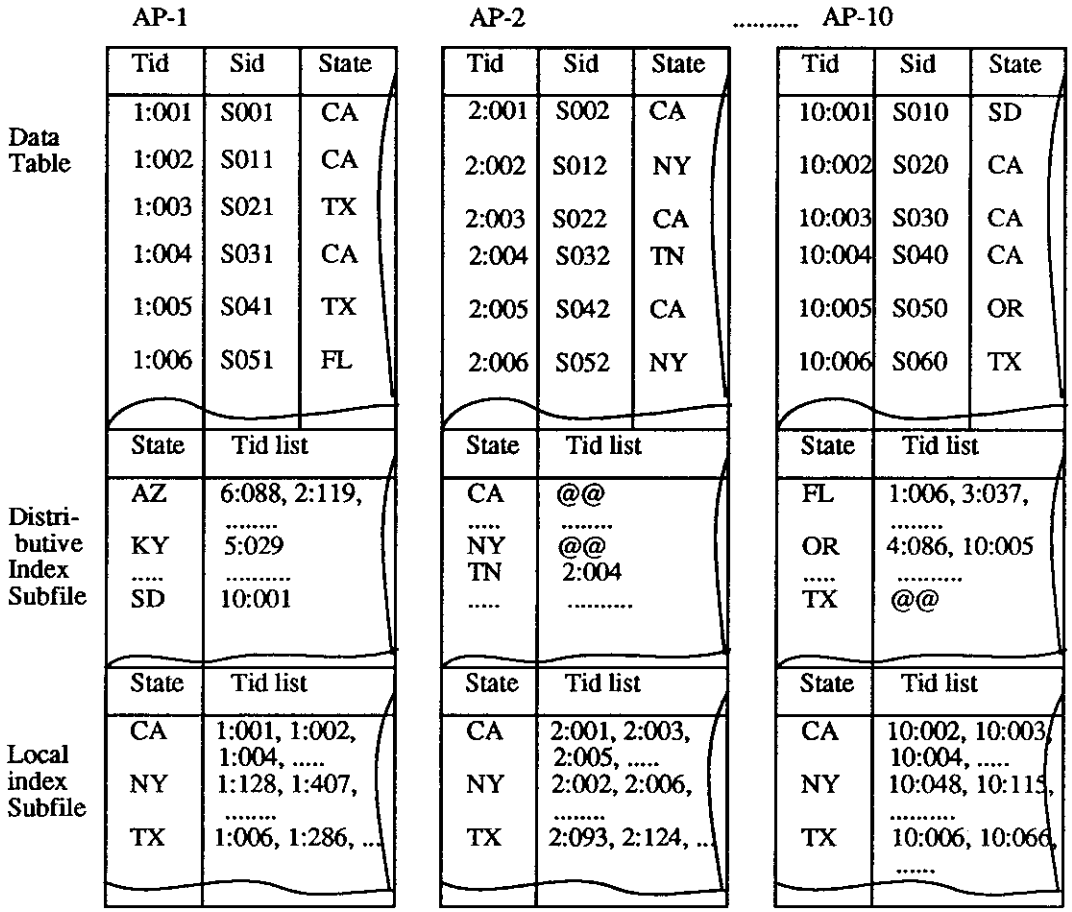

Figure 6. Unified Index Built on a Non-Partitioning Attribute

8-11. Steps 8 through 11 correspond to Steps 2 through 5 respectively as described for the local index mechanism.

The above execution steps show that a unified index mechanism does not incur any extra overhead, when compared with the distributive mechanism for a particular value, if it turns out to actually use the distributive one. Compared with the local mechanism for a value with more record occurrences, the unified index mechanism incurs an overhead because the index-designated AP must bear extra communication work, task startup and termination costs, and an additional search in the distributive index subfile. The total overhead cost for this case is:

$$I_{start} + I_{term} + 2*I_c + I_p*\log_2(m)*\log_m(D/N)$$

Compared with the index values for a fixed index mechanism improperly applied, the overhead brought by the unified index approach is more than offset by the savings in the proper and adaptive use of the well-performing mechanism. We can verify this point using simulation experiments.

5. Simulation Experiments and Results

In order to evaluate the effectiveness of the unified index approach, we established a modeling study to compare the performance of the three index mechanisms. Since the analytical model has difficulty in precisely modeling the *fork-synchronize* (also called *fork-join*) operation required for parallel executions, we chose the simulation approach in our modeling study. In turn, we describe the performance model with queueing network specified, simulation experiments having numerous parameter values, and experimental results with different workloads.

5.1. The Performance Model

We evaluate system performance in a multi-user environment. The query used in our study is a simple Select statement to retrieve records through a secondary index built on a non-partitioning attribute:

SELECT column_name FROM relation WHERE index = :input_value;

The provided index value (i.e., input_value) determines the number of the qualified records $\{R(k)\}$ based on the data distribution of a relation. Since the above query, a typical on-line retrieval transaction, normally has a very short response time, we follow the standard OLTP benchmark and utilize system throughput as the primary performance metric for our evaluations [TPC90].

To better reveal the relative throughput of various index mechanisms, we employ a closed queueing network model with a batch workload [Lazo84]. The batch workload is usually characterized by a fixed number of active jobs (or Multi-Programming Level) with zero think time. Figure 7 illustrates the simulation queueing model for executions utilizing the local index mechanism. While keeping track of the Multi-Programming Level (MPL), the IP supplies transactions and directs all the APs to retrieve records for a given index value. When an IP broadcasts a query under the local index method, the query is actually split (or forked) into a number of sub-queries equal to N, the number of access processors in the system. Each sub-query is an independent job in an individual AP queueing model. As shown in Figure 8 for a typical AP node, a sub-query cycles between the CPU queue using the CPU, and the Disk queue using the disk. In an AP node, while the disk is modeled as a service center with FCFS (First-Come-

First-Served) service discipline, the CPU is modeled as a round-robin service center with time slice set to 3 milli-seconds. We also compute the memory hit ratio for a data relation, then determine whether or not a data page is memory resident. If a data page is not in memory, a disk access in random mode is necessary to bring in the data block. When a sub-query finishes its work within an AP, it moves to the "Synchronize" node, where it waits for all its siblings to complete.

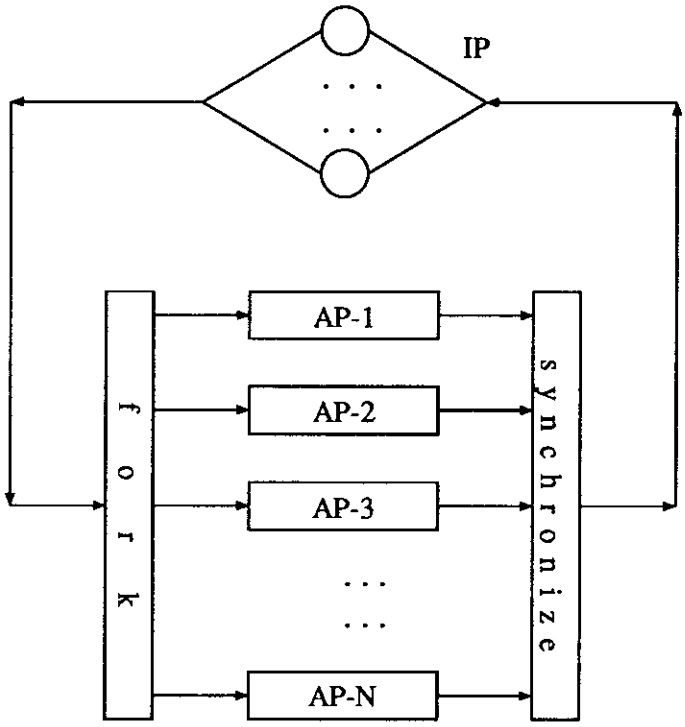

Figure 7. Queueing Model for Local Index Mechanism

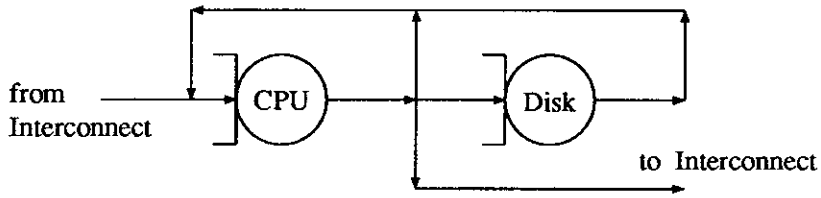

Figure 8. Queueing Model for a Typical AP Node

Figure 9 depicts the simulation queueing model for executions using the distributive index mechanism.

The IP first passes the query to the index-designated AP, which then splits the query into a number of sub-queries equal to $N(k)$, the number of access processors containing the qualified records. The sub-queries are sent in turn by the index-designated AP to only those APs involved in the processing. Later, only those APs, not all the APs in the system, travel to the "Synchronize" node to wait for their sibling sub-queries to finish. Our simulation queueing models closely follow the execution steps of their index mechanisms. Also, in a separate analysis, we found that on-the-wire interconnect bandwidth is definitely under-utilized for our workload, allowing us to ignore it. However, the to-wire and from-wire communication costs are included as processor work. Moreover, we simplify the IP as a delay center because there is no requirement to have an IP in a parallel database system [Hu93]. In one sense, our model is simple and may yield optimistic results, but we use it only for predicting the relative throughput of the different index methods.

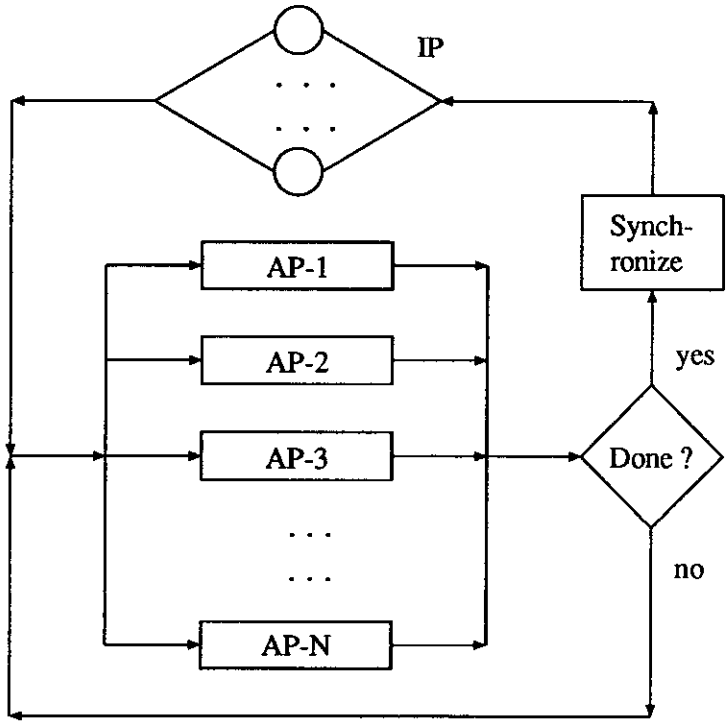

Figure 9. Queueing Model for Distributive Index Mechanism

5.2. Simulation Experiments

We employed the CSIM package, a process-oriented discrete-event simulation package for use with C programs [Schw90], to develop our simulation program. Although simple yet sufficient, the CSIM has a library of routines which implement all of the necessary operations. Because it is process-oriented, it provides a good simulation environment for parallel programs. Our simulation program is outlined in [Hu93]. In our experiments, the values of the performance parameters, chosen to be representative of high-performance technology available today [Cope88] [Dewi90] [Ghan92], are: 500,000 tuples in relation R; Rows of 200 bytes long each saved in 4-Kbyte blocks; N = 32 nodes in the system; 4 MIPS rating for each CPU; 3 megabytes data buffer space per AP; 25.93 milli-seconds for a disk block access in random mode; I_p = 2000 instructions for processing a tuple; I_c = 2000 instructions for sending or receiving a message; I_d = 4000 instructions for initiating a disk IO; I_{move} = 4000 instructions for moving a tuple; I_{start} = 5000 instructions for starting a task on an AP; I_{term} = 10000 instructions for terminating a task on an AP; and m = 200 entries in an index block.

The experiments were performed on a Sun Microsystem's SPARCstation SLC rated at about 12.5 MIPS with 16 mega-bytes main memory space. Limited by available system resources, we set N = 32, representing a medium-sized configuration. Each data point, simulating thousands of transactions, ran long enough to achieve a stable system performance. We obtained the system throughput by excluding both initial and final transients.

5.3. Experimental Results

In order to justify the existence of each of the alternative index methods, we choose three workloads for this performance evaluation. In the first case, we have a workload of single job class with all jobs returning a small number of tuples such that only a small subset of nodes will hold the qualified records. In the second case, we have a workload of single job class again with every job returning a large number of tuples so that all the nodes must get involved. The third case is a mixed workload with some jobs retrieving few tuples and others selecting many rows. In fact, all the jobs come from the query defined in Section 5.1. The number of tuples returned is determined by the data distribution of an attribute. While the single job class workload implies a uniform distribution, the mixed workload case typifies a real world situation, i.e., an uneven data distribution for a given index attribute.

In Figure 10, we present the system throughput as a function of the multi-programming level with each

query returning 5 records. Since the number of qualified records, $\{R(k)\} = 5$, is much smaller than the number of nodes ($N = 32$), only a small subset of nodes will contain the qualified records. The local index always involves all the APs no matter how many records we may get for a given index value, hence quickly saturating the system at low MPL degree. At an MPL of 5, the CPU for each AP is almost 100% utilized, causing throughput to level off. As for the distributive mechanism, it calls only those nodes containing useful records for operation, freeing other nodes for other work, thereby producing much higher throughput. At a multi-programming level of 80, the throughput for the distributive mechanism is five times greater than that of the local mechanism. It is essential to observe, however, that the local index actually outperforms the distributive index at very low multi-programming levels (MPL = 1 or 2). This occurs because of a communication hot spot at the index-designated AP, which must send messages in turn to all the involved APs. This hot spot effect is noticeable at low degree MPL in which only a small degree in concurrency can be realized, leaving most nodes under-utilized with the distributive index mechanism. Finally, the throughput for the distributive and the unified index methods is identical for all multi-programming levels since the unified mechanism actually adapts to the distributive strategy without incurring any overhead for this workload.

Figure 11 shows the throughput for a single job class workload with every query retrieving 500 records. With this workload, the number of qualified records, $\{R(k)\} = 500$, is much larger than the number of nodes ($N = 32$). Almost certainly each node holds some qualified records and is involved in the job operation. This workload represents the best case scenario for the local index method and the worst for the distributive index method. The distributive index has the index-designated AP identify the TupleIDs of the retrieved records, then inform each AP individually. This approach not only incurs extra overhead by notifying all the nodes, but also causes a communication hot spot, thereby slowing down the job response time. Again, the local index quickly saturates the system with an MPL equal to 5, but sustains a higher throughput than the distributive index for all the multi-programming levels. At MPL equal to 30, the distributive index provides a throughput at 7.2 queries per second, while the local index doubles the throughput to 14.2 queries per second. As for the unified mechanism, it maintains the throughput within 3% of that of the local mechanism. While the unified index has to search the index-designated AP first, then switch to the local index mechanism to fetch records, the execution time for this workload in retrieving 500 records is high enough to render the overhead insignificant.

In the third experiment, we employ a mixed workload with queries returning the following number of records with equal chance: 0, 5, 10, 15, and 500 respectively. Since a difference in the number of tuples

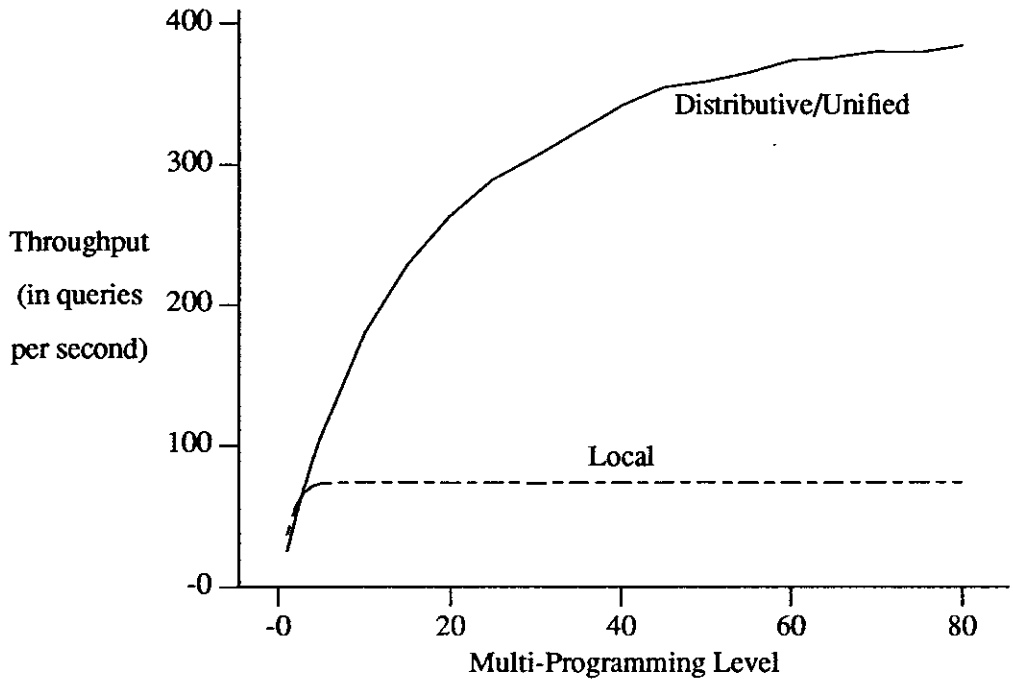

Figure 10. Throughput for Queries Returning 5 Tuples (Single Workload)

retrieved implies a distinguishable amount of system resources consumed, we essentially have a mixed workload with 5 job classes. Given the resource constraints, it is time prohibitive to simulate every point in a Zipf distribution. Therefore, we only selected 5 representative job classes in this experiment. The job classes of returning 5 and 500 records were included since we knew their performance from the previous experiments. We observed that, in decreasing order, the frequency in a Zipf distribution quickly drops with most index values having low frequency (or few records). In order to maintain the flavor of the Zipf distribution, we included two job classes of retrieving 10 and 15 records to represent those index values having few records.

The job class of returning zero records deserves some explanation. When a query retrieves nothing (or zero tuples), it often means either wrong or non-existent input value has been provided. We decided to include this case for realism because, for example, typing errors are common for most human beings, and also because a user with little or no knowledge about the database contents may easily supply meaningless input values to the query. It should be noted that retrieving no row represents the best case scenario for both distributive and unified index methods, since both involve exactly one node, the index-designated AP, to quickly verify the non-existence of the input value. At the same time, retrieving no

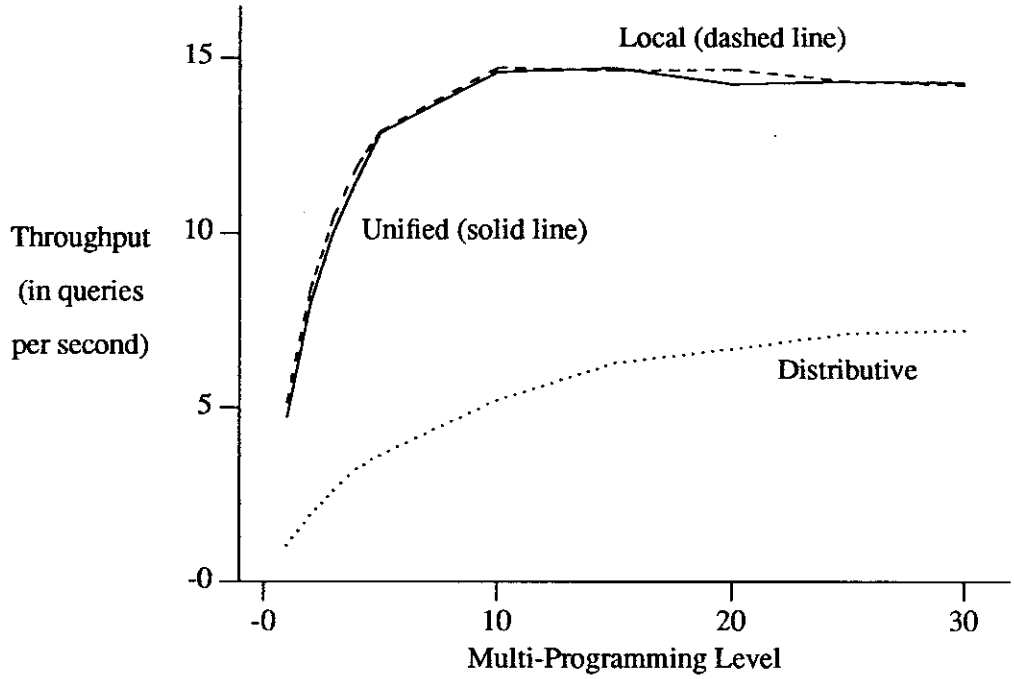

Figure 11. Throughput for Queries Returning 500 Tuples (Single Workload)

row represents the worst case scenario for the local index, because all nodes are involved in searching their own index files and none will find a qualified record.

The performance of the three index mechanisms for this mixed workload is presented in Figure 12. While the fixed distributive index can quickly process the queries returning 0, 5, 10, 15 rows, it will perform poorly in processing the 500-row case. On the other hand, the fixed local index method may execute efficiently in retrieving 500 tuples, but it will waste system resources for other cases. Neither index can work efficiently for all the cases in this mixed workload. As for the unified index mechanism, it actually adapts to the local mechanism when running queries returning 500 tuples and defaults to the distributive mechanism for the others. Except at MPL equal 1 or 2, where the communication hot spot effect impacts somewhat, the unified index mechanism provides the best performance for all other multiprogramming levels. At a multi-programming level of 30, for example, the unified index outperforms the fixed local index by 53%, and betters the fixed distributive index by 73%. We want to emphasize the significance of this result as various skew effects are inevitable in real-world applications. Whichever mechanism performs best in the mixed workload is the real winner.

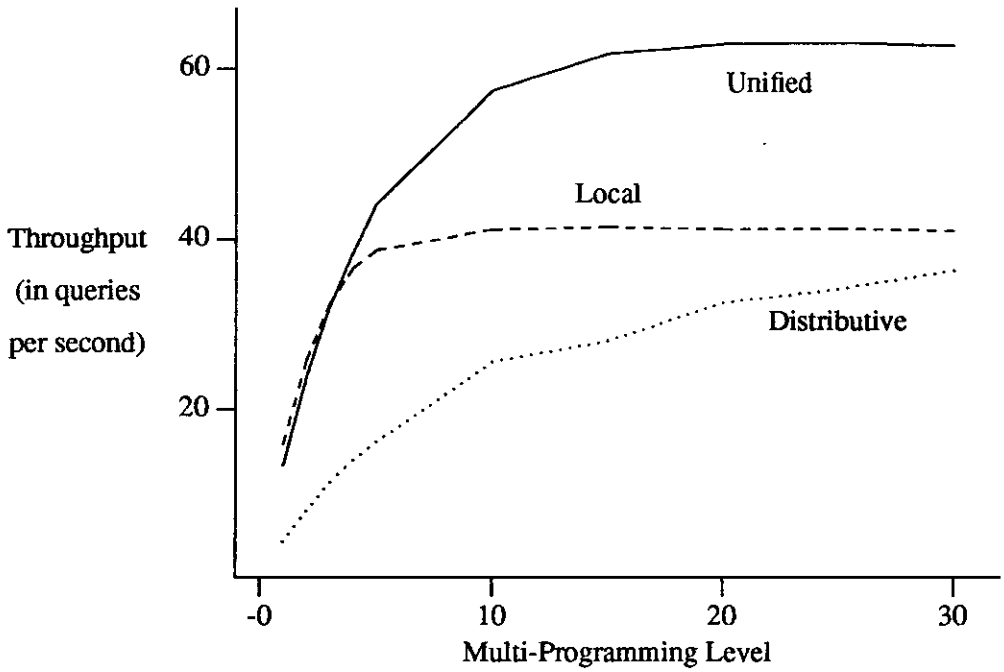

Figure 12. Throughput for Queries of a Mixed Workload (5 Job Classes)

6. Creation and Maintenance

In creating a unified index, the choice to use a local mechanism over a distributive one depends on the number of record occurrences for a given index value. For a given set of values for the system parameters N, I_p, I_c, I_{start}, I_{term},..., etc., we can compute the frequency threshold using the formulas derived for I_{local} and I_{distri}. For a given index value, the local mechanism should be employed if the number of record occurrences having this value is larger than the frequency threshold; otherwise, the distributive mechanism is utilized. Figure 13 gives a high level illustration of setting a frequency threshold θ to separate the two mechanisms for a unified index with Zipfian data distribution. It should be clearly pointed out that the frequency threshold always depends on the efficiency of a system's implementation. Besides, for a given implementation, its threshold is not an absolute value true for any configuration. Instead, its value should be determined relative to the size N of a system configuration, i.e., the number of APs. For instance, 50 records have the same index value in a relation. For this particular index value, a distributive index mechanism is good for a 100-node system, but bad for a 10-node system.

Once the frequency threshold is determined, a unified index can be created via a multi-phase operation.

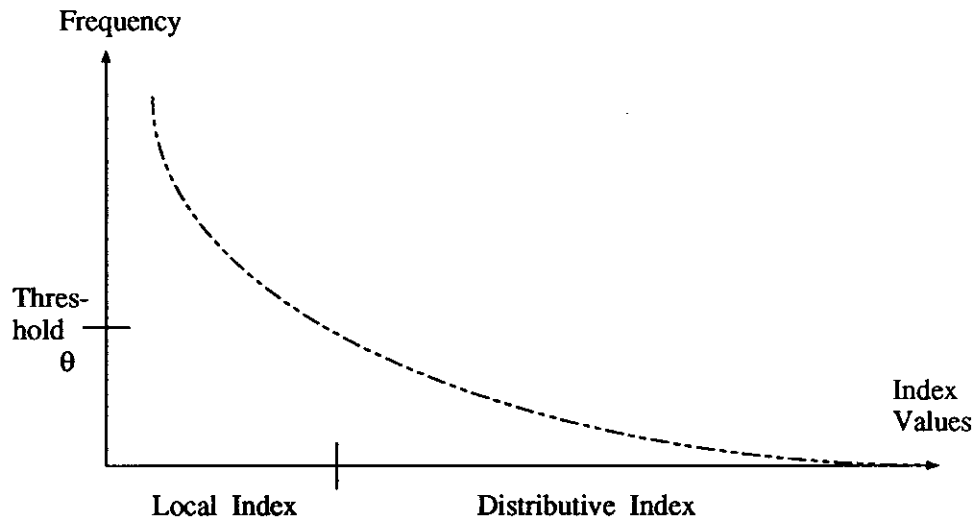

Figure 13. Zipfian Data Distribution on a Unified Index

We describe each phase as follows:

1. In phase one, since the data records are partitioned across all the APs, each AP must group together the TupleIDs of the records having the same index value. For every index value, there should be a counter to record its frequency within that node.
2. In phase two, the system aggregates the total frequency for each index value.
3. In phase 3, a local index method is built for an index value with its frequency higher than the computed threshold θ. A distributive method is used otherwise.

In a dynamic environment, there are many inserts, deletes, and updates to the database contents at a fast speed. Apparently, a system needs to monitor the frequency for each index value whenever there is a change affecting the index. A database system normally maintains a counter in the index row to track the number of tuples having the same index value. For an index value with a local mechanism built, the system needs to convert it into a distributive index when the number of record occurrences drops below the frequency threshold. The index conversion involves these steps: (1) each AP sends the TupleIDs for a given index value to the index-designated AP, (2) each AP removes its local index row for the given value, and (3) the index-designated AP collects all the TupleIDs for the given index value, then sorts them before writing to a distributive index row. Clearly, the conversion is an expensive and IO-intensive

operation since all the changes among all APs must be written to disks. With the Write-Ahead-Logging protocol normally implemented in a DBMS, the number of disk IOs required for the conversion is at least $2*N$.

Figure 13 shows a single frequency threshold when an index is initially created. For an index value with its frequency around the neighborhood of the threshold, it may convert from local to distributive when a single record is deleted, then convert the other way around immediately when another record is inserted. Although the index conversion occurs on the affected index value individually, too many conversions back and forth can consume significant system resources. To circumvent this problem, we propose a *two-threshold-mechanism* in which there are two frequency thresholds $\theta 1$ and $\theta 2$. The range between $\theta 1$ and $\theta 2$ is called as *transition range*. Threshold $\theta 1$, which is the lower bound of the transition range, is equal to the default threshold θ. Threshold $\theta 2$, the upper bound of the transition range, has a value (about 20% to 50%) greater than $\theta 1$, depending on the intensity of update operations. Figure 14 shows two thresholds defined on a unified index.

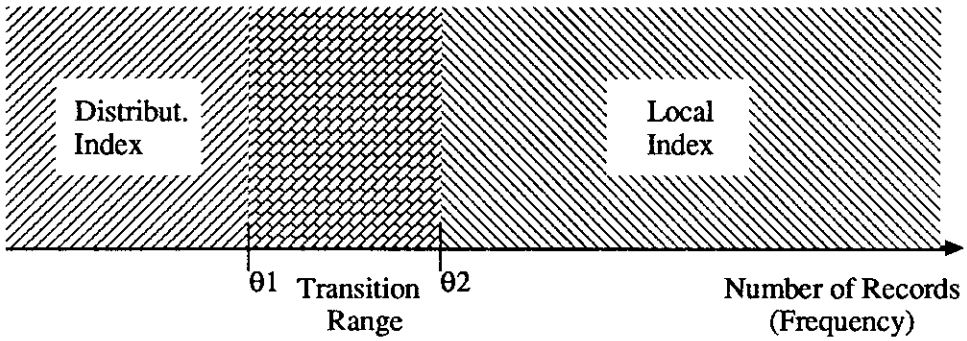

Figure 14. Two-Threshold-Mechanism on a Unified Index

When the two-threshold-mechanism is employed in a unified index, an index value enters the transition range when its frequency changes to fall between thresholds $\theta 1$ and $\theta 2$. Its index method stays the same one as it enters the transition range. Its index method converts only when its frequency falls outside the transition range and a different method is required by the system. We use the following example to illustrate the savings by the two-threshold-mechanism.

Example 1:

Assume a parallel system has its performance parameters as defined in section 5.2. For such a system, we compute its frequency threshold to be $\theta \simeq 60$. We therefore set $\theta 1 = 60$ and $\theta 2 = 78$ for the unified index mechanism [Hu93].

By counting the disk IOs required for both data records and logging records, we estimate 3 disk IOs for any insert/delete under a local mechanism, and 4 disk IOs for an insert/delete under a distributive mechanism.

Suppose a specific index value is initially based on a distributive mechanism with frequency at 59. We now assume a hypothetical set of 100 insert/delete transactions (60 inserts and 40 deletes) affecting this index value. The sequence of the transactions is: insert and delete alternate in the first 20 operations, inserts only in the 2nd 20 transactions, deletes only in the 3rd 20 transactions, again insert and delete alternate in the next 20 operations, and inserts only in the last 20 transactions. After all the changes, the index value will have a local mechanism built for it.

If we employ the single threshold in the index conversion, we have a total of 41 conversions, and 20 transactions applied to distributive index and 80 transactions applied to local index. If we employ the two-threshold-mechanism in the index conversion, we have only 3 conversions, with 80 and 20 transactions applied to distributive and local mechanisms respectively, since our two-threshold-mechanism does not require a conversion until the second threshold is hit. While the single threshold incurs a total of 2,944 disk IOs, the two-threshold-mechanism has only 572 disk IOs in total, corresponding to a 80% reduction in disk block accesses.

7. Conclusions

In this paper, we have studied two conventional index methods, i.e., fixed local index and fixed distributive index, used with parallel database systems. We have demonstrated that both index mechanisms are useful for certain indexes, depending on the number of qualified records, the size of a machine configuration, and other system parameters. Due to the intrinsically skewed distributions of data in real-world databases, some values for an index occur much more frequently than other values. Restricted by their fixed index accesses, conventional index mechanisms are not sufficiently flexible to always yield good performance.

To accommodate the inevitable skew effects, we propose the unified index mechanism which incorporates both local and distributive mechanisms concurrently in a single index. We have designed the unified index file and provided the execution steps. As demonstrated by the simulation experiments, the unified index mechanism can bring significant performance improvements over a fixed local or a fixed distributive index mechanism especially for mixed workloads caused by real world's skewed distributions. We also have shown how to create and maintain the new index method using the two-threshold-mechanism for a dynamic database environment.

In addition to performance improvement, another benefit of the unified index mechanism is its potential for user-friendliness. For instance, Teradata's DBC/1012 supports 3 fixed index mechanisms: local index (or known as Non-Unique Secondary Index in Teradata's terminology), unique distributive index (or known as Unique Secondary Index), and non-unique distributive index (or known as Hashed Non-Unique Secondary Index) [Nech86] [Tera88]. These index options impose an extra burden on users in designing their databases, which is difficult in a dynamic environment. Our unified index mechanism, employed in this situation, would provide just one simple form for the Create Index statement for users. The actual determination is made by the system, which adapts to either local or distributive mechanism to deal with the skew effects. Furthermore, our unified index robustly adjusts to the proper method using the two-threshold-mechanism in case of intense changes. In this manner, users need not worry over index tuning in future maintenance. The ease in creation and maintenance implies that our new index mechanism potentially affords a higher-level of user-friendliness. With the ever increasing cost of human manpower, our unified index mechanism seems to point toward a more cost-efficient parallel database system.

8. References

[Bora90] Boral, H., W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, A Highly Parallel Database System", IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, 1990.

[Chri83] Christodoulakis, S., "Estimating Record Selectivities", Information Systems, Volume 8, No. 2, 1983.

[Come79] Comer, D., "The Ubiquitous B-Tree", ACM Computing Survey, Vol. 11, No. 2, 1979.

[Cope88] Copeland, G., W. Alexander, E. Boughter, and T. Keller, "Data Placement in Bubba", Proceedings of ACM SIGMOD International Conference on the Management of Data, 1988.

[Dewi90] DeWitt, D.J., S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The Gamma Database Machine Project", IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, 1990.

[Ghan92] Ghandeharizadeh, S., D.J. DeWitt, and W. Qureshi, "A Performance Analysis of Alternative Multi-Attribute Declustering Strategies", Proceedings of ACM SIGMOD International Conference on the Management of Data, 1992.

[Hu93] Hu, R.-C., "Investigating Skew Effects in Shared-Nothing Parallel Database Systems", Ph.D. dissertation, Computer Science Department, UCLA, February 1993.

[Lazo84] Lazowska, E.D., J. Zahorjan, G.S. Graham, and K.C. Sevcik, "Quantitative System Performance: Computer System Analysis Using Queueing Network Models", Prentice-Hall Inc., Englewood Cliffs, NJ., 1984.

[Nech84] Neches, P., "Hardware Support for Advanced Data Management Systems", IEEE Computer, Vol. 17, No. 11, 1984.

[Nech86] Neches, P., "Teradata Corporation Presents the Data Base Computer DBC/1012", Computer Science Department Seminar, UCLA, May 1986.

[Schw90] Schwetman, H., "CSIM Reference Manual (Revision 14)", Microelectronics and Computer Technology Corporation, Austin, TX., 1990.

[Ston86] Stonebraker, M., "The Case for Shared Nothing", IEEE Database Engineering, Vol. 9, No. 1, 1986.

[Tand88] Tandem Performance Group, "A Benchmark of NonStop SQL on the Debit-Credit Transaction", Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1988.

[Tera88] Teradata Corporation, "DBC/1012 Data Base Computer Concepts and Facilities", Teradata document C02-0001-05, Los Angeles, CA., 1988.

[TPC90] Transaction Processing Performance Council (TPC), "TPC Benchmark B Standard Specification", August, 1990.

[Zipf49] Zipf, G.K., "Human Behavior and the Principles of Least Effort" Addison-Wesley Publishing Company, Reading, MA., 1949.