Two Techniques for On-Line Index Modification in Shared Nothing Parallel Databases

Kiran J. Achyutuni

Informix Software Inc.
4100 Bohannon Drive
Menlo Park, CA 94025
e-mail:kiranja@informix.com

Edward Omiecinski Shamkant B. Navathe

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
e-mail: {edwardo,sham}@cc.gatech.edu

October 28, 1995

Automatic tuning of data placement is crucial for sustaining the performance of a shared nothing parallel database system. Whenever data is moved across nodes in the parallel database system, the indexes need to be modified too. Index modification overhead can be quite severe because there can be a large number of indexes on a relation. In this paper, we study two alternatives to index modification, namely OAT (One-At-a-Time page movement) and BULK (bulk page movement). OAT and BULK are two extremes on the spectrum of the granularity of data movement. OAT and BULK differ in two respects: first, OAT uses very little additional disk space (at most one extra page), whereas BULK uses a large amount of disk space. Second, BULK uses sequential prefetch I/O to optimize on the number of I/Os during index modification, while OAT does not. Using an experimental testbed, we show that BULK is an order of magnitude faster than OAT. In terms of the impact on transaction performance during reorganization, BULK and OAT perform differently: when the number of indexes to be modified is either one or two, OAT has a lesser impact on the transaction performance degradation. However, when the number of indexes is greater than 2, both techniques have the same impact on transaction performance.

1 Introduction

Placement of data in a shared nothing parallel database system is crucial to the performance of the database system. A database is inherently dynamic, i.e., its data and access characteristics change often, the workload changes, and the relations grow or shrink. The placement of data is dependent on these dynamic factors. Consequently, there is no one placement of relations that is optimal for the lifetime of the database system. In order to sustain the performance of the system, data has to be frequently moved across disks in order to load balance the disks [19, 20].

It is common that relations have many indexes on them. For example, Sybase SQL server allows upto 250 indexes on a relation [3]. Whenever data is moved across nodes, the indexes on the relations have to be modified too. For example, if 10000 records are to be moved, and there are 10 indexes on the relation, then it amounts to 100,000 index modifications, which can require significant I/O and processing cost. Furthermore, the reorganizer has to delete 100,000 entries from the indexes at the source node, and insert 100,000 entries at the destination node. If one includes communication overhead, contention for hot spots like the higher levels of the indexes, the index overhead to make all the index changes can be quite significant. Since it is highly desirable for a database system to have on-line tuning capability [2, 16, 10, 18], on-line index modification can degrade the performance of the transactions significantly. Hence, it is of crucial importance to carefully study the various design possibilities in on-line index modification algorithms. This paper is a first step in making a performance evaluation of on-line index modification algorithms.

One straightforward approach to index modification is to rebuild the indexes on-line [17]. That is, build a new index, and then drop the existing index. However, this approach can be extremely expensive and quite unnecessary for the following reasons:

1. index build requires a scan of the data followed by a sort step. This is expensive. Further, it is also unnecessary because most of the information is already available in the existing index.

2. dropping and recreating indexes can change their object ID. This causes a recompilation of stored procedures [3], which is unnecessary.

Therefore, we do not consider this approach to index modification in this paper. Instead our approach is to design methods that make modifications to the existing index.

There are essentially two distinct ways to move the data and modify the indexes. These two techniques are the two extremes on the spectrum of granularity of data movement. The first technique moves one date page at a time, and modify the indexes for records in that datapage. It is an incremental approach, and needs very little additional disk space. The second technique makes a copy of the entire chunk of data that is to be moved at the destination node, and then modify the indexes at the source and the destination node. Though there are two copies of data, updates are allowed on only one copy. This is a file reorganization approach, and needs twice the disk space occupied by the data to be moved. We will refer to the first technique as OAT for one-page-at-a-time, and the second technique as BULK. In this paper, we will compare OAT and BULK on the following metrics: the time to complete reorganization and the impact on transaction response time and throughput.

The model of the parallel database system we consider is a shared-nothing parallel database. Fragments of relations on different nodes have separate indexes, also known as partial indexes. The paper is organized as follows: Section 2 details the related work. Section 3 describes the OAT method and shows its correctness. Section 4 introduces new on-line bulk index insert and delete algorithms. These are used by BULK, which is also described in the same section. In section 5, we describe the experimental testbed on which we conducted the experiments. In section 6, we make a performance evaluation of BULK and OAT. In section 7, we make a qualitative comparison of BULK and OAT, and state our conclusions in section 8.

2 Related Work

Though there has been much work in the area of on-line reorganization in recent years [20, 9, 16, 6, 11, 12, 7, 8], there has hardly been any work that considers index modification overhead during on-line reorganization. Perhaps [11] is the only paper that discusses a solution for on-line index modification. However, the techniques in the paper are limited to centralized databases and requires the use of locks. They also do not provide any performance results. Using locks during reorganization can degrade performance significantly. OAT is a technique that enhances the availability of the data by not holding locks during index modification. Since it moves one page at a time, it can be compared to the method in [11]. We found that OAT performs better than the technique in [11]. A comparison of the method in [11] and OAT can be found in [1].

In [14], an on-line algorithm for the reorganization of key sequenced files is presented. The principal technique used to move a chunk of data from one disk to another is to make a copy of the data on the destination disk. Updates are allowed on only one copy, and at the correct time, a switch is made synchronously to transfer the updates to the new copy. In Tandem NonStop SQL [18], a partition of data is moved from one disk to another in four phases:

1. Sequential Read: In this phase, a copy of the existing (original) partition is made using a fast technique such as sequential I/O. Therefore, the data is moved without any user application experiencing additional lock contention. But this phase allows inconsistent data to be copied to the target partition.

2. Audit Fixup 1: In this phase, the dirty copy of the data is brought up to date with the original partition's data. This is accomplished by examining the audit trail (log) for the original partition to find records that have changed since the dirty copy was made. This phase ends when the audit-fixup process reads the last record in the audit trail. At this point, the data in the target partition looks exactly like the data in the original partition except for uncommitted transactions. Also, any changes made to the data after the audit fixup phase ends will not be reflected in the target partition.

3. Audit Fixup 2: In both phases 1 and 2, reads and modifications are allowed on the original partition. In the third phase, locks are requested to make sure all outstanding

user transactions against the table are completed and applied to the new partition before it is made logically part of the table's definition. After the lock is granted, the audit fixup process is called again to finish applying the relevant audit.

4. Cleanup: The original partition is deleted in this phase.

We adopt a similar strategy in BULK. However, the paper [18] does not consider index modification, and also does not provide any performance results. On-line index modification and its performance is the focus of BULK and this paper.

The primary feature of BULK is its ability to perform bulk inserts and deletes into indexes concurrently with transactions. An efficient batch insertion algorithm is presented in [4]. Unfortunately, it is an off-line algorithm. In [1], we compare three on-line batch index modification algorithms. The first is the conventional insert one at a time. The second is an adaptation of the off-line algorithm in [4]. The third technique uses sequential prefetch I/O. We found that the sequential prefetch I/O performed the best. BULK uses sequential prefetch I/O.

In [13], the authors present an on-line method for the dynamic redistribution of data which is based on reallocation of file fragments. Their main focus is on evaluating a heuristic for load balancing in the presence of changing access patterns. A limitation of their study is that they do not consider index modification.

To the best of our knowledge, this is the first paper that deals with the performance evaluation of on-line index modification methods in parallel databases.

3 The OAT method

A source node can be defined as the node from which the data pages have to be moved to other disks. Similarly, a destination node can be defined as the node at which data pages have to be stored. Without loss of generality, we assume that there is only source node and one destination node in the system at any given time. If data is moved from one source node to many destination nodes, each destination node is considered one at a time. Similarly, we can handle the following cases: multiple source nodes to single destination node, and multiple source nodes to multiple destination nodes.

In this section, we describe the OAT method. OAT stands for One-data-page-at-A-Time data movement. In this method, one data page at a time is transferred from the source node to the destination node. Thus, the space overhead is just one page. Each time the data page is moved, all the indexes on that data page are modified. Since each data page contains many records, modifying all the indexes for all the records on the page can take significant time. If locks are held on the page and the records until all the index entries are modified (as suggested in [11]), those transactions that require the page being moved can suffer severe performance degradation. OAT is a technique that recognizes the fact that index modification can take a long time. Therefore, *the principal idea behind OAT is to allow access to the data page during index modification without holding locks.* OAT achieves

Source node

```
1. Initialize RBSN to 1.
2. While there are more data pages to move
3.      X latch RB
4.      Move page into RB
5.      Unlatch RB
6.      For each index on relation
7.           Extract keys from the page and sort them.
8.           Delete keys from the index using conventional delete algorithm.
9.      end for
10.     BEGIN TRANSACTION
11.     Get a X latch on RB.
12.     SEND page to destination node
13.     RECEIVE acknowledgment from
        destination Node
14.     Delete page from RB
15.     Increment RBSN by 1
16.     Unlatch RB
17.     END TRANSACTION
18.     RECEIVE PROCEED message from destination node
19. end while
```

Figure 1: The source node algorithm of the OAT method

Destination node

```
1. Initialize RBSN to 0.
2. While there are more pages to receive
3.     RECEIVE page from Source node
4.     BEGIN TRANSACTION
5.     Get a X latch on RB.
6.     Increment RBSN
7.     Move page into RB
8.     Unlatch RB
9.     SEND acknowledgment to source node
10.    END TRANSACTION
11.    For each index on relation
12.        Extract keys from the page and sort them.
13.        Insert keys from the index using conventional insert algorithm.
14.    end for
15.    X latch RB
16.    store page in RB on disk
17.    Unlatch RB
18.    SEND PROCEED message to source node
19. end while
```

Figure 2: The destination node algorithm for the OAT method

```
1. S latch RB. (Each node has a separate RB. The sub-transaction
   latches the local RB).
2. Get RBSN of the RB.
3. Find all items requested by the transaction. Also search the
   RB to find all satisfying records.
4. Unlatch RB.
5. Commit sub-transaction.
6. Send satisfying keys and the RBSN to the client.
```

Figure 3: Sequence of steps in the sub-transactions

1. `Spawn sub-transactions on all the relevant nodes.`
2. `Await results and the RBSN from the nodes.`
3. `If` $\texttt{RBSN}_{source}$ `-` $\texttt{RBSN}_{destination}$ `= 1 then global transaction executed correctly.`
4. `If` $\texttt{RBSN}_{source}$ `-` $\texttt{RBSN}_{destination} \leq$ `0 then global transaction may have read duplicates. Take appropriate action by initiating a compensation transaction at the destination node.`
5. `If` $\texttt{RBSN}_{source}$ `-` $\texttt{RBSN}_{destination} >$ `1 then global transaction may have missed data in transit. Take appropriate action by initiating a compensation transaction at the destination node.`

Figure 4: The Central Transaction Manager Algorithm (From Theorem 1 later in the paper)

this by storing the page in a separate buffer space called RB (for Reorg Buffer). The source and the destination nodes have seperate RBs. Although, OAT moves one data page at a time, it can be easily modified to move tens of pages at a time. In this paper, we restrict OAT to move exactly one page at a time.

The OAT method has the following features: (a) It uses a buffer called the Reorg Buffer (RB) to temporarily store the data pages being moved. The purpose of RB is to enhance availability of the page being reorganized. There is a different RB at the source node as well as the destination node. (b) latches are held only for the actual duration of the physical transfer of a page from one node to another. Latches are not held on the data page when the page is not in transit, e.g., when the indexes are being modified.

The RB has five principal components:

1. an sequence number RBSN - the RBSN is incremented by one each time the a page is brought into the RB for reorganization.
2. a latch on RB
3. a hash table to quickly search for a key in the pages in the RB.
4. a sorted list of keys extracted from the page in RB. The index is modified using this list.
5. a set of buffer pages. This can be easily implemented by partitioning the database buffer pool. The buffer manager should guarantee that the pages in RB will not be swapped out. This can be easily done by setting hints on the page.

The data pages in the RB are available to the transactions during the time the reorganizer is busy modifying the indexes. That is, at the source node, transactions can access the pages in RB until the reorganizer has completed the deletion of the index entries. After the index entry deletion is complete, the reorganizer gets an exclusive latch on the RB. It then sends the pages to the destination node. It releases the latch on the RB only after it receives an acknowledgment from the destination node. The destination node sends an acknowledgment only after storing the pages in its RB, and incrementing its RBSN. Because of exclusive latches, the page is not available to the transaction only for a short duration when the page is in transit from one node to another. Figures 1 and 2 show the source and destination reorganization algorithms for the OAT method. Although index insertion and deletions can go on in parallel at the destination node and the source node respectively, we deliberately do not allow parallelism. Note that the source node reorganization algorithm awaits a PROCEED message from the destination node before starting the process of moving another page. The reason we do this is because, in our initial experiments, we observed that transactions suffer a greater degradation in performance if the reorganization is allowed to do inserts and deletes in parallel. And the reason for this is simple - if the reorganization consumes more resources (such as CPU and disk), it interferes more with the transactions causing greater performance degradation.

Note that there is exactly one copy of the page accessible to the transactions in the database system. Availability can be further increased if there are two copies of the pages, one at the source node, and the other at the destination node. But the downside is that the consistency of the two copies must then be guaranteed. In this paper, we do not study the multi-copy approach.

Since secondary indexes are being modified, the index entries corresponding to the tuples in the page being moved will be scattered all across the leaf pages of the index. This can result in a lot of random I/Os. One way to minimize the random I/O is to sort the keys in ascending order, and then access the leaf pages of the index in the sorted key order. But the efficacy of this method depends on the length of the sorted list - the longer the list, the more effective sorting could be in minimizing the I/Os. For accruing the potential benefit, OAT sorts the set of keys before beginning the modification of the indexes.

The transaction path has to change a bit because of on-line reorganization. The sequence of steps followed by sub-transactions is shown in Figure 3, and the central transaction manager in Figure 4. It is inevitable for every transaction to search the RB for the keys whenever the reorganization is in progress. This is because of the following:

1. For each key to be retrieved, the transaction looks up the B-tree and retrieves the corresponding RID. If the page corresponding to the RID is being reorganized, the record should be retrieved from the RB. However, if the B-tree search fails to return an RID, the transaction still has to look up RB, because the reorganizer could have deleted the corresponding index entry.

2. If the transaction is performing page scans to satisfy a query, it has to look up the RB to determine if the pages being reorganized belong to the relation being scanned. If so, the pages in the RB have to be processed.

In either case, RB has to be looked up. Because of on-line reorganization, a transaction performing update, insert, and delete operations is affected as follows. Whenever a tuple in RB is modified, the RB should be appropriately modified.

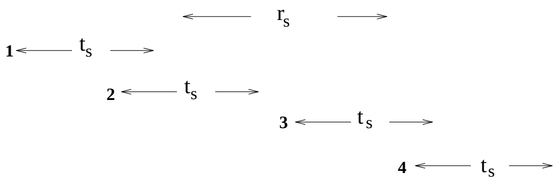
1. **Update**: Suppose the transaction has modified an attribute of a record from a_1 to a_2. In the normal case where the reorganization is not in progress, the transaction would delete the index entry corresponding to a_1 and insert an index entry corresponding to a_2. However, when reorganization is in progress and the record updated is in RB, then the transaction has to perform actions dependent on whether it is at the source node or the destination node: If it is on the source node, then it neither has to delete the index entry or insert a new index entry. The reason it does not have to delete the corresponding index entry is because the reorganizer would do it in any case. And the reason it does not have to insert a corresponding index entry is because the page is anyway being moved to another node. If the transaction is on the destination node, the actions depend on whether the reorganizer has already inserted the corresponding index entry for a_1. This can be easily determined by looking at the position of the reorganizer in the sorted key list. If the reorganizer has already inserted a_1, the transaction should do its normal index modification work, i.e., delete the index entry for a_1, and insert the entry for a_2. If the reorganizer has not already inserted a_1, then the transaction could allow the reorganizer do the index modification. The transaction changes the position of a_1 to a_2 in the sorted list. If a_2 is less than the current key being inserted by the reorganizer, then the transaction itself does the index modification.

2. **Insert**: If the transaction is inserting a new tuple in the page, then the index actions again depend on whether the transaction is executing on the source node or the destination node. If it is at the source node, the transaction does not have to bother modifying the indexes (for reasons similar to the update case). If it is at the destination node, then the best way would be for the transaction itself to do the insert into the indexes.

3. **Delete**: If the transaction is at the source node, then the tuple should be deleted from the page in the RB. At the destination node, the actions are more complicated just as in the update case. The transaction has to determine if the reorganizer has already inserted the key into the index. If so, it should delete the entry from the index. Otherwise, the transaction has to delete the key from the sorted list used by the reorganizer.

We will now show that the OAT reorganization algorithm is correct. There are two aspects to showing the correctness of OAT:

1. Queries will not read the same data item twice, once at the source and the second time at the destination node.



4 possible states of the transaction with respect to the reorganization.

Source Node \ Dest. Node	1	2	3	4
1	OK	OK	DUP	DUP
2	OK	OK	DUP	DUP
3	MISS	MISS	OK	OK
4	MISS	MISS	OK	OK

MISS: Transaction misses data item
DUP: Transaction sees data item twice
OK: Transaction sees data item exactly once

Figure 5: The various states of the transaction with respect to the reorganization.

2. Queries will not miss any data being reorganized because of the data being in transit between the nodes.

Since the global transaction is split into sub-transactions and sent to different nodes, a global transaction can see a data item twice or completely miss it. This could happen when the data is in transit. We will now show that by using the algorithms in Figures 1 through 4, the transactions will not miss or see data twice.

Denote the reorganization process at the source node as r_s and at the destination node as r_d. In the OAT method, the data is in transit between lines 10 and 17 in Figure 1, and lines 4 and 10 in Figure 2. These lines are executed within transaction boundaries, and can be referred to as r_s and r_d respectively. This constitutes a reorganization step. Let the sub-transaction at the source node and the destination node be represented as t_s and t_d respectively. When a transaction starts at the host, the client spawns sub-transactions at the source and the destination nodes, namely t_s and t_d. Figure 5 shows the four possible states of t_s with respect to r_s. The state is indicated by a number on the left side of the arrow. For example, in state 1, the transaction t_s starts and ends before the reorganization r_s begins. Similarly, t_d can be in any one of the four states with respect to r_d. The table on the right hand side of Figure 5 indicates the situations in which a global transaction could see a data item twice, or miss it. For example, if the sub-transaction of the global transaction is in state 1 at the source node, and another sub-transaction of the same global transaction is in state 3 at the destination node, then there is a likelihood that a data item can be seen twice. One way of detecting if a transaction executed incorrectly is to merge the execution schedules of each node and examine the global schedule. If there is a global cycle, then it would be highly likely that a transaction saw a data item twice or missed it. But this approach is messy and expensive. It requires the maintenance of local and global schedules. The following theorem presents an elegant alternative. It is cheap and requires the maintenance of sequence numbers of the RB at the nodes.

Theorem 1 *Let the transaction require the page currently being moved by the reorganization. Let SN_{source} be the sequence number of the RB at the source node ($RBSN_{source}$) returned by the t_s, and SN_{dest} the sequence number of the RB at the destination node ($RBSN_{dest}$) returned by t_d. Then the following conditions can be used to efficiently detect the correctness of the global transaction that spawned t_s and t_d during on-line reorganization in a parallel database:*

1. *the transaction executed correctly if and only if SN_{source} - SN_{dest} = 1.*
2. *the transaction saw data twice (once at the source node and the other at the destination node) if and only if SN_{source} - $SN_{dest} \leq 0$.*
3. *the transaction missed the data in transit if and only if SN_{source} - $SN_{dest} > 1$.*

<u>*Proof:*</u> First, consider the situation when there is no on-line reorganization. Then RBSN_{source} - $\mathrm{RBSN}_{dest} = 1$ because of the initializations of the sequence numbers. Therefore $\mathrm{SN}_{source} = \mathrm{RBSN}_{source}$, and $\mathrm{SN}_{dest} = \mathrm{RBSN}_{dest}$. Hence the global transaction executes correctly. The condition RBSN_{source} - $\mathrm{RBSN}_{dest} = 1$ is an invariant whenever a page is not in transit. When a page is in transit, RB is X latched by the r_s and r_d. During the transit, RBSN_{dest} is equal to RBSN_{source} for a brief time, but by the time r_s completes, RBSN_{source} is once again 1 more than RBSN_{dest}.

1. First we prove the necessary condition. Let the transaction execute correctly. This means that the transaction has seen the data item exactly once. This can happen only if one of t_s and t_d see the data item. Let's suppose that t_s saw the data item. Then t_s must be in either state 1 or state 2 (or state 3 or 4). The only way t_d will not also see the the data item is by being in states 1 or 2 (or state 3 or 4). Then r_s and r_d start after t_s and t_d respectively. This means that t_s holds the S latch on RB, which implies that a page cannot be transferred to the destination node until t_s releases the S latch. Similarly, the destination node cannot store the incoming page in RB until t_d releases the latch on RB. Therefore, the page cannot be in transit, and therefore RBSN_{source} - RBSN_{dest} should be 1. Therefore $\mathrm{SN}_{source} = \mathrm{RBSN}_{source}$, and $\mathrm{SN}_{dest} = \mathrm{RBSN}_{dest}$, and SN_{source} - $\mathrm{SN}_{dest} = 1$.

   Next we prove the sufficient condition. Let SN_{source} - $\mathrm{SN}_{dest} = 1$. This means $\mathrm{RBSN}_{dest} = \mathrm{RBSN}_{source}$ - 1, which implies that the page was not in transit. If the page is not in transit, then exactly one of t_s and t_d can see the data item, and hence executes correctly.

2. Let the global transaction see the same data item twice. That is, both t_s and t_d see the data item. This can happen only if t_s sees the data item before the r_s can ship it to the destination node. Therefore, t_s is in state 1 or 2. Since t_d also sees the data item, it must be in state 3 or 4. Let the value of RBSN_{source} be n. Then, at the time t_s began, RBSN_{dest} is $n-1$. Therefore $\mathrm{SN}_{source} = n$. Since t_d begins after r_d starts

and can continue only after r_d releases the X latch on RB, RBSN$_{dest}$ value increases to n. However, any number of reorganization steps can complete before t_d begins. Therefore SN$_{dest}$ is at least n, and SN$_{source}$ - SN$_{dest}$ ≤ 0.

Next we prove the sufficient condition. Let SN$_{source}$ - SN$_{dest}$ ≤ 0. That is, RBSN$_{dest}$ is at least RBSN$_{source}$. First, consider the case RBSN$_{dest}$ $=$ RBSN$_{source}$. There are two situations in which this could occur. First, it could occur during transit. Second, t_s starts before r_s, and t_d starts after r_d. In the first case, since r_s and r_d hold X latches on the RB, it is impossible for the transactions to have read the sequence number when the page was in transit. Hence the only possibility is the the second case, i.e., t_s is in state 1 or 2, and t_d is in state 3 or 4. This implies that both t_s and t_d see the data item. Subsequently, the global transaction sees the data item twice. A similar argument can be used to handle the case where RBSN$_{dest}$ $>$ RBSN$_{source}$.

3. Let the global transaction not see the data item. This means neither t_s and t_d see the data item. This is possible only if t_s is in state 3 or 4, and t_d is in state 1 or 2. Let RBSN$_{source}$ $= n$. When the page is not in transit, RBSN$_{dest}$ $= n-1$. Since t_d starts before r_d, SN$_{dest}$ is $n-1$. Since t_s starts after r_s, SN$_{source}$ is $n+1$, or more precisely, at least $n+1$. This because any number of reorganizations steps can be completed before t_s begins. Therefore SN$_{source}$ - SN$_{dest}$ > 1.

   Next we prove the sufficient condition. Let SN$_{source}$ - SN$_{dest}$ > 1. Without loss of generality, let SN$_{source}$ - SN$_{dest}$ $= 2$. This can happen only if t_d starts before r_d, and t_s starts after r_s. If t_s starts after r_s, t_s will not see the data item because the data item was shipped to the destination node. Since t_d starts before r_d, it will also not see the data item. Since neither sub-transaction sees the data item, the global transaction will not see the data item in transit.

□

The above theorem provides a very simple and powerful means of detecting whether transactions are executing correctly in the presence of on-line reorganization. The theorem is general in that it applies to any kind of data movement across nodes in a parallel/distributed database system, provided sequence numbers are maintained as in the OAT algorithm. The central transaction manager, shown in Figure 4, makes a decision about the correctness of the transaction execution based on the RB sequence numbers. However, making a decision purely on the RB sequence numbers can sometimes be incorrect because the transaction may not have required the page being moved. For example, a transaction may require page 10 where as the reorganizer is moving page 1. Thus there is a possibility of signaling false misses or duplicates. This is the reason why the theorem states that the transaction must require the same page currently being moved by the reorganizer. If a transaction missed seeing the data item, then the central transaction manager can re-issue the sub-transaction at the destination node. On the other hand, if the transaction saw the data item twice, the central transaction manager can abort the transaction and restart the transaction. More research is required to determine the most efficient way to perform these actions.

4 The BULK method

The BULK method is shown in Figure 7. There are five main steps in the BULK method. The first step makes a copy of the data at the destination node. Using this copy, the indexes are modified at the destination node in step 2. However, by the time BULK completes index modification at the destination node, the copy at the source node could be different from the copy at the destination node because updates are allowed at the source node. In order to reflect the updates, BULK has to collect the updates and perform them again at the destination node. This is performed in step 3 and is similar to the approach taken in Tandem NonStop SQL [18] (see Related Work Section). In step 4, higher level meta data is modified to divert the transactions to the destination node. For example, suppose the relation is range partitioned on two nodes into two ranges: A-M, and N-Z. Using BULK, the range H-M was moved to the destination node. Then the higher level meta data should be modified to indicate that the ranges are now A-G and H-Z. In step 5, the original copy of the data on the source node is deleted.

To modify the indexes, BULK uses an on-line bulk insert and delete method. The on-line bulk insert method is shown in Figure 6. The bulk index modification techniques use sequential prefetch I/O to fetch index leaf pages. In our experiments, each I/O operation fetches 16 pages at a time. In some commercial implementations such as DB2, 32 pages can be fetched in one I/O call. When the index is first created, all the leaf pages are logically and physically contiguous. Also, all the leaf pages are stored before the higher level index internal pages. Many commercial databases, e.g., DB2, give a certain degree of control on the layout of the indexes. However, with splits and merges, the leaf pages may no longer be physically contiguous. When leaf pages are fetched using sequential I/O, a set of pages that are physically contiguous but not necessarily logically contiguous are stored in the buffer. The main idea behind bulk insert is to maximize the work on each leaf page every time it is fetched into the buffer, and every effort is made to fetch each leaf page exactly once. Therefore, to maximize the work on each leaf page, the very first step of the bulk insert algorithm is to extract the the set of keys to be inserted and deleted from a index from the data, sort and store in a file referred to as the bulk file. When a leaf page is fetched from the disk, the bulk file is searched to find all the index entries that can be inserted into the leaf pages in the index before the leaf page is released. In order to make the search of the bulk file efficient, we keep the bulk file in main memory. If the bulk file does not fit into main memory, then it becomes an interesting optimization problem. We do not examine this case in this paper. For the purpose of this paper, we assume that the bulk file fits in main memory.

Note that the copy of the data at the destination node is not available for transactions until after the completion of step 4. It is possible that transactions at the destination node

```
1. Sort the bulk file according to its key value and store it in
   main memory.
2. While there are more leaf pages to fetch
3.   begin
4.   Fetch a set of consecutive 16 pages into the buffer using sequential I/O.
5.   Set a hint on the pages so that the buffer manager does not swap them out.
6.    For each leaf page in the buffer do
7.    begin
8.       fix the leaf page in the buffer.
9.       Determine the range of keys that fit on the current leaf page. If
         necessary, do a random I/O to fetch the higher level index page.
10.      While there are more keys in the bulk file that fit on this leaf page
11.        begin
12.          if there is space in the leaf page,
13.              squeeze the key into the leaf page.
14.          Else
15.            unfix the leaf page, but do not reset the hint on the page.
16.            Perform a conventional insert into the page. This will
               cause a page split, and one-half of the keys on the current
               leaf page will be moved to a new page.
17.            Set a hint on the new page so that it would not be swapped
               out of the buffer.
18.            Determine the new range of keys that fit on the current leaf page.
19.            Fix the leaf page in the buffer.
20.      end
21.      reset the hint on the leaf page so that the buffer manager can swap
         it out.
22.      unfix the leaf page in the buffer.
23.  end.
24.end.
```

Figure 6: Bulk insert using Sequential I/O to fetch 16 pages in one I/O call.

```
1. Make a copy of the pages to be moved from the source node, and
   send it to the destination node.
2. For each attribute on which there is an index
      extract the keys and store them in a file called the bulk file.
      Use the Bulk Insert algorithm shown in figure 6.
3. If necessary, apply the Phases 2 and 3 of the Tandem NonStop SQL method
   to reflect any updates to the pages at the source node that were
   missed during steps 1 and 2.
4. By obtaining locks, modify higher level meta data to divert the
   the transactions to the destination node.
5.  For each attribute on which there is an index
      extract the keys and store them in a file called the bulk file.
      Use the Bulk Delete algorithm (Not shown in the paper - it is
      similar to Bulk Insert algorithm).
```

Figure 7: The sequence of steps in BULK.

will access the index entries corresponding to the moved data even before step 4 completes. In order to prevent the transactions from accessing the data, some information must be kept regarding the validity and availability of the pages, i.e., if transactions try to read/modify data items at the destination node before completion of step 4, the read/modify should be forced to fail. One solution is to keep a table in main memory of the page ids of the pages being reorganized. A transaction that needs a page from this set can abandon its attempts to read/modify the page. More research is needed to examine the various issues involved.

We can show the correctness of BULK in a similar fashion as OAT. In this case, step 4 will be performed as a transaction.

5 Experimental Testbed

To conduct the experiments, we built a functional shared-nothing parallel database system. Each node has its own local lock manager, buffer manager, and a catalog. The lock manager implements a variety of lock modes such as Shared, Exclusive, SIX, and Intension locks. It also supports lock conversion, deadlock detection, instant locks, and conditional locks. The lock manager uses a hash table to quickly locate the locks. The buffer manager use the LRU scheme for replacing pages in th buffer. It also allows the transactions to set hints on the pages so that the transactions can control the pages that are held in the buffer. The transactions are required to call the `fix()` routine before they can access the page. The `unfix()` is called as soon as the page is no longer needed by the transaction. The buffer manager has a hash table to support quick access to the pages. There are B-tree partial indexes on the fragments of the relations. The B-tree supports high concurrency, and is based on the ARIES/KVL described in [5]. The structure of the database system at a node

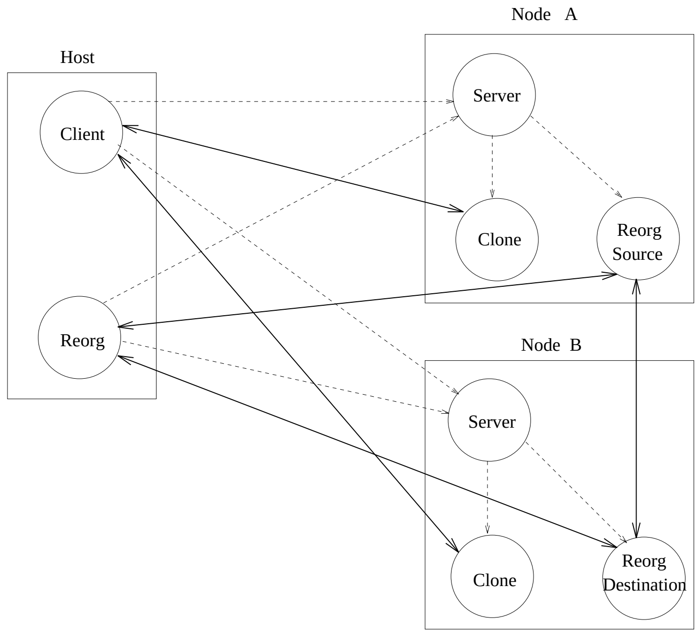


Figure 8: The process architecture of the shared nothing system

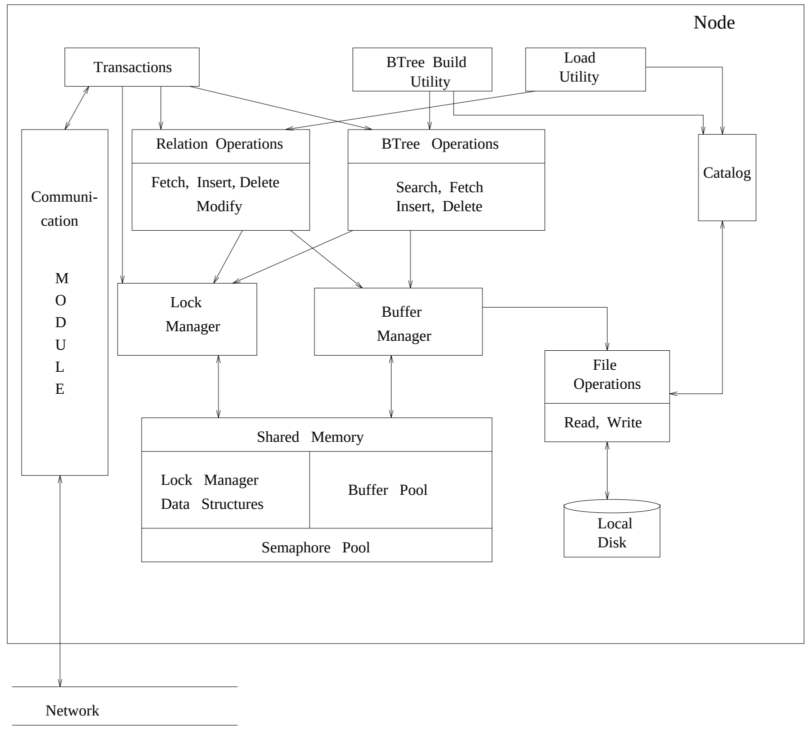


Figure 9: The database system internals at a node

is shown in Figure 9. The process architecture of the parallel database system is shown in Figure 8. The process architecture is based on the process-per-client paradigm, i.e., for each client, there is a dedicated thread at the server called the `clone`. The parallel database system can also operate in a simulation mode based on a switch given during compile time. The simulation mode is achieved by using a few CSIM statements. To make performance measurements, it is very useful to use the system in the simulation mode, because it easily allows the measurement of quantities such as CPU utilization, and disk utilization. In the simulation mode, the paradigm is a thread-per-client paradigm. The reorganization processes are denoted by `reorg source` and `reorg destination` respectively.

The communication layer was built using the Unix Transport Layer Protocol. In the simulation mode, the network delay was a constant 2msec. Since the network is not a bottleneck and not even a highly utilized resource during the reorganization, a crude approximation to the network delay was sufficient for our experiments.

6 Performance

In this section, we compare the performance of OAT and BULK. The metrics for comparison are the time to complete reorganization, and the impact on the response time and throughput of transactions. For the experiments, we restrict the number of nodes in the parallel database system to be two. This is necessary, because as we mentioned before, there is only one source node and one destination node in the system at any given time. By considering a two node system, we will be able to clearly distinguish the impact of OAT and BULK on the transactions. If we considered a system with more than two nodes, all transactions not executing on the source and the destination node will not be affected by on-line reorganization, because they will not see the extra CPU, disk, buffer, lock and latch contention due to on-line reorganization. Therefore, the true performance impact of on-line reorganization on transactions that suffer degradation will be masked by the performance of transactions that are not affected by the reorganization. The masking effect would especially occur when quantities such as average transaction throughput and mean response times are measured.

We considered one relation with 5 attributes. Again, there was no point in having more than one relation in the system because the reorganizer moves records from only one relation at a time. Each tuple was 40 bytes in length. The relation has two fragments. Fragment 1 was placed on node 1, and contained 80000 tuples. The second fragment on node 2 contained 40000 tuples. There were 5 secondary indexes on the relation. The page size was 2K bytes. There were 128 buffer pages on each node of the system. Each node has 1 disk.

The transactions were a simple debit-credit type of transactions. Each transaction accesses one index to locate the page of a data item, and then retrieves the page from disk and accesses the data item. Each transaction retrieves exactly one data item. After retrieving the data item, the transaction holds the CPU for an exponential time with a mean of 25 msecs. We restrict the transactions to be read-only, i.e., there are no updates. This restriction does not change our results in any way, except that the reorganization will take a slightly longer time to complete, and the throughput would be lower and response times would be higher if the updates were included. But the relative performance of BULK and OAT would remain the same as in a read-only case.

We perform two kinds of experiments:

1. The reorganization of disk hot spots: That is, the only set of data items requested by the transactions are in a small set of pages. In our experiments, we set the number of hot pages to 200. Furthermore, all the hot pages were located on the source node. Therefore, the objective of reorganization is to move one half of the hot spot pages to the destination node. After the reorganization is complete, the hot set pages are equally divided among the two nodes. Since the transaction retrieves exactly one key, the central transaction manager spawns a sub-transaction on exactly one node. This is because central transaction manager knows the location of the hot set pages. Thus, in the experiments reported in this paper, there are no duplicate and miss problems. We would expect the throughput to double after the reorganization is complete, and the response time to be cut in half.

2. The reorganization of data that is not a hot spot: In this case, the number of pages to be moved from one node to another is again 100. The transactions retrieve a random record, and are sent to a random node for processing. This is an important case because a large part of data reorganized in a data placement reorganization is not necessarily a hot spot.

First, we consider the disk hot spot case. From Figure 10, we can observe that BULK is an order of magnitude faster than OAT, and index modification overhead is significant. With only one index to modify, the time to reorganize jumps from 39 secs to 1654 seconds for the OAT method, and from 41 seconds to 87 seconds for the BULK method. When there are 5 indexes to be modified, BULK takes 327 seconds, whereas OAT takes about 10592 seconds. The reason OAT takes a very long time can be attributed to the following factors: since the pages are hot spots and the transaction throughput is high (40 transactions/second), and the number of buffer pages is 128, the pages required by OAT are quickly swapped out of the buffer. Therefore, every time OAT accesses a page, it fetches it from disk using random I/O, and this causes its performance to degrade tremendously. The reason BULK is significantly faster than OAT is because of two reasons: first, it is the use of sequential I/O. Second, it makes fewer accesses to the disk to fetch data. Each time a disk request is issued to a disk that is already bottlenecked, a lot of time is wasted in waiting. OAT fetches one page at a time, and hence encounters a large waiting time.

These numbers show that it is important to consider the overhead of index modification in any on-line reorganization work. Until now, this has not been the focus in the literature.

For up to 2 indexes, OAT allows a higher transaction throughput than BULK, and also allows a better transaction response time than BULK. For more than 2 indexes, the

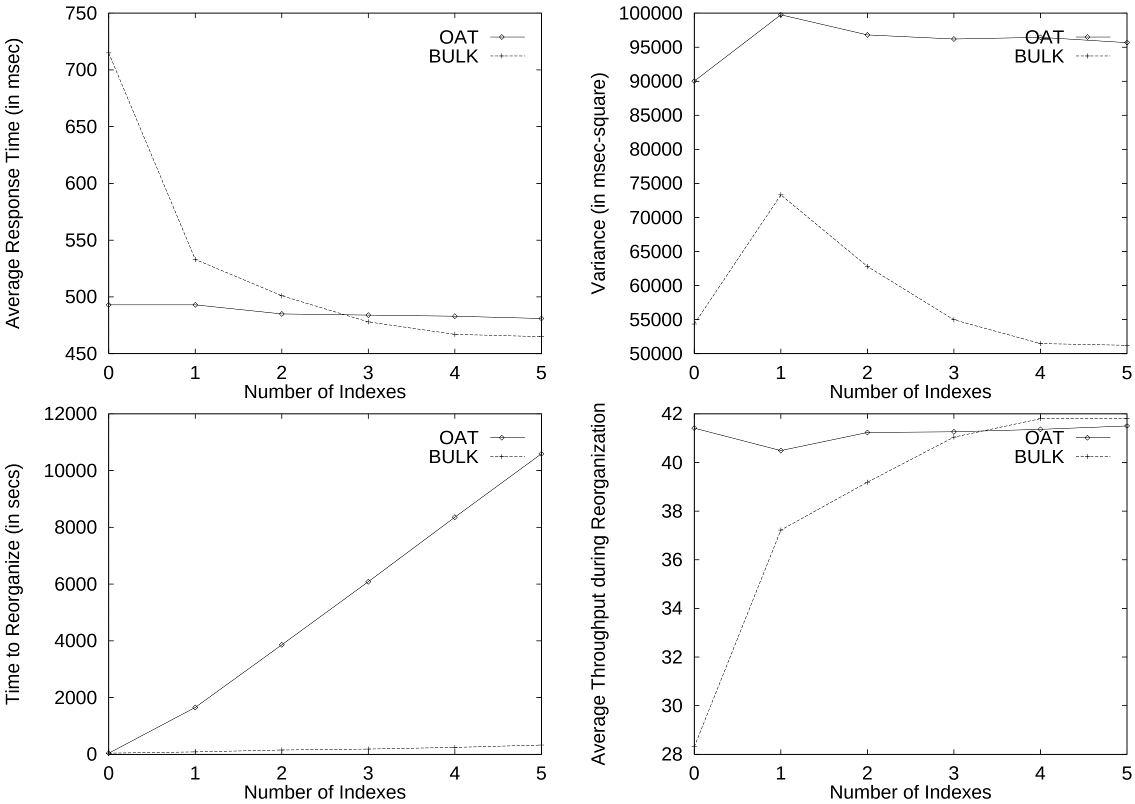


Figure 10: Comparison of OAT and BULK. Hot Set Size = 200 pages

performance is almost the same. The reason for this can be understood by examining the Figures 11 and 12. The are four graphs in each of these figures. The upper left quadrant contains the response time distribution of transactions during reorganization. The upper right quadrant contains the normalized transaction throughput during reorganization. By normalized, we mean that the reorganization duration is scaled down to the interval 0.0 through 1.0 for both BULK and OAT. This is necessary because OAT and BULK take different times to complete the reorganization. The normalized transaction throughput provides some insight into the reason why the average transaction throughput during reorganization differs in OAT and BULK. The lower left quadrant contains the transaction throughput during reorganization. The lower right quadrant contains an enlarged view of the transaction throughput for the time interval 0 through 500 seconds.

The transaction throughput with OAT exhibits a gradual monotonic increase of throughput during reorganization. This is because OAT transfers one page at a time to the destination node. Also, after the page is transferred, the transactions can access the page at the destination node. Therefore, as more and more hot pages are transferred to the destination node, the disk bottleneck at the source node is relieved, and the transaction throughput increases gradually. The BULK method has a different characteristic. BULK makes a copy of the data at the destination node. The transactions cannot access the copy until after the

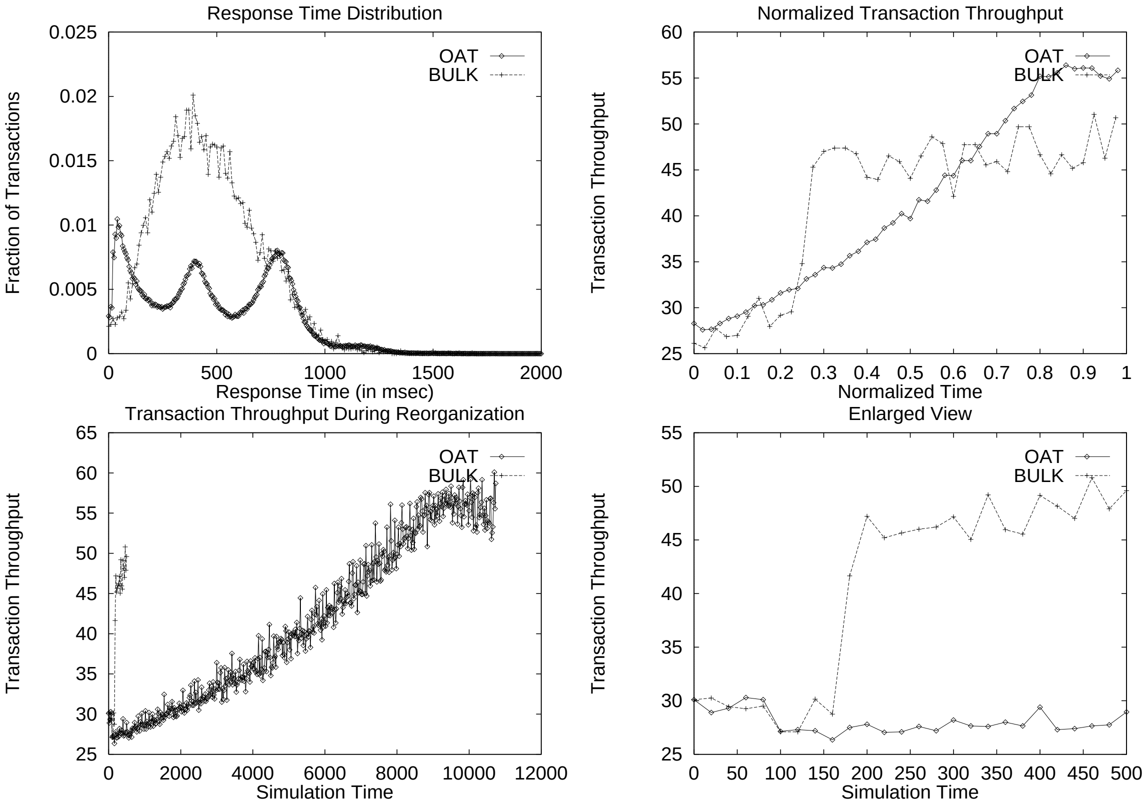


Figure 11: Response Time Distribution and Throughput profile. Hot Set Size = 200 pages. Number of Indexes = 5.

indexes at the destination node are modified to incorporate the new data. But after the inserts are complete, the transactions can access the data at the destination node. This is the reason why the transaction throughput exhibits a sudden jump as soon as the index inserts are complete at the destination node. The transaction throughput is slightly less than what it would be after reorganization, because of the resource contention.

From the normalized throughput graphs, it is easy to see why the transaction throughput is less with BULK for 1-index modification than for 5-index modification. For the 1-index modification case (Figure 12), the jump in throughput occurs after more than 55% of the reorganization duration. Further, the throughput is more than OAT for only 15% of the normalized time. Note that the throughput curve for BULK is mostly below that of OAT in the normalized graph. In the 5-index case (Figure 11), the throughput jump occurs after about 25% of the reorganization duration. Further, the BULK throughput curve is evenly distributed across both sides of the OAT curve. Hence the transaction throughput for BULK increases with the number of indexes to be modified.

There are three peaks in the response time distribution for OAT and two peaks in BULK. Actually, only one peak is noticeable for BULK in Figure 11, but there are two in Figure 12. First let us consider the OAT case in Figure 11. The leftmost peak is the

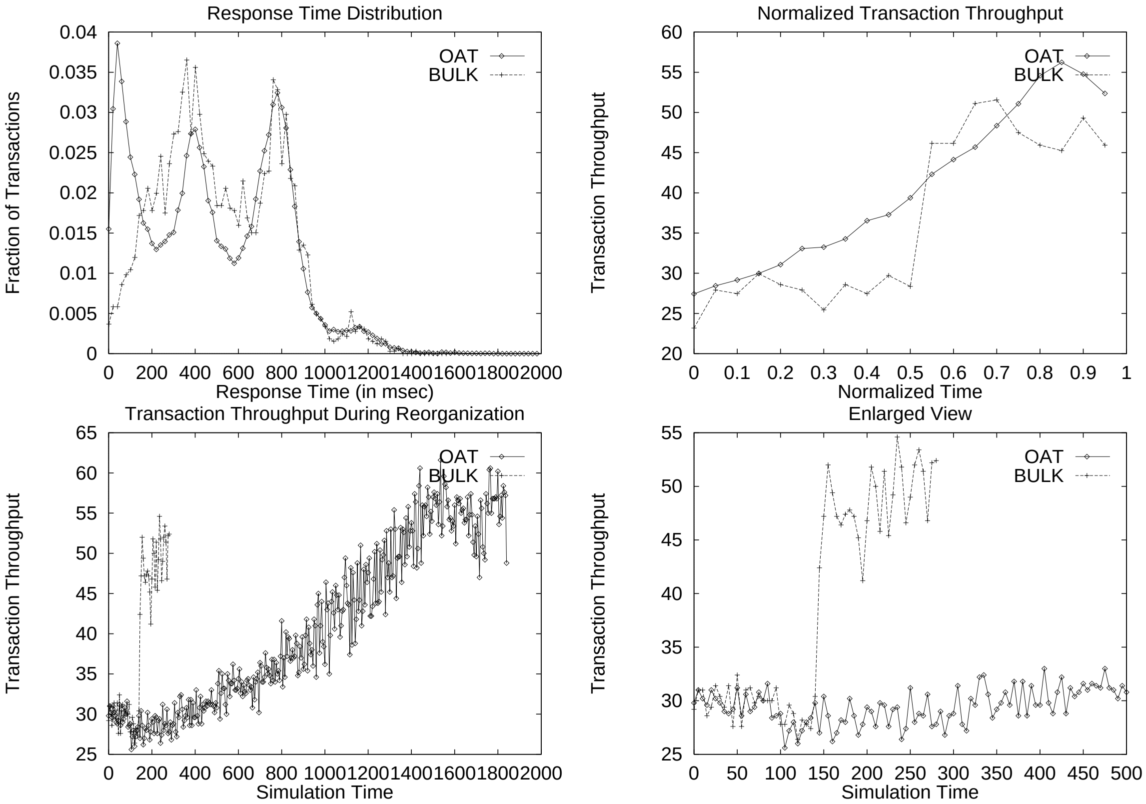


Figure 12: Response Time Distribution and Throughput profile. Hot Set Size = 200 pages. Number of Indexes = 1

first peak, the middle peak is the second peak, and the rightmost peak is the third peak. Notice that the response time at the second peak is approximately half the response time at the third peak. The response time of the third peak is the average response time of the transactions before the reorganization began. The response time at the second peak corresponds to the response time after reorganization. Since OAT moves one page at a time, the response time of transactions gradually improves and moves from the third peak to the second peak. This is the reason why the heights of the two peaks are about the same. Before the reorganization began, there was no traffic at the destination node. With every hot page moved to the destination node, the traffic slowly increases. The first peak corresponds to the response time of transactions which experience no waiting delay of any sort at the destination node. This occurs in the early stages of the reorganization.

BULK has only two peaks and not three because of the same phenomenon observed in the transaction throughput graphs. After the inserts are complete at the destination node, step 4 of the BULK algorithm would instantaneously divide the transaction traffic equally among the two nodes. Hence there is no first peak in the BULK technique. Thus, the right peak corresponds to the average transaction response time before the reorganization began, and the left peak is the transaction response time after the reorganization. Notice from

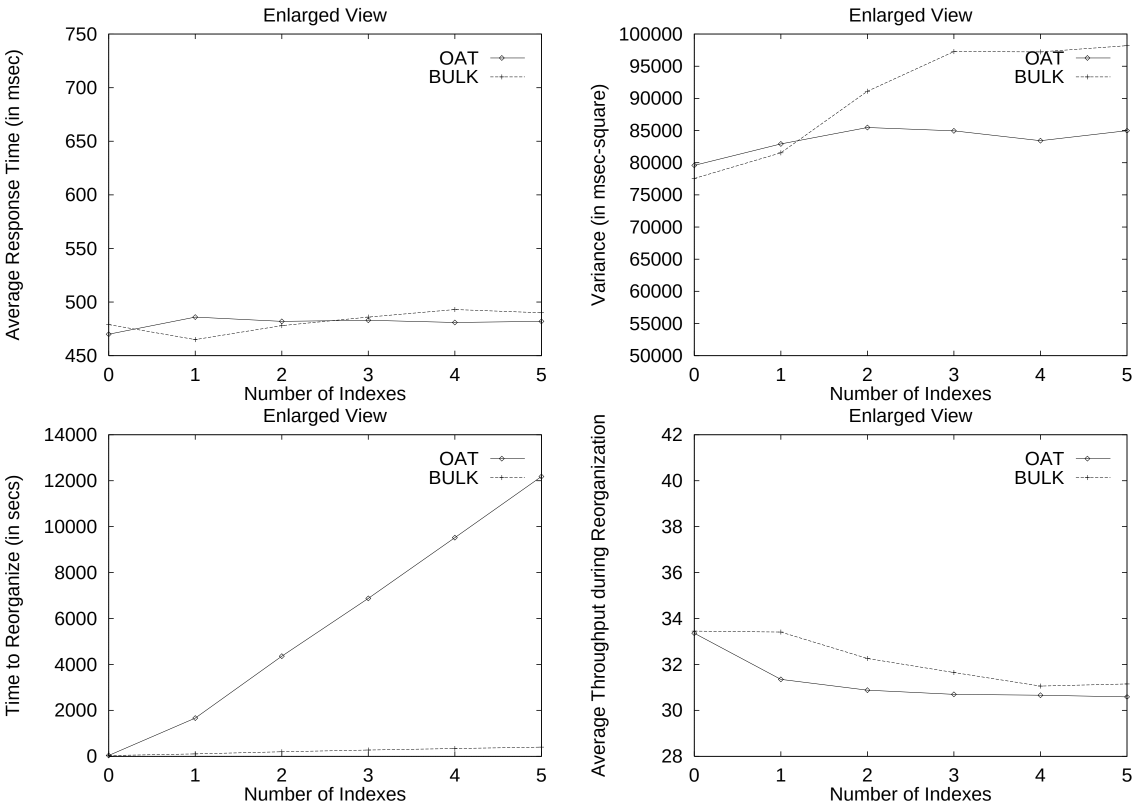


Figure 13: Comparison of OAT and BULK. Number of (non hot set) Pages to reorganize = 200 pages.

the normalized transaction throughput graph in Figure 12 that the insert completes in the middle of the reorganization duration. This is the reason why the heights of the two peaks are the same in Figure 12. On the other hand, the insert completes at about the 25% mark in Figure 11. Hence the right peak is dominated by the left peak in Figure 11.

Until now, we considered the reorganization of disk hot spots. But not all the data to be reorganized belongs to the hot spot. In fact, a large amount of data would not be a hot spot. When non-hot spots are reorganized and moved across nodes, any degradation experienced by transactions is mainly due to resource contention such as CPU, disk and communication lines. Figures 13 and 14 show the performance of transactions during reorganization. As we can see, there is not a significant difference between OAT and BULK except in the time to complete reorganization.

Therefore, the choice of an index-modification technique is mostly driven by hot spot considerations.

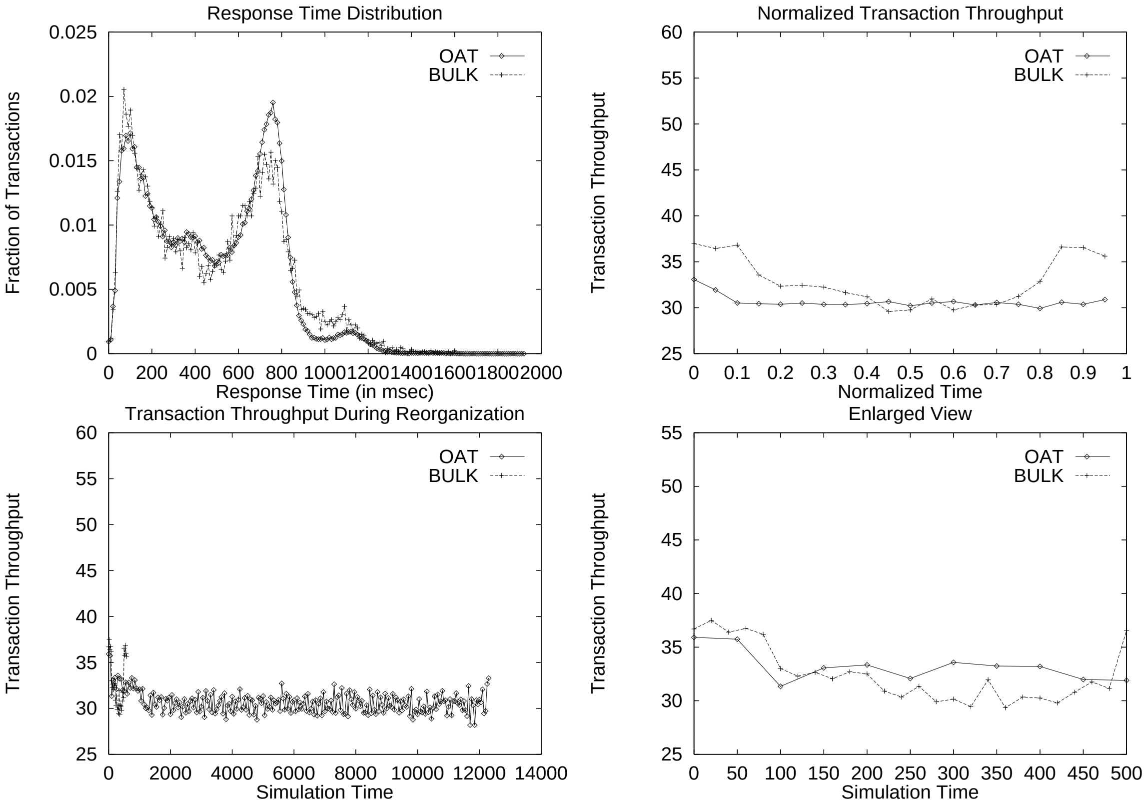


Figure 14: Response Time Distribution and Throughput profile. Number of (non Hot Set) pages to reorganize = 200 pages. Number of Indexes = 5.

7 Conclusions and Future Work

In this paper, we address an important problem in parallel databases - namely what are the techniques and costs involved in moving data across nodes in a shared nothing parallel database system. This is an important issue because any industrial strength parallel database system must have the capability of moving data across nodes in an on-line fashion. Data movement is necessary because the optimal data placement changes with time and usage of the parallel database system, and significant performance improvements can be obtained by reorganizing. Data placement reorganization has been studied before, but none address the critical issue of index modification during reorganization. This paper fills an important void.

We compared two techniques for index modification during reorganization in parallel databases. OAT and BULK are two extremes on the spectrum of the granularity of data movements. BULK and OAT have their advantages and disadvantages. BULK is fast, but uses twice the amount of disk space. OAT is agonizingly slow, but consumes very little extra disk space. BULK requires no modification to the transaction and query path. OAT requires every transaction to visit the Reorg Buffer (RB). This will require some modification to the execution engine of the database system.

In terms of performance degradation of transactions, the choice of the technique will depend on the number of indexes to be modified. If there are only one or two indexes, OAT is clearly preferable because it allows a higher transaction throughput. For more than two indexes, both the techniques perform similarly in terms of average response time and average transaction throughput. But BULK is better in terms of response time for individual transactions, because a larger fraction of the transactions have a lower response time (see the response time distribution graphs and the variance graphs).

If there is a time constraint in completing the reorganization, then BULK is the obvious choice. But the penalty is the space overhead, and possibly some performance degradation for transactions. On the other hand, if there is no time constraint, but a space constraint, OAT is clearly the choice.

The reorganization time with OAT can be decreased by moving more than one page in each step. As the number of pages moved in each step increases, OAT tends towards BULK. The space overhead of BULK can be decreased by moving smaller amounts of data in each step. As the size of the data moved in each step decreases, BULK tends towards the OAT method.

There are a number of avenues for future research. In this paper, we did not consider recovery issues. Recovery is especially important for long duration activities such as on-line reorganization. Another area for future research are the techniques that the central transaction manager must employ when it detects that transactions have either missed or seen data twice.

The techniques presented in this paper have important applications beyond on-line reorganization. For example, in shared-nothing parallel databases, tuples might be forced to migrate due to updates performed on their partitioning attribute. For example, relation R might be hash partitioned on attribute A. If R.A is updated to a new value then the tuple might be forced to migrate to a new node. This migration would involve index modification. When a large number of tuples are modified by using an UPDATE SQL statement, efficient index modification techniques are needed.

References

[1] Kiran J. Achyutuni. *On-Line Tuning of Data Placement in Parallel Databases*. PhD thesis, Georgia Institute of Technology, College of Computing, In Preparation 1995.

[2] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Comm. ACM*, 35(6):85–98, June 1992.

[3] Sybase Inc. *Sybase SQL Server*, release 10.0 edition, Feb 1994.

[4] S. D. Lang, J. R. Driscoll, and J. H. Jou. Batch insertion for tree structured file organizations – improving differential database representation. *Information Systems*, 11(2):167–175, 1986.

[5] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceeding of the 16th VLDB Conference*, Brisbane, Australia, August 1990.

[6] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings ACM SIGMOD Intl. Conf. Management of Data*, pages pp. 361–370, June 1992.

[7] E. Omiecinski. Incremental file reorganization schemes. In *Proc. 11th Intl. Conf. Very Large Data Bases*, pages pp. 346–357, San Mateo, CA, Aug. 1985. Morgan Kaufmann Publishers.

[8] E. Omiecinski. Concurrent file conversion between b+ tree and linear hash files. *Information Systems*, 14(5):pp. 371–383, 1989.

[9] E. Omiecinski, L. Lee, and P. Scheuermann. Performance analysis of a concurrent file reorganization algorithm for record clustering. *IEEE Transactions of Knowledge and Data Engineering*, To appear.

[10] E. Omiecinski and P. Scheuermann. A parallel algorithm for record clustering. *ACM Transactions on Database Systems*, 15(4):pp. 599–624, Dec. 1990.

[11] B. Salzberg and A. Dimock. Principles of transaction-based on-line reorganization. In *Proceedings 18th Intl. Conf. Very Large Databases*, pages pp. 511–520, San Mateo, CA, Aug 1992. Morgan Kaufmann Publishers.

[12] P. Scheuermann, G. Weikum, and P. Zabback. Automatic tuning of data placement and load balancing in disk arrays. Technical Report DBS-92-91, ETH-Zurich, Department of Computer Science, ETH-Zentrum, CH-8092 Zurich, Switzerland, 1992.

[13] P. Scheuermann, G. Weikum, and P. Zabback. Disk cooling in parallel disk systems. *Bulletin of the Technical Committee on Data Engineering*, 17(3):29–40, September 1994.

[14] G. S. Smith. Online reorganization of key-sequenced tables and files. *Tandem System Review*, 6(2):pp. 52–59, Oct 1990.

[15] G. Sockut and R. Goldberg. Database reorganization - principles and practice. *ACM Computing Surveys*, 11(4):371–395, Dec 1979.

[16] G. H. Sockut and B. R. Iyer. Reorganizing databases concurrently with usage: A survey. Technical Report TR 03.488, IBM, Santa Teresa Laboratory, San Jose, CA, June 1993.

[17] V. Srinivasan and M. Carey. On-line index construction algorithms. In *Proc. 4th Internation Workshop on High Performance Transaction Systems*, Asilomar, September 1991.

[18] J. Troisi. NonStop availability and database configuration operations. *Tandem Systems Review*, 10(3):18–23, July 1994.

[19] P. Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases: An International Journal*, 1(2):137–166, April 1993.

[20] G. Weikum, P. Zabback, and P. Scheuermann. Dynamic file allocation in disk arrays. In *ACM SIGMOD Proceedings*, pages 406–415, 1991.