Information Systems Vol. 16, No. 3, pp. 345–356, 1991
Printed in Great Britain. All rights reserved

0306-4379/91 $3.00 + 0.00
Copyright © 1991 Pergamon Press plc

ANALYSIS OF A DEFERRED AND INCREMENTAL UPDATE STRATEGY FOR SECONDARY INDEXES

EDWARD OMIECINSKI, WEI LIU and IAN AKYILDIZ
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, U.S.A.

(Received 22 April 1989; in revised form 18 July 1990; received for publication 21 February 1991)

Abstract—Many relational database systems use secondary indexes to reduce the access cost of retrieving data in response to a user's query. However, a secondary index incurs an additional cost due to the update maintenance of the index. In some cases, this cost may be greater than the cost to update the desired tuples. This paper examines a deferred index update strategy which does an incremental update of the index. The approach introduced, which uses a differential file, can reduce the cost of updating a secondary index by slightly increasing the cost that will be associated with searching the secondary index. This is true as long as the differential file size does not become too large. As such, a model is presented for solving the distribution of the size of the differential file. The maximum size of the differential file is predicted by interpreting this distribution. In addition, the analytical results are compared with simulation results.

Key words: Differential file, secondary index, incremental update, relational database

1. INTRODUCTION

The advantages of using an index for the retrieval of tuples from a relational database is well known [1, 2]. The benefit of having indexes is offset by the cost of maintaining the indexes in the face of updates. For some updates, the cost of maintaining the index may be greater than the cost of updating the desired set of tuples in the relation [3]. In this paper, we describe a strategy for updating secondary indexes which can be classified as deferred and incremental. By deferred, we mean that the index is not updated when the user's update statement is executed. Rather, the appropriate changes are simply recorded for later update of the index. By incremental, we mean that only certain recorded changes will be applied to the index at a given time. We stress that only the index updates are deferred and that the tuple updates are performed by the user's update statement. We provide a performance analysis of our method and show that it can reduce the cost of updating the index at the slightly increased expense of searching the index.

As pointed out in [3], the update maintenance cost (i.e. the cost of updating the appropriate tuples and indexes) are dependent on the following: the type of scan used to search for the tuples to be modified and the type of predicates specified in a user's update statement. Simple update cost formulas are presented in [3]. In general, an update of a relation and its associated indexes is performed by first choosing a relation scan method; then retrieving and modifying the desired tuples; and finally by modifying the index entry for each modified tuple, if necessary.

We assume as in [3], that access to a relation in our database is either through a sequential file scan or through indexes. We assume that each index is organized as a $B+$ tree, where at the leaf pages, each key value is followed by a sorted list of *TIDs*, i.e. identifiers of the tuples where the key value appears. There are two basic methods for retrieving tuples via indexes. One is the single-index method [2], where one of the available indexes on the relation is used. For this approach, one tuple identifier is selected at a time (from the index) based on the specified selection predicate. Afterward, the corresponding tuple is retrieved and checked with the other selection predicates, if any. The other approach is the *TID* intersection method [3]. With this method, multiple indexes are searched and a list of *TIDs* of qualifying tuples is formed for each index. These lists are later intersected and the corresponding tuples are retrieved and checked with any remaining selection predicates. In this paper, we consider only the updating cost of the indexes and not the updating cost of the tuples. In addition, we assume that the single index method is used.

At this point, we would like to discuss the index update costs. When evaluating the cost of an index update, it is important to distinguish the order in which the leaf pages of the index are examined. There are two cases to consider [3]. The first case is when the modification is done on an index following a scan that is unordered with respect to the order of *TID* groups in the index leaves. For our situation, this happens when the modification is done to a secondary index following either a sequential file scan or an index scan on some other index. The second case is when the modification is done on an index following a scan that is ordered with respect to the *TID* groups in the index leaves. For our situation, this happens when the secondary index is modified and only one key value is used or when the index that is modified is used as the scan method. For the relational database system, System R [2], the latter case cannot occur since it would lead to (possibly) accessing a tuple multiple times. This is due to making changes to the index while it is being scanned. Since we use a deferred update strategy, the index which is to be modified can be used as the scan method. This is also true in INGRES [4].

The cost [3], α, in number of I/O accesses, to fetch and update a new leaf page is given as $(2 + \lambda)$. The constant 2 is due to one access to read the leaf and one access to rewrite it. The term λ is the number of accesses due to reading intermediate level pages in the index. The value of λ depends on the buffer management policy [1].

An index update may be viewed as consisting of two parts: a delete followed by an insert. For example, when a secondary key of a tuple changes value, the *TID* for that tuple is removed from the *TID* group for the *old-key value* and is inserted into the *TID* group for the *new-key value*. A typical update changes the value of an attribute of a tuple to a constant or to a new value which is calculated from some expression. Assume that the *new-key value* is calculated using the current value of the key, E tuples are to be updated and the key value of successive tuples, to be updated, is different. For the insert part, this implies that each *TID* will be assigned to a different leaf page. The I/O cost, for the insertion, is αE. If the *new-key value* is simply assigned a constant for all the updated tuples but the remaining assumptions are the same, then we assume that the *TIDs* will fit on the same leaf page. The I/O cost, for the insertion, for this case is just α. Now to consider the delete part of the update. If we are doing an unordered scan and the successive E *TID* deletes are done from different leaf pages, then the I/O cost is αE.

We illustrate a (somewhat) typical update and its associated cost using the formulas presented above. We use the following relation:

EMPLOYEE(NAME,NUMBER,DEPARTMENT,SALARY,TELEPHONE).

We use an SQL update statement to give a 10% salary increase to employees working in the "Shoe" department. The update is shown below:

```
UPDATE EMPLOYEE
SET SALARY = SALARY *1.1
WHERE DEPARTMENT = 'SHOE'
```

Let us assume the following conditions:

— there is a secondary index on SALARY
— there is a secondary index on DEPARTMENT which is chosen as the access path
— the number of tuples to be updated is 50, i.e. $E = 50$
— the height of each B+ tree index is 3, if the root of each index resides in main memory then $\lambda = 1$ and thus $\alpha = 3$

With the above assumptions, the number of page accesses required to perform the update of the SALARY index would be 300, i.e. $2\alpha E$. Now, let us look at the tuple update cost. The DEPARTMENT index is searched to find the *TIDs* of the qualifying tuples and that this accounts for 2 I/Os, as long as the 50 *TIDs* are on one leaf page. Nex, the 50 qualifying tuples are read and rewritten. This costs 100 I/Os, if the tuples are on distinct pages. Thus, the total tuple update cost is only 102. This is approx. 1/3 of the cost associated with updating the secondary index. From this example, the motivation for reducing the update cost of secondary indexes is clear.

2. PREVIOUS RESEARCH

In [5], the use of a differential file is proposed. The differential file stores all updates, leaving the main file unchanged. For a query, it needs to be decided whether the requested record is in the differential file. If this decision is made incorrectly then the differential file is searched as well as the main file and a double access is made. Eventually, the differential file and main file are merged. By consolidating changes in this manner, it is possible to reduce backup costs and speed the process of database recovery. In [5], they were interested in designing a Bloom filter which would indicate whether a given record could be found in the differential file. As the differential file grows, the discrimination power of the Bloom filter becomes worse. This causes an increased number of unnecessary searches of the differential file and precipitates the file merging. In [6], the differential file is studied in the context of backup and recovery. They present an analytical model and a design algorithm. In [7], differential files are used to support hypothetical relations.

In [8, 9], the use of a differential file for updating a main file which is organized as a B+ tree is presented. In both [8] and [9], the approach is to perform an efficient batch update of the tree-structured main file. They both show that sufficient savings (in number of page accesses) can be realized if the updates are batched instead of executed individually. Both of their approaches apply only to a primary index file. In [8], they are concerned with determining the optimal time at which the batch update should take place. In [9], they use a tree structure for the differential file and assume that it can reside in main memory.

Instead of performing a batch update or file reorganization, others [10] have proposed to do incremental updating of the main file. By incremental updating, we mean that one or more of the main file records will be updated at various times, i.e. not all records are updated at once. Usually an update will be triggered by a query. That is, if some of the retrieved records are ones that need to be updated, then the update will take place at that time and the updated records will be rewritten.

In [10], they advocate an incremental update policy when updates involve a set of records and since the retrieval and update of records overlap, a reduction in I/O costs can be expected. In [10], the differential file does not contain the updated records, rather the update procedure. So, when a record is retrieved, a filtering mechanism determines whether there is an associated update procedure(s) in the differential file. If there is, then the update is done at that time. This approach is for the deferred update of the actual tuples and nothing is said about the update of existing indexes. Also, no analytical model or simulation is provided to substantiate the effectiveness of their approach.

In [11], an incremental and deferred update strategy is presented for the maintenance of text indexes. Their emphasis is on concurrency control methods and efficient data structures for the deferred update information. In particular, they examine the use of a bit vector, a transaction oriented key-word list and a key-word oriented transaction list as data structures. These data structures can be used for storing lock information which is used in detecting possible inconsistencies as well as for storing the deferred update information. They also compare the storage needed for these three data structures in five different environments.

The architecture for integrating a mainframe database system and a large number of workstation database systems is presented in [12]. One feature of this system is the deferred index update strategy. When a query needs to use an outdated index, it updates it first by batching, sorting and merging all updates together [12]. In a similar vein, a deferred view materialization strategy is presented in [13] which uses a differential file and updates a materialized view just before data is retrieved from it. A performance analysis is also provided in [13]. These two approaches are different from what we term incremental because all of the updates in the differential file are performed whenever the index or view is accessed. This may be reasonable for materialized views but we do not believe that it is for indexes, since one query would shoulder the costs of all the updates.

3. DEFERRED AND INCREMENTAL INDEX UPDATE

Our deferred and incremental index update approach uses a differential file which grows as updates are executed and shrinks as queries are executed. A record in the differential file consists of the following attributes: a *TID*, the *old-key value* and the *new-key value*. It looks similar to a record in the database log, which is used for recovery. For our purpose, we assume that the differential file is a simple sequential file with an update inplace capability. This is a suffcient file structure as long as the differential file can reside in main memory. In addition, a record with a particular *TID* value occurs only once in the differential file, no matter how many times the corresponding tuple has been updated. We do not need to keep a trace of all changes made to a particular tuple's key value since the update of the tuple has already taken place. We simply need to know what *TID* list currently contains the particular *TID* (for deletion) and what *TID* list should now contain the particular *TID* (for insertion). We can employ a standard locking protocol to insure consistency.

Alternate differential file structures, such as tree-based or hash-based files are possibilities as well as having separate differential files for the deletion part of an update and insertion part of an update. These options will be explored in future work.

The update procedure is simple and is shown below:

1. Access tuples to be modified (via a single index or file scan). If access is through the secondary index, then the query procedure (which follows) must be invoked.
2. Modify tuples.
3. For each indexed attribute, of the modified tuples, that has been modified:
   if the *TID* for this tuple does not appear in the differential file then write a new record to the differential file
   else update the *new-key value* for an existing record in the differential file

A naive query procedure, which uses the secondary index as the access method, is shown below.

1. Follow the appropriate path from the root to a leaf page of the index (the leaf page contains key values and their associated *TID* lists).
2. Search the differential file for key values which match the requested key values contained on the leaf page.
   - 2.1. If the *old-key value*, from a record in the differential file, matches then delete the corresponding *TID* value from the *TID* list for the key and set the *old-key value* to null.
   - 2.2. If the *new-key value*, from a record in the differential file, matches
     then if the *old-key value* is null
     then
       insert the corresponding *TID* value in the *TID* list for the new key and set the *new-key value* to null
     else
       save the *TID* value in a temporary *TID* list.
   - 2.3. If both both the *new-key value* and the *old-key value*, for a record in the differential file are null, then delete that record from the differential file.
3. If necessary, access the next leaf page and go to Step 2.
4. Retrieve the tuples for the *TID* lists which have been found (including the temporary *TID* list). The *TID* lists contain any modifications as performed in Step 2.

We now present a simple example, which illustrates the query and update procedures. We have the following relation, PARTS(PART #, CITY, PARTNAME) whose contents is illustrated in Fig. 1. We have a secondary index on CITY, which is illustrated in Fig. 1. There is also an index

SECONDARY INDEX

CITY	TID LIST
LONDON	002, 006
NEW YORK	001, 004
PARIS	003, 005

PARTS RELATION

TID	PART#	CITY	PARTNAME
001	101	NEW YORK	NUT
002	102	LONDON	BOLT
003	103	PARIS	SCREW
004	104	NEW YORK	COG
005	105	PARIS	CAM
006	106	LONDON	WASHER

Fig. 1. Index and relation contents.

TID	UPDATED TUPLE		
001	101	LONDON	NUT

DIFFERENTIAL FILE

TID	OLD KEY	NEW KEY
001	NEW YORK	LONDON

Fig. 2. Modifications after update is executed.

on PART #, but since its contents will not change we defer from showing it. The differential file is initially empty.
Assume that the following update is executed:

UPDATE PARTS
SET CITY = 'LONDON'
WHERE PART # = 101

Since an index on Part # exists, we assume that the query optimizer would utilize the index in the access plan produced for the update, although this is not mandatory. The desired tuple is accessed via the PART # index and the tuple is modified, i.e. the CITY value is changed from NEW YORK to LONDON. Since the index update is deferred, the CITY index is neither searched nor modified at this time. However, a record is stored in the differential file. The results are illustrated in Fig. 2.
Now, suppose the following query is executed:

SELECT*
FROM PARTS
WHERE CITY = 'NEW YORK'

For the preceding query, a typical query optimizer would use the CITY index as the access path. That is, the TID list for NEW YORK would be retrieved. In addition, the differential file would be searched. In this case, the old key value for a record in the differential file matches the desired value. As indicated in Step 2.1 of the query procedure, the TID value i.e. 001, would be deleted from the TID list for NEW YORK. The remaining TID values would be used to retrieve the desired tuples. In addition, the old key value corresponding to TID value 001, in the differential file would be set to null. This is shown in Fig. 3.
Now, suppose the following query is executed:

SELECT*
FROM PARIS
WHERE CITY = 'LONDON'

Once again, we assume that the access path chosen by the query optimizer is the CITY index. As such, the TID list for LONDON is accessed. The differential file is searched and the desired key value is matched with the new key value for a record in the differential file. As described in Step 2.2 of the query procedure, the *TID* value, i.e. 001, is inserted into the *TID* list for LONDON and the new key value in the differential file record is set to null. Since both the old key and new key values are null, the differential file record is deleted, i.e. the deferred update has been completed. The end result is shown in Fig. 4.
We should note that if the execution order of the above two queries were commuted, then the differential file would still contain a record for the tuple whose *TID* value is 001. The old key value for that differential file record would be null but the new key value would still be LONDON. This

SECONDARY INDEX

CITY	TID LIST
LONDON	002, 006
NEW YORK	004
PARIS	003, 005

DIFFERENTIAL FILE

TID	OLD KEY	NEW KEY
001	Ø	LONDON

Fig. 3. Modifications after NEW YORK query is executed.

SECONDARY INDEX

CITY	TID LIST
LONDON	001, 002, 006
NEW YORK	004
PARIS	003, 005

Fig. 4. Modifications after LONDON query is executed.

can be seen from Step 2.2 of the query procedure. The reason for doing this will be explained shortly. Hence, to have that record deleted from the differential file, another LONDON query would have to be executed.

The query procedure can be improved by reducing the number of scans of the differential file. By using buffer space efficiently, a group of leaf pages can be read into the buffer and one scan of the differential file can be made for each group. From the above search algorithm, one can see that if a record from the differential file satisfies only Step 2.1, then the *TID* from that record will be deleted from the index. If a record from the differential file satisfies only the first if-condition of Step 2.2, then the *TID* will be used for record retrieval but not for index modification, at this time. We do this so that only one record in the differential file is needed per tuple, regardless of the number of key value changes that have taken place on any given tuple. This is proven in Theorem 1. Thus, the maximum differential file size, in records, is equal to the relation size in records (tuples). Although, a record in the differential file should be much smaller than the tuple size. Remember, that the actual tuple update has taken place, it is just the index update which is deferred.

Only, when Steps 2.1 and 2.2 are done for a given record in the differential file will the update be complete. Hence, a particular *TID* will occur in the index at most one time and possibly be absent from the index during the deferred update. This is shown by Lemma 1, whose proof appears in the Appendix. Also, there is no inconsistency since the differential file is searched before records are retrieved.

Lemma 1

According to the search and update algorithm, a given *TID* in the differential file, e.g. *TID i*, will appear in 0 or 1 *TID* lists in the index.

Before we present Theorem 1, whose proof appears in the Appendix, we first want to give the following definition of a consistent view of a relation (file). A consistent view is defined in the context of a search for a particular key.

Definition

If the *TIDs* retrieved by searching the index and differential file correspond to the tuples in the relation that contain the desired search key, then we say that there is a consistent view of the relation (file).

Theorem 1

The deferred update scheme presents a consistent view of the relation, regardless of the number of updates.

We should note that if the second if-condition was not included in Step 2.2 of the search, i.e. checking if the *old-key value* is $\emptyset$ before modifying the *new-key value*, then an inconsistent view could arise. Consider the following record in the differential file: *TID* = 1001, *old-key* = Chicago and *new-key* = Miami. Let us search for the key value equal to Miami. Suppose, we set the *new-key* equal to $\emptyset$ in the differential file and insert *TID* 1001 in the Miami *TID* list. Now *TID* 1001 occurs in two *TID* lists. Now, suppose there is an update for *TID* 1001 such that the key value should be changed to Atlanta. For a subsequent search for the key value equal to Miami, *TID* 1001 would still be retrieved since it appears in the Miami *TID* list, although the key value in the corresponding tuple is Atlanta. We would not know that *TID* 1001 should be deleted from that list. Hence, an inconsistent view of the relation is seen. An alternative which avoids this problem is to keep a differential file record for each individual update associated with a *TID*. However, the size (in records) of the differential file could possibly become a great deal larger than the number of records in the relation, i.e. our worst case situation.

The usefulness of our incremental and deferred index update approach hinges on the size of the differential file. If the differential file is small enough to reside in main memory, then there is a small penalty associated with searching the differential file. The searching consumes CPU time but no I/O time. Again, the differential file will be searched for each query which does an index scan of the associated secondary index. For the actual update cost, we no longer include the cost to search the index since the index search is done by the query. However, the additional overhead of writing a leaf page(s) back to secondary storage, i.e. updating the *TID* lists, is attributed to the query. Overall, we save the double search of the index, i.e. there is no need for the update procedure to search the index for the old key value nor for the new key value.

Since the differential file size is important, in the following section, we develop an analytical model to determine the likelihood that the differential file reaches a particular size. Also, in a subsequent section, we compare the analytical model with results from our simulations.

4. ANALYTICAL MODEL OF DIFFERENTIAL FILE SIZE

The performance analysis of database systems has become an important research topic in the last decade. Analytical models are used in describing the system's behaviour and predicting the performance [6, 14, 15]. Executing transactions against a database system can be viewed as a stochastic process. As such, a direct and inexpensive way to predict performance measures is with an analytical model. In the context of our problem, we develop an analytical model which describes the size of the differential file. With an update transaction for the differential file, some population, i.e. number of records, arrives and with a search transaction, some population departs. If the arrival rate does not exceed the departure rate, then the differential file will enter an equilibrium (steady) state. The analytical model is solved iteratively and the iteration converges if the system is in an equilibrium (steady) state. In case of a non-equilibrium state, i.e. the arrival rate is greater than the departure rate, the differential file will not stabilize. This situation can be detected in the iteration by observing that the current distribution is centered on the maximum size. We can conclude that in this case, the records will eventually accumulate to reach the upper limit of the size of the differential file, i.e. one record for each record in the relation. We assume that the relation itself is in an equilibrium state, i.e. the number of tuples in the relation is fixed, since we do not consider insertions nor deletions in our model. Incorporating insertions and deletions in an area of future work.

Let us denote $P[size = s]$ as the probability that the differential file size is s, for $s = 0, 1, \ldots, S_{max}$ and $P[i \longrightarrow s]$ as the probability that the size of the differential file before the transaction is i, and after the transaction it is s. According to our update and search scheme, a record in the differential file contains three fields: the *TID*, the *old-key value* and the *new-key value*. For a single update, a new record will be added to the differential file if and only if the *TID* of the record being updated is not contained in any of the existing differential file records. For a search transaction (single key or range of keys), if the *old-key value* for a differential file record is matched, then this value is set to $\varnothing$ and the indexed file is modified. A match on the *new-key value* will result in the record being deleted if and only if the *old-key value* in that record is $\varnothing$. Therefore, $P[i \longrightarrow s]$ is determined by the number of records with the *old-key value* of $\varnothing$.

To characterize the above phenomenon, we use $P[empty\,|\,s]$ to denote the proportion of records with an empty *old-key value* when the differential file contains s records. It is obvious that for each record in the differential file of size s, the probability that its *old-key value* is empty is also $P[empty\,|\,s]$. The search transaction is treated as if it consisted of two steps. First, when the *old-key value* is matched, it is set to $\varnothing$. This results in $P[empty\,|\,s]$ being changed to a new value, $N(s,j)$, where $j = 1$ denotes a single key search and $j = 2, \ldots, R_{max}$ denotes a range search consisting of two keys through R_{max} keys. The variable R_{max} denotes the maximum number of keys in a range query. The calculation of $N(s,j)$ will be given later. The second step is to match the *new-key value*. It should be noted that $P[empty\,|\,s]$ changes from transaction to transaction. For example, when one record is added to an empty differential file, then $P[empty \mid 1] = 0$. After a search $P[empty \mid 1]$ will become either 1 or remain 0, depending on whether the *old-key value* is matched or not. Another consideration is, that if $P[size = s] = 0$, then $P[empty \mid s]$ is undefined. In other words, if the differential file has not reached a size of s, then it is meaningless to consider that case. The above two considerations direct us in the calculation of $P^{(t)}[empty \mid s]$ where t denotes that t transactions have been processed. We use $P^{(t-1)}[empty \mid s']$ for all stages s' that can reach s. That is, the differential file can change from a state having s' records to a state having s records. Also, $P^{(t-1)}[empty \mid s']$ is undefined if $P^{(t-1)}[size = s'] = 0$.

Next, we develop an iteration model to calculate $P[size = s]$ based on $P[empty \mid s]$. After computing the distribution for the size of the differential file, we can compute the maximum size of the differential file after t transactions have been processed. This parameter is crucial in determining whether there exists enough main memory to contain the whole differential file and thus speed up the update and search transactions considerably. The maximum size can be predicted from the size distribution as follows. It should be the first value following the average size with probability less than $1/t$. For example, if $P[size = s] = 10^{-6}$, then it means that it is unlikely that the size s will be reached once within 100,000 transactions since the probability is $1/10^6$.

We use the following notation in our model:

$Prob[up]$ = probability of an update
$Prob[ss]$ = probability of a single key search
$Prob[rs]$ = probability of a range search
S_{max} = maximum size (in records)
R_{max} = maximum number of keys in range search
K_{max} = number of distinct key values in the relation

Initially, we set $P^{(0)}[size = 0] = 1$ and $P^{(0)}[size = i] = 0$, for $i = 1, \ldots, S_{max}$. All $P^{(0)}[empty \mid s]$ are undefined for $s = 1, \ldots, S_{max}$. Iterate over equations (1–5), i.e. $t = 1, \ldots, ceil$, until the difference between the average size of the differential file in two successive iterations is less than an epsilon ($\epsilon = 10^{-4}$) value.

The probability that after t transactions, the differential file size is s, is the sum of the product of the following two probabilities: the probability that the differential file was of size i after $t-1$ transactions and the probability that the next transaction causes a transition into a file size of s records. This is shown in equation (1):

$$P^{(t)}[size = s] = \sum_{i=0}^{S_{max}} P^{(t-1)}[size = i] \cdot P^{(t-1)}[i \rightarrow s], \tag{1}$$

where $P^{(t-1)}[i \rightarrow s]$ is computed by equation (4).

$P^{(t)}[empty \mid s]$, in equation (2), is defined in a somewhat similar manner, except now, we have to take into account $Q^{(t-1)}[i \rightarrow s]$ for every i-value that can lead us to a state with s records. $Q^t[i \rightarrow s]$, as will be defined in equation (5), is the proportion of records whose *old-key value* is $\varnothing$ for a differential file that has i records after $t-1$ transactions and has s records after t transactions:

$$P^{(t)}[empty \mid s] = undefined, \quad \text{if } P^{(t)}[size = s] = 0, \tag{2a}$$

$$P^{(t)}[empty \mid s] = \frac{\sum_{i=0}^{S_{max}} P^{(t-1)}[size = i] \cdot P^{(t-1)}[i \rightarrow s] \cdot Q^{(t-1)}[i \rightarrow s]}{\sum_{j=0}^{S_{max}} P^{(t-1)}[size = j] \cdot P^{(t-1)}[j \rightarrow s]}, \tag{2b}$$

where $Q^{(t-1)}[i \rightarrow s]$ is defined in equation (5).

The following equation yields the new proportion of records with an *old-key value* of $\varnothing$. This is relative to the differential file size and the number of keys which appear in a search request. This formula will be used in defining $Q^t[i \rightarrow s]$ as shown shortly.

$$N^{(t-1)}(s, j) = P^{(t-1)}[empty \mid s] + (1 - P^{(t-1)}[empty \mid s]) \cdot \frac{j}{K_{max}}, \tag{3}$$

for $s = 1, \ldots, S_{max}$ such that $P^{(t-1)}[size = s] > 0$ and $j = 1, \ldots, R_{max}$.

Equations (4a–d) depict the four possibilities for a transition in the differential file size. Formula (a) indicates that we have an update for a new *TID*, hence, the file size is increased by one record. Formula (b) indicates that we could have an update on an existing record in the differential file or a query (single key or range) that matches records which have a non-empty *old-key value*. Formula (c) represents the case where there is a search that matches the *new-key value* for δ records which have an empty *old-key value*. So, those records can be deleted:

$$P^{(t-1)}[s \rightarrow s+1] = Prob[up] \cdot \left(1 - \frac{s}{S_{max}}\right), \quad (\text{if } P^{(t-1)}[size = s] > 0), \tag{4a}$$

$$\begin{aligned} P^{(t-1)}[s \rightarrow s] = {} & Prob[up] \frac{s}{S_{max}} + Prob[ss]\left(1 - N^{(t-1)}(s,1) \cdot \frac{1}{K_{max}}\right)^{s} \\ & + Prob[rs] \cdot \sum_{j=2}^{R_{max}} \frac{1}{R_{max}-1}\left(1 - N^{(t-1)}(s,j) \cdot \frac{j}{K_{max}}\right)^{s}, \\ & (\text{if } P^{(t-1)}[size = s] > 0), \end{aligned} \tag{4b}$$

$$\begin{aligned} P^{(t-1)}[s \rightarrow s-\delta] = {} & Prob[ss] \cdot \binom{s}{\delta}\left(N^{(t-1)}(s,1) \cdot \frac{1}{K_{max}}\right)^{\delta} \cdot \left(1 - N^{(t-1)}(s,1) \cdot \frac{1}{K_{max}}\right)^{s-\delta} \\ & + Prob[rs] \cdot \sum_{j=2}^{R_{max}} \frac{1}{R_{max}-1}\binom{s}{\delta}\left(N^{(t-1)}(s,j) \cdot \frac{j}{K_{max}}\right)^{\delta}\left(1 - N^{(t-1)}(s,j) \cdot \frac{j}{K_{max}}\right)^{s-\delta} \\ & (\text{if } P^{(t-1)}[size = s] > 0 \ \text{ and for } \delta = 1, \ldots, s) \end{aligned} \tag{4c}$$

$$P^{(t-1)}[i \rightarrow j] = 0 \qquad \text{for } \textit{all other cases}. \tag{4d}$$

Again, we consider the possible state transitions for the differential file in equations (5a–e). Except, that here we calculate the proportion of records in the differential file whose *old-key value* is $\varnothing$. The value for $Q^{(t-1)}[i \rightarrow j]$ depends on the proportion of records whose *old-key-value* matches a search key, as shown in formulas (b) and (c).

$$Q^{(t-1)}[0 \rightarrow 0] \quad \textit{is not defined} \text{ and } \textit{is never used}, \tag{5a}$$

$$Q^{(t-1)}[0 \rightarrow 1] = 0 \tag{5b}$$

$$\begin{aligned} Q^{(t-1)}[s \rightarrow s] = {} & Prob[up] \cdot P^{(t-1)}[empty \mid s] + Prob[ss] \cdot N^{(t-1)}(s,1) \\ & + Prob[rs] \cdot \sum_{j=2}^{R_{max}} \frac{1}{R_{max}-1} N^{(t-1)}(s,j), \quad (\text{if } P^{(t-1)}[size = s] > 0), \end{aligned} \tag{5c}$$

$$\begin{aligned} Q^{(t-1)}[s \rightarrow s-\delta] = {} & \frac{Prob[ss]}{Prob[ss] + Prob[rs]} \frac{s \cdot N^{(t-1)}(s,1) - \delta}{s - \delta} \\ & + \frac{Prob[rs]}{Prob[ss] + Prob[rs]} \cdot \sum_{j=2}^{R_{max}} \frac{1}{R_{max}-1} \frac{s \cdot N^{(t-1)}(s,j) - \delta}{s - \delta}, \\ & (\text{if } P^{(t-1)}[size = s] > 0), \end{aligned} \tag{5d}$$

$$Q^{(t-1)}[s \rightarrow s'] = 0, \qquad \text{for } \textit{all other cases}. \tag{5e}$$

5. COMPARISON OF ANALYTICAL AND SIMULATION RESULTS

In order to validate our analytical model, we simulated the system. The goal of our simulation is to keep track of the differential file size after each individual transaction as well as the maximum and average sizes. We divide transactions into two classes: update and search. The search class is further divided into single key search and range search. The update class can be divided into four types. However, it appears to be difficult to devise an analytical model which would take into account all four types. Our approach is to develop a model for the most straightforward type of update and then to later make additions to our model to handle one additional class at a time.

Table 1. Results for analytical model and simulation

		Maximum file size		
Update (%)	Query (%)	Simulation	Model	% Difference
10	90	29	33	13.79
20	80	57	60	5.26
30	70	84	90	7.14
40	60	119	137	15.13
50	50	157	179	14.01

Hence, our preliminary analytical model handles only one type of update, i.e. updating a key value for a single tuple (*TID*), and since we are interested in comparing the results of the simulation with the analytical model, we only run the simulation for this one type of update.

The simulation is transaction driven where transactions are randomly generated. We have a parameter for the percentage of updates (or the percentage of searches). From Table 1, we see that the percentage of transactions that are updates vary from 10 to 15%. For a given update, a *TID* is chosen randomly. For searches, we have an additional parameter for the percentage of single-key searches (or the percentage of range key searches). To generate a range key search we randomly choose a number between some minimum and maximum range, e.g. 2–10 consecutive keys. In the simulation we held the percentage of range and single key transactions constant, i.e. 50% range and 50% single. The relation consists of 5000 tuples with 200 distinct key values. Each simulation was run for 400,000 transactions.

The results of the simulation and the analytical model are shown in Table 1. In the fourth row, we see that the model differs from the simulation by approx. 15% for the maximum differential file size. This is the largest difference exhibited between the results of the simulation and the analytical model. In the second row, we see that the model differs from the simulation by only 5%. So, we see that our analytical model is a good predictor of the maximum size of the differential file.

In Fig. 5, we show the change in the maximum differential file size as the number of transactions processed increases, for the five different simulations. The maximum differential file size is shown as a percentage of the number of tuples in the relation. We see for the simulation run with only 10% updates that the maximum differential file size is approx. 0.5% of the number of tuples in the relation, i.e. 29 records in the differential file 5000 tuples in the relation. For the largest percentage of updates, 50%, the maximum differential file size reaches approx. 3.1% of the number of tuples in the relation. In addition, the size of the differential file record is expected to be much smaller than the size of a tuple, e.g. 1/4 the size. Thus, the overall space used by the differential

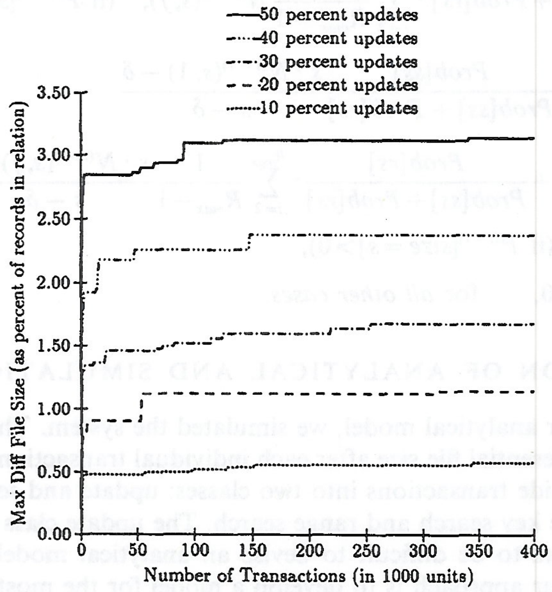

Fig. 5. Maximum differential file size vs transactions processed.

file would be reduced accordingly. Hence, with increasing amounts of main memory becoming available, the differential file would require only a modest amount of memory. By storing the differential file in main memory, we would not be penalized for any secondary storage accesses when searching the differential file.

6. CONCLUSIONS

In this paper, we have proposed a deferred and incremental index update strategy which uses a differential file. The performance of our scheme is relative to the size of the differential file. As such, we have devised an analytical model to predict the maximum differential file size as well as a simulation model to yield the maximum size. We have shown that the maximum size is not very large and that our simulation and analytical model results are close. As a point of future work, we need to extend our analytic model so that all four classes of updates can be modeled instead of just the current single *TID* update.

REFERENCES

[1] G. Sacco. Index Access with a finite buffer. *VLDB Conf. Proc.* 301–309 (1987).
[2] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price. Access path selection in a relational database management system. *SIGMOD Conf. Proc. ACM*, pp. 23–34 (1979).
[3] M. Schkolnick and P. Tiberio. Estimating the cost of updates in a relational database. *ACM TODS* **10**, 163–179 (1985).
[4] M. Stonebraker, E. Wong, P. Kreps and G. Held. The design and implementation of INGRES. *ACM TODS* **1**, 189–222 (1976).
[5] D. Severance and G. Lohman. Differential files: their application to the maintenance of large databases. *ACM TODS* **1**, 256–267 (1976).
[6] H. Aghili and D. Severance. A practical guide to the design of differential files for recovery of on-line databases. *ACM TODS* **7**, 540–565 (1982)
[7] J. Woodfill and M. Stonebraker. An implementation of hypothetical relations. *VLDB Conf. Proc.* pp. 157–166. (1983).
[8] S. Lang, J. Driscoll and J. Jou. Improving the differential file technique via batch operations for tree structured file organizations. *Data Engineering Conf. Proc. IEEE*, pp. 524–532 (1986).
[9] J. Scrivastava and C. Ramamoorthy. Efficient algorithms for maintenance of large database indexes. *Data Engineering Conf. Proc. IEEE*, pp. 402–408 (1988).
[10] S. Cammarata. Deferring updates in a relational database system. *VLDB Conf. Proc.*, pp. 286–292 (1981).
[11] P. Dadam, V. Lum, U. Praedel and G. Schlageter. Selective deferred index maintenance and concurrency control in integrated information systems. *VLDB Conf. Proc.*, pp. 142–149 (1985).
[12] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS ±. *IEEE Comput.* **19**, 19–25 (1986).
[13] E. Hanson. A performance analysis of view materialization strategies. *SIGMOD Conf. Proc. ACM*, pp. 440–453 (1987).
[14] K. Kinsley and C. Hughes. Evaluating database update schemes: a methodology and its application to distributive systems. *IEEE TSE* **14**, 1081–1089 (1988).
[15] S. Lavenberg. *Computer Systems Performance Modeling Handbook*. Prentice-Hall, Englewood Cliffs, NJ (1983).

APPENDIX

Lemma 1

According to the search and update algorithm, a given *TID* in the differential file, e.g. *TID i*, will appear in 0 or 1 *TID* lists in the index.

Proof

(By induction on the number of transactions.)

Basis. Before any transaction, *TID i* appears in a single *TID* list. This is trivially true since the tuple corresponding to *TID i* has one value associated with the key. In addition, the differential file is empty.

Induction. Assume that after $j-1$ transactions, *TID i* appears in 0 or 1 *TID* lists. We will show that after j transactions, *TID i* will appear in 0 or 1 *TID* lists.

Case I. TID i appears in 1 *TID* list after $j-1$ transactions. If the jth transaction is an update, then it does not change any *TID* list. If the jth transaction is a search then it might match the $old\text{-}key_i$ value or the $new\text{-}key_i$ value. A search for the $old\text{-}key_i$ value propagates a deletion of *TID i* from the *TID* list for $old\text{-}key_i$ and $old\text{-}key_i$ is set to $\varnothing$ for the differential file record. Thus, *TID i* now appears in 0 *TID* lists. A search for the $new\text{-}key_i$ value does not cause a change in any *TID* list. So, *TID i* would still appear in 1 *TID* list. Although, *TID i* would be returned with the *TID* list for $new\text{-}key_i$.

Case II. TID i appears in 0 *TID* lists after $j-1$ transactions. Once again, only a search can cause a modification to a *TID* list. The jth transaction cannot be a search for the $old\text{-}key_i$ value since the $old\text{-}key_i$ value indicates which *TID* list, *TID i* is currently contained in and for this case $old\text{-}key_i$ is equal to $\varnothing$. Hence, *TID i* will still appear in 0 *TID* lists. Therefore, the search must be for the $new\text{-}key_i$ value (if it applies to *TID i*). The search propagates an insertion of *TID i* into the *TID* list for $new\text{-}key_i$ and $new\text{-}key_i$, in the differential file record, is set to $\varnothing$. Hence, *TID i now appears in* 1 *TID* list.

Thus, after j transactions *TID i* will appear in 0 or 1 *TID* lists □

Theorem 1

The deferred update scheme presents a consistent view of the relation, regardless of the number of updates.

Proof

(By induction on the number of updates.)

Basis. If there is only 1 update for any particular *TID*, e.g. *TID i*, then Lemma 1 shows that subsequent searches yield a consistent view of the relation with respect to *TID i*. This applies for every *TID* that matches the search key.

Induction. Assume that after $j - 1$ updates, a search gives a consistent view of the relation. We will show that after j updates, a search will give a consistent view. We will look at the update associated with 1 *TID* but the same argument can be used for multiple *TIDs*. After $j - 1$ updates, a given record for *TID i* in the differential file will satisfy one of the following:

(a) $old\text{-}key_i \neq \varnothing$ and $new\text{-}key_i \neq \varnothing$
(b) $old\text{-}key_i = \varnothing$ and $new\text{-}key_i \neq \varnothing$.

In either of the above situations, the jth update would require changing $new\text{-}key_i$ to the value specified in the jth update. The $new\text{-}key_i$ value indicates where *TID i* should be inserted. It does not matter that we overwrite a previous update's value in a record in the differential file. Since the *TID* has not been stored in that *TID* list, there is no modification to do. In addition, we want to reflect the fact that *TID i* is now associated with this new value since the corresponding tuple in the relation already has this value stored. Thus, when a subsequent search is done for this new value, the *TIDs* in the index and differential file will correspond to the tuples that contain this new value. □