Incremental Organization for Data Recording and Warehousing

H. V. Jagadish[1] **P. P. S. Narayan**[2,4] **S. Seshadri**[3] **S. Sudarshan**[3,5]
Rama Kanneganti[2]

[1]AT&T Labs
180 Park Avenue,
Florham Park, NJ 07932-0636
jag@research.att.com

[2] Bell Laboratories
Murray Hill, NJ 07974
ppsn@research.bell-labs.com
rama@emailbox.lucent.com

[3]Indian Institute of Technology,
Mumbai 400 076, India
{seshadri,sudarsha}@cse.iitb.ernet.in

Abstract

Data warehouses and recording systems typically have a large continuous stream of incoming data, that must be stored in a manner suitable for future access. Access to stored records is usually based on a key. Organizing the data on disk as the data arrives using standard techniques would result in either (a) one or more I/Os to store each incoming record (to keep the data clustered by the key), which is too expensive when data arrival rates are very high, or (b) many I/Os to locate records for a particular customer (if data is stored clustered by arrival order).

We study two techniques, inspired by external sorting algorithms, to store data incrementally as it arrives, simultaneously providing good performance for recording and querying. We present concurrency control and recovery schemes for both techniques. We show the benefits of our techniques both analytically and experimentally.

1 Introduction

A fundamental characteristic of many data warehouses and data recording systems ([JMS95]) is that they record data by appending new data observations to a database. Examples of such systems include point-of-sale data collection systems used in large retail businesses, tracking and billing of telephone calls, collection of stock trading data, and operational-data collection systems in factories and computer networks, which record data from a large number of sensors.

A challenge for these systems is to support very high recording rates (of the order of millions of recordings an hour) while *simultaneously* providing efficient access, based on a pre-specified search key, to the recorded data. For instance, a system recording telephone calls must not only be able to record information fast, but must also be able to efficiently retrieve all call information for a specified customer.

There are two ways of organizing the records: *clustered by search key*, or *clustered by arrival order.* Clustering by arrival order results in records for a particular search key being scattered at random locations on disk, and therefore does not meet our requirements.

The standard way to implement clustering by a search key is to organize the records into a B$^+$-tree file (or hash file) organization, with the search key attributes as the clustering/indexing attributes, and to insert records into the tree (respectively, hash index) as they arrive. In the application domains mentioned above, the values of the indexed attribute of the incoming records are typically randomly distributed over the population. As a result, each successive record in the input stream is likely to end up in a different leaf of the B$^+$-tree (different hash bucket). Since buffer space is likely to be much smaller than the size of the index, at least one I/O is needed for fetching the appropriate leaf node (hash bucket) for each incoming record, one

**Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997**

[4]The work of this author was done while he was at the Indian Institute of Technology, Mumbai

[5]The work of this author was done partly while he was at what was then AT&T Bell Labs.

I/O for writing it back, and possibly more I/Os for internal nodes. Performing disk I/O for each record in the input stream is very costly, greatly reducing the rate at which data can be recorded.

A commonly used work-around in data warehouses is to collect the records and update the database only periodically (such as each night) using bulk-load techniques. The obvious drawback is that the database is significantly out-of-date. Further, bulk loading is done off-line, during which time the database is typically unavailable.

Our goal is to design a technique that supports both insertion and queries with reasonable efficiency, and without the delays of periodic batch processing.

In this paper we study two techniques, based on external sorting algorithms, to achieve these objectives:

1. The first technique stores the records *lazily* in a B$^+$-tree file organization (clustered by the specified key), and is based on external merge-sort. Instead of inserting records into a B$^+$-tree as they arrive, they are organized in-memory into sorted runs. Runs are written to disk when memory is full, and runs on disk are merged to get larger runs. After several levels of merging, when the merged runs have grown relatively large, they are merged into the final B$^+$-tree file organization.

   This technique is based on the same idea as the Log-Structured Merge tree (LSM tree) proposed by O'Neil *et al* [OCGO96]. However there are significant differences, which we discuss in Section 8.

2. The second technique stores records lazily in a hash file organization, and is based on external distribution sort with several levels of partitioning. The hashing based technique is conceptually a dual of the merging based technique, but the implementation details are very different.

As compared to direct insertion, both techniques reduce the number of blocks of data that must be read from and written to disk for insertion and further, perform mainly sequential I/O, rather than random I/O, thereby reducing seek costs as well.

Although the techniques are based on well-known external sorting algorithms, there are important differences from sorting:

1. Unlike external sorting, where an entire file is sorted, records must be organized incrementally, as they arrive.

2. Queries must be allowed on the records, and must be able to retrieve all relevant records that have already been inserted.

3. Concurrency control and recovery must be handled efficiently; neither of these is an issue for external sorting algorithms. We present efficient concurrency control and recovery schemes for both our techniques.

We compare our schemes with the standard scheme of directly inserting records into a B$^+$-tree (respectively, hash file) analytically, as well as empirically by implementing our techniques in a relational storage manager called *Brahma* developed at IIT Bombay.

Our performance results show that, over a wide range of parameters, our techniques can significantly reduce the cost of insertion as compared to direct insertion, while not impacting queries unduly. The results also show that the sorting-based technique outperforms the hashing-based technique—a somewhat unexpected result. Both techniques are of greatest value when the records are small compared to the page size; record sizes of tens to a few hundred bytes, with a page size of 4KB to 8KB, are typical examples.

2 Stepped-Merge Algorithm

Incoming records are stored lazily in a relation whose records are organized in a B$^+$-tree file organization. We call the B$^+$-tree in which the records must finally reside as the *root B$^+$-tree* or more simply, the *root relation*. There are also several intermediate B$^+$-trees, organized into multiple levels, as we will see below.

2.1 Insertion

The insertion algorithm is shown below. The values K and N are parameters to the algorithm.

Algorithm Stepped-Merge-Insertion

1. Collect incoming data in memory in a current run, organized as an in-memory tree. When memory is full, call it the previous run. Start a new run (initially empty) and make it the current run.

   Write out the previous run to disk, constructing a B$^+$-tree on the run as it is written out. The B$^+$-tree is constructed bottom up since the data is sorted. Both the in-memory run and the one just constructed are called Level 0 runs.

2. When K Level i runs, for $0 \leq i < N - 1$, accumulate on disk read back the sorted runs from disk, perform a (K-way) merge and write back a single larger sorted run to disk, calling it a Level $i + 1$ run. Delete the old Level i runs. As before, the run is stored in a B$^+$-tree file organization.

3. When K Level $N - 1$ runs accumulate, merge them, but instead of writing them to a new run, insert the

entries into the root relation. The root relation is also organized using a B$^+$-tree file organization.

The rationale behind the above algorithm is that a large number of records are inserted at a time, in sorted order, into the root B$^+$-tree. As a result multiple records would end up in each leaf, and the number of I/O operations per record is reduced, at the (smaller) cost of increased I/O to create the intermediate runs. A more detailed analysis is presented later.

A *run-index* stores pointers to all the runs currently in existence, including the run currently being constructed in memory. When K runs are merged to get a single run at a higher level, pointers to the K runs are deleted from the run-index, and replaced by a pointer to the single higher-level run. And when a new run is created in memory, a pointer to it is added to the run-index. All the trees together with the run index constitute a *multi-tree index.*

We now consider some simple optimizations. While creating the Kth run of a Level i, instead of writing it out to disk and reading it back again for merging, it can be directly merged with the other Level i runs. As a result of recursively applying this optimization, runs of several different levels may get merged simultaneously. Applying the optimization to multiple levels, one in-memory run of level 0, and $K-1$ runs of each of levels $0\dots i$ on disk, will get merged to form a single run of level $i+1$ on disk. With $K=2$, this will save about half the I/O operations required otherwise for merging. Furthermore, no level (except level 0) will have more than $K-1$ runs at a time with this optimization.

The average length of a Level 0 run in Stepped-Merge can be increased to double the size of memory by using the run length doubling trick developed for external merge-sort (see, *e.g.*, [Knu73]). All disk accesses, except for the writes to the root relation, in Step 3, are sequential writes. If more than one run is allocated on the same disk, the disk arm may have to move to fetch from or write to different runs. This seek overhead is easily reduced by using large disk buffers, and can be eliminated by using multiple disks, with a careful allocation of runs to different disks. Further implementation details are described in the full version of this paper.

The idea of having intermediate levels of B$^+$-trees and merging them is the same as that used in the LSM tree [OCGO96]. However, Stepped-Merge and the LSM tree differ in significant details; Section 8 describes the differences.

2.2 Queries

Queries can be executed even as data is being organized into runs. In general, there are up to $K-1$ runs at each level $0 \le i < N$. Further, there is newly inserted data in memory that has not yet been inserted into a run. We store the data in memory indexed by the specified key; for simplicity, we assume it is indexed by a B$^+$-tree, although this is not essential and other in-memory tree structures or hash structures may be used.

Instead of looking up a single relation, queries have to (a) lookup the root relation and (b) search the run-index to find (up to) $K-1$ runs at each of the N levels (including the current in-memory run), and perform a lookup on each of these runs. (Assuming that the optimization of merging runs from multiple levels at once is used.) This is an acceptable price if $(K-1)\cdot N$ is not too large, and lookups are relatively infrequent.

2.3 Deletion

Aged records must be deleted from a data warehouse (and possibly archived). Fortunately in most such applications, deletion can be done lazily, and does not have an impact on correctness - either the applications themselves may ensure that logically deleted records are not accessed, or a view mechanism may be used to filter out these records from the applications. Either way, applications do not query data that is old, and could have been deleted. In such an environment deletion can be done efficiently in the background by a batch process that sequentially scans the root relation. If user transactions perform deletions, the idea of having special records to indicate deletions described in [OCGO96] could be used.

2.4 Analysis

We derive an estimate of the number of I/Os incurred for each insertion by the Algorithm Stepped-Merge. Table 1 lists the parameters we use in estimating the I/O costs of various operations. We assume that the root relation is large enough that we can assume its height remains constant during one round of the algorithm. We have the following theorem:

Theorem 2.1 *The total I/O cost of Algorithm* Stepped-Merge *for inserting S pages-full records into a B^+-tree of (final) height h, with L pages and a fanout of d, with s being the size of the final level run and N the number of levels of runs before insertion into the root relation, is*

$$(2+\frac{1}{d})\cdot N\cdot S\cdot T_t + (C_h + \sum_{i=1}^{h} C_i)\cdot(S/s)\cdot(T_s+T_t)$$

where C_i denotes the number of node I/Os from level i of the root B^+-tree, and is obtained as

$$C_i = \lceil\frac{L}{d^{h-i}}\rceil\cdot(1-(1-\frac{1}{\lceil\frac{L}{d^{h-i}}\rceil})^m)$$

Independent Parameters (Both Algorithms)	
M	Size of memory in pages
r	Number of records per page
T_s	Time to seek to a specified (random) location on disk
T_t	Time to transfer one page to/from disk
S	Size of input stream, in pages (in the period of interest)
N	Maximum levels before records are inserted into root relation
Independent Parameters (Stepped-Merge)	
K	Maximum number of runs at a level
d	Average fanout of internal nodes of B$^+$-tree
h	Height of the root relation
Dependent Parameters (Stepped-Merge)	
P_i	Size (in pages) of a run at level i ($P_0 = M$; $P_{i+1} = K \cdot P_i$; So $P_i = K^i M$)
s	Size in pages of final level runs inserted into root relation ($s = K^N \cdot M$)
Independent Parameters (Stepped-Hash)	
K	Maximum number of disk blocks for a hash bucket
R	Number of memory pages reserved for bucket partitioning
Dependent Parameters (Stepped-Hash)	
M_0	Number of memory pages available for managing insertions ($M_0 = M - R$)
X	Number of ways final level bucket is partitioned when inserted into root relation (X = Number of buckets in the root relation / $(M_0 K^N)$)

Table 1: Parameters Used in Analysis

where m is the number of distinct keys in the s pages of records inserted at a time into the root relation. □

The first term in the formula measures the cost of insertion of a record into the various intermediate runs. The second term measures the cost of insertion into the root relation. Details of the derivations are presented in the full version of the paper.

Consider now the cost of direct insertion of records into a B$^+$-tree, without using Algorithm Stepped-Merge. Since the order of insertion of records is random, and the final B$^+$-tree is likely to be much larger than memory, the probability of finding a page in memory is very small. However, to be conservative in our comparison, we will assume that the root node of the B$^+$-tree as well as the next level node are in memory; the rest must be read from disk, and coupled with a write of the leaf page, the cost of inserting S pages worth of records directly into a B$^+$-tree of height h is $S \cdot r \cdot (h-1) \cdot (T_s + T_t)$.

Numerical comparisons of the two costs will quickly demonstrate the benefit of Stepped-Merge over direct insertion, for a wide range of parameter values. This is borne out by experiment as we will discuss in Section 7.

2.5 Cost of Look-Up

Now let us consider the I/O cost of looking up records when using algorithm Stepped-Merge. Instead of looking up a single relation, queries have to look up the root relation and up to K runs at each level. (This is conservative; $(K-1)/2$ is a better average-case estimate.) We assume for simplicity that the index on each run has the same height as the root B$^+$-tree, the root node of each is in memory, and records with the specified key value fit into a single leaf page in each tree. Thus the total cost of a single lookup is $(K \cdot N + 1) \cdot (h-1) \cdot (T_s + T_t)$.

Contrast this with the cost of $(h-1) \cdot (T_s + T_t)$ in a single B$^+$-tree index. For a fixed value of s, N depends on K, and it can be shown that $K \cdot N = K \cdot \lceil log_K(s/M) \rceil$, which is an increasing function of K, for $K \geq 2$. Therefore, it is minimum at $K = 2$. This is experimentally confirmed by our performance analysis in Section 7, which also shows that the actual increase in cost with a small number of levels is quite low.

3 Concurrency Control and Recovery in Stepped-Merge

To implement the Stepped-Merge algorithm in a database system, the transactional issues of concurrency control and recovery must be handled. We deal with these issues in the next two subsections.

3.1 Concurrency Control

There are two aspects to concurrency control for Stepped-Merge — that between *normal* transactions (by which we mean inserts and queries), and between normal transactions and reorganization.

Concurrency control between insertions and queries can be handled in the traditional manner, through key-value locking or interval locking, with a few minor caveats.

For example, some techniques, such as next-key locking [Moh90], are not efficient in our context, since they require inserters to traverse a B$^+$-tree, which incurs I/O that we are trying to avoid.

Alternatively multi-version 2PL can be used to ensure that reads do not interfere with updates (see database textbooks, such as [SKS96], for details). Versioning is particularly simplified because update transactions in our environment merely append new records and thus there exists only one version for each record.

If multi-versioning 2PL is used, records must contain a timestamp corresponding to the time when the transaction that inserted them committed. Read-only transactions read the system timestamp as of when they start, and see all and only relevant records with a timestamp less than their start timestamp.

Concurrency control between normal transactions and index reorganization cannot be handled as easily, since index reorganization is time consuming and potentially involves large parts of the database. We discuss below the interaction between index reorganization and normal transactions, first for updates, and then for queries.

The only type of update performed by normal transactions (in our model) is an insertion into the current in-memory run. Before performing such an insert, the updater finds and shared-locks the pointer to the current in-memory run. The shared-lock is held until transaction commit. Reorganization acquires an exclusive lock on the pointer before transferring the contents of the run to disk; the lock can be released early, after creating an empty in-memory run and updating the pointer in the run-index to point to it.

Queries access the run-index to find what runs they have to search, in addition to the root relation. Concurrency control on the run-index must ensure that:

1. A transaction does not search a given run as well as one of the runs that was merged to get the given run, since a record could then be found twice.

2. A transaction does not miss data in a run because the run got deleted, due to absorption in a higher level run that was accessed by the transaction prior to the absorption.

A naive solution is for query transactions to shared-lock the run-index, and reorganization to exclusive-lock the run-index so that no reorganizations can occur while a transaction is running. However this would result in very poor concurrency since reorganizations take time.

A better alternative is to use versioning of the run-index. When runs are reorganized, instead of updating the existing run-index, a new version is made and is updated. Thus each version of the run-index contains pointers to a consistent set of runs, which cover all data that has been inserted when the run-index version was created, and without any duplication of records in two or more runs pointed to by the version. A pointer to the current run-index cur_index is also maintained. Versioning of the run-index, and can be performed whether or not the data itself is being versioned.

Runs (including the current in-memory run) can be deleted only after (a) all the records in them have been inserted into later runs, (b) the current version of the run-index does not contain the run, (c) no further transactions will find the run, and (d) no transaction is using the run. Straightforward latching mechanisms are used to enforce these rules.

Whereas versioning of the run-index ensures that a consistent set of runs is accessed by a query, it does not ensure that the root relation is accessed in a state consistent with the runs — without additional mechanisms, a query could find records in the root relation that it saw earlier in some run. We have two alternatives. The first solution is based on key-value locking; the basic idea is that queries share-lock the range of key values accessed, while reorganization exclusive locks them. However, this solution provides less concurrency. See the full version of the paper for details.

The second solution, which we call *epoch numbering*, requires insertions into the root relation to be done as follows: all records in some set of runs are inserted into the root relation, and then the set of runs is deleted from (a new version of) the run-index. The epoch number starts from 0, when the first run-index is created. The epoch number is incremented when a new version of the run-index is created such that some set of final-level runs from the previous version have been deleted (because all the records in the runs have been added to the root relation). Thus, multiple versions of the run-index may have the same epoch number.

Further, the records inserted into the root relation have an epoch number stored with them, which indicates the epoch during when they were inserted. The first version of the run-index where the runs have been deleted will have an epoch number higher than the epoch number stored with these records.

Given the above property, a lookup reads the epoch number of its version of the run-index, and simply rejects a record if its epoch is greater than or equal to the epoch of run-index version; any such record would either have been read from the runs in which they were stored earlier, or would have been inserted after the transaction started and due to the serialization requirements they should not be retrieved.

3.2 Recovery for Stepped-Merge

We assume that records are inserted by update transactions, which each insert one or more records. Then each transaction merely inserts its records into the current run transactionally. Logging of the insertion is straightforward. We assume that some recovery technique, such as Aries [MHL+92], is used. The in-memory run is reconstructed from the log records upon recovery from a system crash.

When a new run is created by either merging old runs, or by copying an in-memory run to disk, logging can be suppressed since a crash during the run creation will not lead to information about the records getting lost; on restart recovery, we can delete the partially constructed run and restart the merge/copy. Hence, instead of logging the creation of the run, it is more efficient to create the run without any logging, and flush the run to disk to make it persistent.

Finally, merging of runs into the root relation can be executed as a normal transaction, logging the changes to the root relation.

All versions of the run-index must be recoverable, since (a) they may point to data that has not yet been moved to the root relation, and (b) they may point to runs that no other run-index points to. Hence updates to the run-index must be logged in the usual fashion.

Now consider the logging overhead for our techniques. Each record gets logged once when it is first inserted into the database, and once when it is inserted into the root relation. Thus, the total logging overhead is about twice that of direct insertion into a relation. The I/O for logging is sequential, and only full blocks of data are written. Overall, the extra cost of logging is not a big overhead.

If records are transferred incrementally from a run to the root relation (using the key-value locking technique) the deletion from the run has to be logged as well, so that records get inserted into the root relation at most once.

4 Stepped-Hash Algorithm

The Stepped-Hash algorithm, presented in this section is the equivalent of Stepped-Merge algorithm for the case when the final clustering of data is based on a hash file organization. Data finally resides in a *root hash table*, which is also referred to as the *root relation*. The insertion algorithm is similar in spirit to an external distribution sort and is shown below.

Algorithm Stepped-Hash-Insertion

1. When a record is received, compute its hash value h, and store it in an initial hash table, which we call the Level 0 hash table. That is, add the record to bucket $h \bmod M_0$ of the hash table.

   Each bucket consists of up to K blocks, the last of which is in-memory. In-memory blocks are written out only when they are full, and the blocks for a bucket on disk are kept doubly-linked.

2. When a bucket $B_{i,j}$ of Level i, where $0 \leq i < N-1$ accumulates K full blocks of data, partition the bucket K ways into Level $i+1$ buckets. A record with hash value h is added to bucket $B_{i+1,m}$ where $m = h \bmod (M_0 \cdot K^{i+1})$.

   Each of the K buckets to which records in $B_{i,j}$ may be distributed has one block in the memory buffer. After processing all records of $B_{i,j}$, all K in-memory buffer blocks are flushed to disk, even if they are not full. After records in bucket $B_{i,j}$ have been partitioned, the blocks in $B_{i,j}$ are freed.

3. When a bucket at Level $N-1$, $B_{N-1,j}$, accumulates K blocks, the data is inserted into the root hash table using a hash function $h \bmod (M_0 \cdot K^N \cdot X)$, where X can be any value.

   X can be chosen such that each bucket in the root hash table does not have more than K blocks. X can be dynamically changed, for instance with extensible hashing.

Intuitively, the hash tables form a tree, where nodes are hash tables. During partitioning, records move from a node to its children; which child a record goes to is based on its hash value. Each final level bucket is partitioned X ways when inserting into the root relation. The number of buckets in the root relation is $M_0 \cdot K^N \cdot X$.

An extra data structure, which we call the *bucket-index*, is used to keep track of the last block (on disk) of each bucket. Available memory (M pages) is divided into two parts: R pages are reserved for partitioning of buckets, and the remaining $M_0 = M - R$ pages are available to hold the Level 0 hash table.

Queries calculate the hash value h for the lookup key and search the appropriate hash buckets at each level, before searching the root hash table. The bucket-index is used to find the hash buckets at each level.

4.1 Cost of Insertions

Table 1 lists the parameters we use in the cost estimate for Stepped-Hash. For simplicity, we assume that the directory on each intermediate level has the same height as the directory for the root hash table, and the height is represented by H_d for all the hash tables.

Some of the blocks of a partition at level $i+1$ may overflow as records are inserted into it during partitioning at Level i. The fraction of overflow blocks to K is represented by the term δ.

Theorem 4.1 *The total I/O cost of Algorithm* Stepped-Hash *for inserting S pages-full of records, into a root hash table of directory height H_d is,*

$$S \cdot (T_s + T_t) + \frac{S}{K} \cdot (T_s + K \cdot T_t + (K \cdot (2+\delta) + H_d) \cdot (T_s + T_t)) \cdot (N-1) + \frac{S}{K} \cdot (T_s + K \cdot T_t + (X \cdot (2+\delta) + H_d) \cdot (T_s + T_t))$$

where N is the number of levels before records are inserted into the hash table and K is the maximum number of disk blocks for a hash bucket. □

The three components of the formula above respectively estimate costs for: (a) insertion into the Level 0 hash table, (b) insertion into the intermediate hash tables, and (c) insertion into the root relation. Although the value of δ is non-trivial to compute, we can overestimate it as 1.

Consider now the cost of direct insertion of records into the root hash table without using Algorithm Stepped-Hash. The cost of inserting S pages worth of records directly into a root hash table is $S \cdot r \cdot (H_d + 1) \cdot (T_s + T_t)$.

The analytical formulae here are even more involved than for Stepped-Merge, but once more through numerical substitution it is possible to convince oneself of the benefit of Stepped-Hash over direct insertion into a hash table. This expectation is confirmed by experiments we performed, as we will present in Section 7.

4.2 Cost of Look-Up

Now let us consider the I/O cost of looking up records when using Stepped-Hash. Apart from looking up the root relation, the hash tables on each of the intermediate levels will also have to be looked up. Instead of looking up a single relation, queries have to look up the root relation and up to K blocks at each level. So, we get a total of $K \cdot (N + 1)$ operations to scan the buckets and $H_d \cdot N$ to read the directories. (Level 0 directory need not be read).

Thus the total cost of a single lookup, assuming that records with the specified key value fit into a single bucket and no partitioning is in progress is $((K + H_d) \cdot N + K) \cdot (T_s + T_t)$.

5 Concurrency Control and Recovery for Stepped-Hash

As in the case of Stepped-Merge, concurrency control between transactions is straightforward, and is handled by conventional means such as key-value locking.

Although concurrency control between normal transactions and reorganization in the case of Stepped-Hash bears some similarity to the corresponding scheme for Stepped-Merge, the schemes are different since during reorganization records are inserted into a hash table that already contains other records. For the same reason recovery is also a little more complicated in the case of Stepped-Hash. For lack of space we do not describe either: see the full version of the paper for details.

6 Discussion

Bloom filters (bitmap filters) can be used to avoid looking up many of the runs that do not contain any records for a query key, as is done in, *e.g.*, [SL76].

The direct insert algorithm clearly benefits from a parallel disk system, since such a system supports a larger number of seeks per second. Parallel I/O can also be used with our techniques. The output runs or buckets can be striped across multiple disks, so that they transfer data out in parallel. Since I/O units are large (multiple pages) the main benefit here is from the increased disk bandwidth due to striping, rather than the larger number of seeks that can be supported.

Although our cost formulae give a single time estimate, they can be decomposed into the number of I/O operations (terms multiplied by T_s) and the amount of data transferred (terms multiplied by T_t). The components can then be used to derive time estimates for a parallel disk system, assuming requests are distributed uniformly across all disks.

Both Algorithm Stepped-Merge and Stepped-Hash can handle temporary periods of high insertion loads very well, by simply postponing the merging of runs or partitioning of buckets at intermediate levels. In such a situation inserters are favored at the expense of queries, which have to perform more I/Os at intermediate levels. Conversely, at times when the insert load is less, the number of levels can be dynamically reduced, thereby making queries faster.

Although our techniques are described for a primary index organization which stores records, it can equally well be used for secondary indices, storing index entries instead of records. If our techniques are used on a primary index of a relation, the entries in a secondary index should store the primary key of the record rather than a disk pointer, since the disk location of the record keeps changing.

It is possible to create a hybrid of the B^+-tree and hash schemes: Attach one or more "bins" to the "internal" nodes of a B^+-tree, into which records could be inserted, rather than carrying them all the way to the leaves. When a bin gets full, distribute the contents over the bins of the child nodes (as happens with hash buckets between levels)[1].

7 Performance Study

In order to measure the actual benefits of Stepped-Merge and Stepped-Hash, we implemented them on top of the *Brahma* database storage manager developed at IIT Bombay. We used the existing B^+-tree implementation, which supports bottom-up building of the trees, for run creation. For Stepped-Hash, a simple hash table implementation on top of the database storage manager was used.

We have not yet implemented the concurrency control schemes, but we ran insertions and lookups seri-

[1]This enhancement was suggested by David Maier, to whom we express our gratitude

ally, intermixed with each other. With multi-version concurrency control, queries will cause minimal interference with on-going transactions, so there should be no significant effects due to lock contention. We have not yet implemented the recovery schemes. However, the logging overheads of our schemes are low, and with a separate disk for logs our performance results should not be affected excessively.

The datasets we used for the experiments comprised a sequence of 20-byte records, each with an eight byte primary key consisting of a search key value and a unique identifier to distinguish records with the same key value. Insertions were generated using a uniform random distribution of key values. The page size was fixed at 4KB.

The total buffer memory was 328KB. In the case of direct insert, all 328KB was used for the database buffer, while in the case of Stepped-Merge, 128KB was used for in-memory runs. While the buffer memory size is a small number, it was purposely kept so, to stay in scale with the size of the datasets we have used for experimentation. In Stepped-Hash, M_0 was fixed at 32 buckets and the final hash table was fixed at 8192 buckets.

7.1 Cost of Insertion

Our first set of experiments measured the cost of inserting records. The cost of inserts was measured at each stage as the root relation grew from 0 to 3.2 million records during the course of the experiment.

In Figures 1 and 2 we compare the total cost of record insertion for Stepped-Merge and Stepped-Hash (with different values for K and N) with direct insertion into a B^+-tree and hash table respectively as the size of the B^+-tree/hash table grows. The costs are averages of the insertion cost from the beginning up to the measurement point. The graphs show that the I/Os per record for direct inserts in both cases are significantly higher than the stepped algorithms. Observe that the I/Os per record for direct insertion starts off at around 1 when the height of the root B^+-tree is around 1, and increases quickly to over 2.

Although both the stepped algorithms are much better than direct insert, the Stepped-Merge algorithm had a significantly lower number of I/Os per record, almost half the number of I/Os as Stepped-Hash in the case of $K = 2, N = 2$. The curve for the Stepped-Merge algorithm shows a steady increase in the cost of inserts as the size of the root relation increases, whereas the Stepped-Hash algorithm shows a near constant cost. This is mainly an artifact of our implementation of hashing, where we start off with a fixed number of buckets, which does not grow. The relevant numbers to study are towards the end of the curves, where the number of leaves in the B^+-tree is roughly the same as the number of blocks in the hash table.

As N, the number of intermediate levels, and K, the fanout/fanin increase, the I/Os per record decrease significantly with both the stepped algorithms.

Figures 3 and 4 highlight the cost of inserting into the root relation, ignoring the cost of creating of the intermediate levels, for Stepped-Merge and Stepped-Hash respectively. (Unlike the previous two graphs, the values in these are *not* averages from the beginning but are costs at the measurement point.) It can be seen that these costs are just a little over 1 even at a ratio of 122 of root relation size to final run size (for $K = 2, N = 2$), for Stepped-Merge. The costs are lower for smaller ratios (that is, with higher K and N). The results are similar for Stepped-Hash. In contrast, the cost is about 2 for direct insert even at fairly small sizes of the root relation.

7.2 Cost of Querying

The next set of experiments were designed to find the overhead of querying data using our technique. Batches of 20 record lookups (with records present in the data set) were repeatedly performed as more records were inserted. As a result, for Algorithm Stepped-Merge queries were forced to look up intermediate runs. For the case of Algorithm Stepped-Hash, queries were forced to look up buckets being partitioned.

Twenty queries were run after every 16000 records were inserted, and this was repeated until an additional 1,600,000 records had been inserted into an existing root relation of 3.2 million records. For $K = 2$ and 3, the value of N was varied such that the size of the final run went from 250 pages to 8000 pages. For $K = 4$ it is not possible to get such an N, so we have points at 128 and 512 pages. The Bloom filters used a bitmap per run with 4 times as many bits as records in the first level run.

The results in Figure 5 show that the number of I/Os with our Stepped-Merge technique, especially with the Bloom filter optimization, are within reasonable distance of the number of I/Os with a single B^+-tree lookup for smaller K and N. The number of I/Os for $K = 2$, $N = 3$ without Bloom filters works out to a little over four I/Os per look up, but reduced to 3.125 with Bloom filters, which is about one I/O more than the cost with a single B^+-tree. Comparing the results in Figure 5 and Figure 6 clearly show that Stepped-Hash performs significantly worse on lookups than Stepped-Merge.

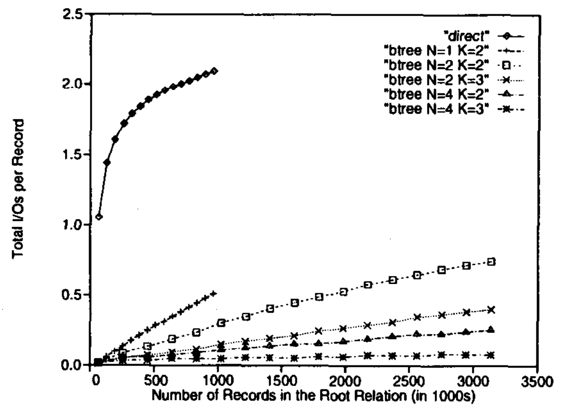

Figure 1: Stepped-Merge: Total Insertion Cost

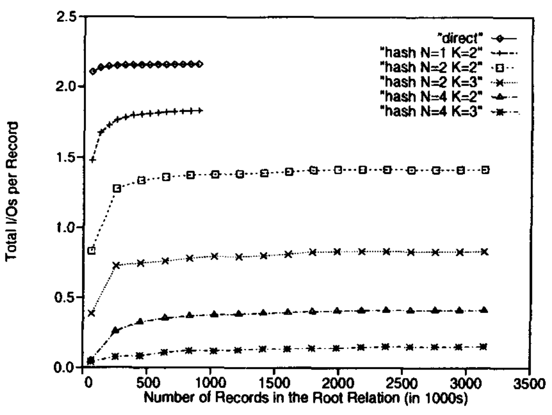

Figure 2: Stepped-Hash: Total Insertion Cost

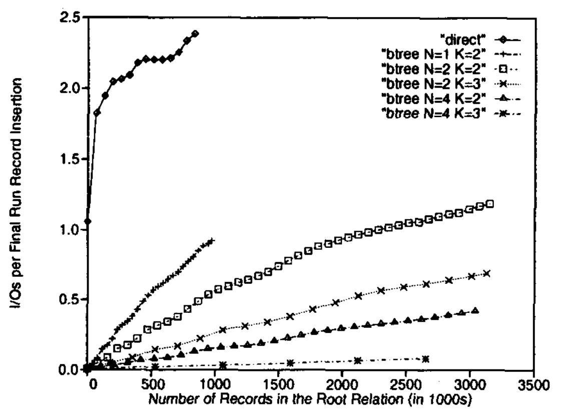

Figure 3: Stepped-Merge: Cost of Insertion into Root Relation

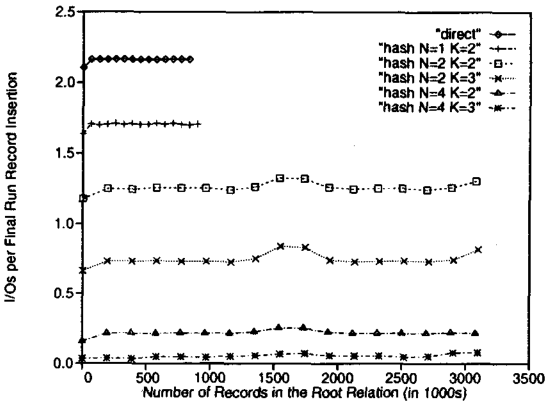

Figure 4: Stepped-Hash: Cost of Insertion into Root Relation

7.3 Sensitivity to Record Size

The final set of experiments were designed to study the sensitivity of insertion costs to the size of the records. As expected, the benefit of our techniques decreases as the number of records that fit in a page decreases. But even with as few as 16 records per page, Stepped-Merge continues to outperform direct insertion; for Stepped-Hash, the crossover point is around 45 records per page. For lack of space we omit details.

In summary, our experimental results demonstrate that Stepped-Merge and Stepped-Hash provide a significant win with respect to insertion costs over the corresponding direct insertion algorithms, in return for a small increase in look-up cost. Stepped-Merge has a slight edge in terms of insertion cost over Stepped-Hash and a considerable benefit in terms of look-up cost.

8 Related Work

The idea of maintaining a log of recent changes separately from the main data file is quite old; see for example [SL76], which discusses differential files. The idea of using Bloom filters has also been explored in [SL76]. However, their goal was not to save I/O as compared to standard structures like B$^+$-trees, but to avoid changing the main file. They do not consider issues of multi-level organization of differentials, and concurrency control and recovery issues.

Lists of updates-to-be-applied are maintained by online index construction/reorganization techniques (*e.g.*, [SC91]). These, however, are temporary structures, existing only while the index is being constructed/reorganized, and are not used by queries; improving insert speeds is not a goal.

The work that is most closely related to ours is the LSM-tree, described by O'Neil *et al* [OCGO96]. Our first technique, Stepped-Merge, although developed independently, can be seen as a variant of the LSM tree: both are based on the same core idea of a multi-level organization of B$^+$-trees. Our hash-based algorithm is, however, novel.

An important difference is that the LSM tree has a single B$^+$-tree at each level whereas Stepped-Merge has up to K B$^+$-trees at each level. The LSM tree is therefore better for queries, since only one tree need be looked up at each level whereas K trees may need to be looked up in Stepped-Merge. However, the LSM tree is likely to be costlier for inserts since data may

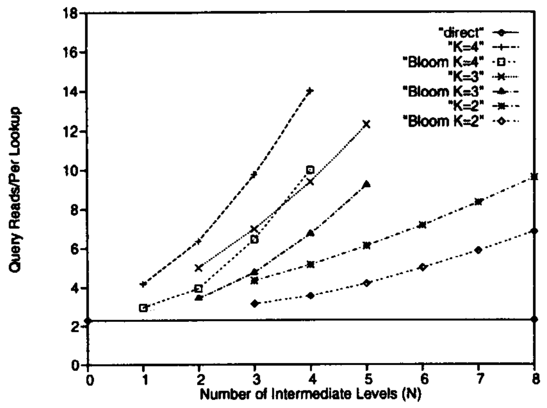

Figure 5: Stepped-Merge: Cost of Lookups

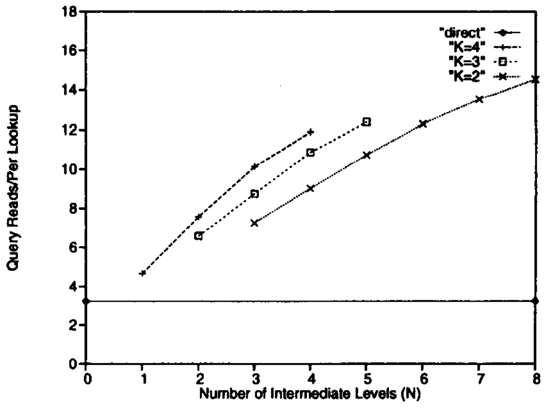

Figure 6: Stepped-Hash: Cost of Lookups

be read and written back up to K times at each level. [OCGO96] studies issues of how much memory and how many disks should be used to support a given load at the cheapest cost; we do not consider this issue, and measure instead the number of I/O operations and data volume to be transferred. The analytical formulae for **Stepped-Merge** and the LSM tree therefore measure different quantities, and cannot be compared directly.

The LSM tree handles updates, whereas we have not addressed updates so far. Conversely, we have described a concurrency control scheme, whereas [OCGO96] does not — it only outlines features that a concurrency control scheme must have. Transfer of data from one level to another is more incremental for the LSM tree but the price paid is that concurrency control is more complicated. We believe our recovery technique makes fewer changes to standard recovery techniques and should be easier to implement.

Unlike [OCGO96], we have presented a performance study of our techniques based on an actual implementation. Future work includes implementing the LSM tree in our system and empirically comparing its performance with **Stepped-Merge**.

9 Conclusions and Future Work

We studied two techniques to cluster data incrementally as it arrives, one based on sort-merge and the other on hashing. We have presented efficient concurrency control and recovery schemes for both techniques. We have demonstrated the benefits of our techniques both analytically and through an empirical performance study of an actual implementation. One contribution of this paper has been to show that a well-designed sort-merge based scheme performs better than hashing.

We believe it should be reasonably easy to integrate our techniques into an existing database system. Future work includes extending our techniques beyond insert-only environments, to allow updates of existing data. We believe our techniques will play an important role in the design of data recording systems and data warehouses in the future.

References

[JMS95] H.V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. The chronicle data model. In *Procs. of the ACM Symp. on Principles of Database Systems*, 1995.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Vol.3 — Sorting and Searching.* Addison-Wesley (Reading MA), 722pp., 1973.

[Moh90] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *IBM Almaden Res.Ctr, Res.R. No.RJ7008, 27pp.*, March 1990.

[MHL+92] C. Mohan, D. Haderle, Bruce Lindsay, Hamid Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), March 1992.

[OCGO96]
Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree. *Acta Informatica*, 33:351-385, 1996.

[OW93] Patrick O'Neil and Gerhard Weikum. A Log-Structured History Data Access Method (LHAM). High-Performance Transaction Systems Workshop (HPTS) 1993.

[SKS96] A. Silberschatz, H. Korth and S. Sudarshan *Database System Concepts.* McGraw Hill, 3 edition, 1997.

[SL76] D.G. Severance and G.M Lohman. Differential files: Their applications to the maintenance of large databases. *ACM Transactions on Database Systems*, 1(3):256–367, September 1976.

[SC91] V. Srinivasan and M. J. Carey. On-line index construction algorithms. *Proc. High Performance Transaction Systems Workshop*, Sep. 1991.